

Session 5

Relational Databases (SQL I)



NLAB:

Data at Scale

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

Store data in an arbitrary structure



Need arbitrary code to deconstruct object

Arbitrary code to construct new object



Relational databases

Potentially* need arbitrary code to deconstruct object the first time.

Store data in an a fixed "good" structure

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

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



Relational databases

Potentially* need arbitrary code to deconstruct object the first time.

Store data in an a fixed "good" structure

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

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 worlds" databases - week 6)

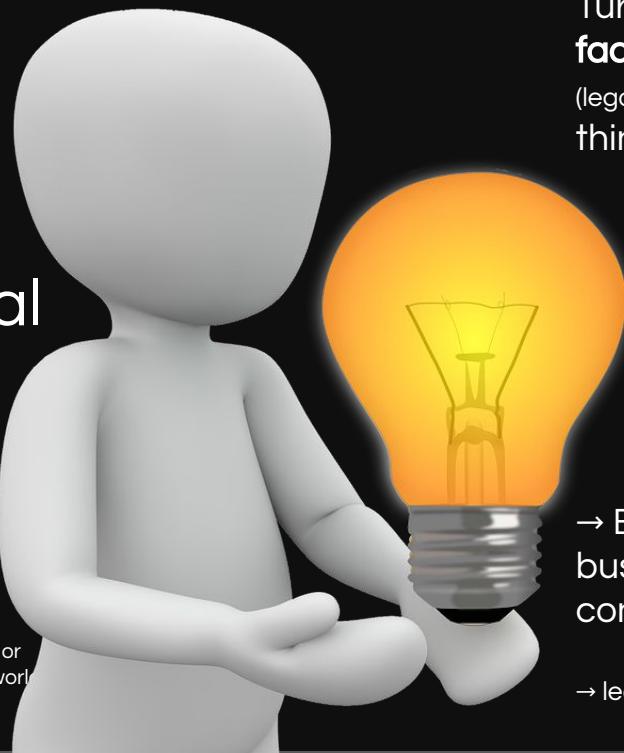
Turns out (rather than objects) **facts** make good basis (lego) pieces to construct things (i.e. cars, reports).

→ 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

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



Attributes:
properties of an entity

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

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

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

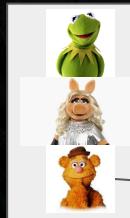
Let's be formal with our terminology...

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
<i>Miss piggy</i>	<i>pink</i>	<i>pig</i>
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.

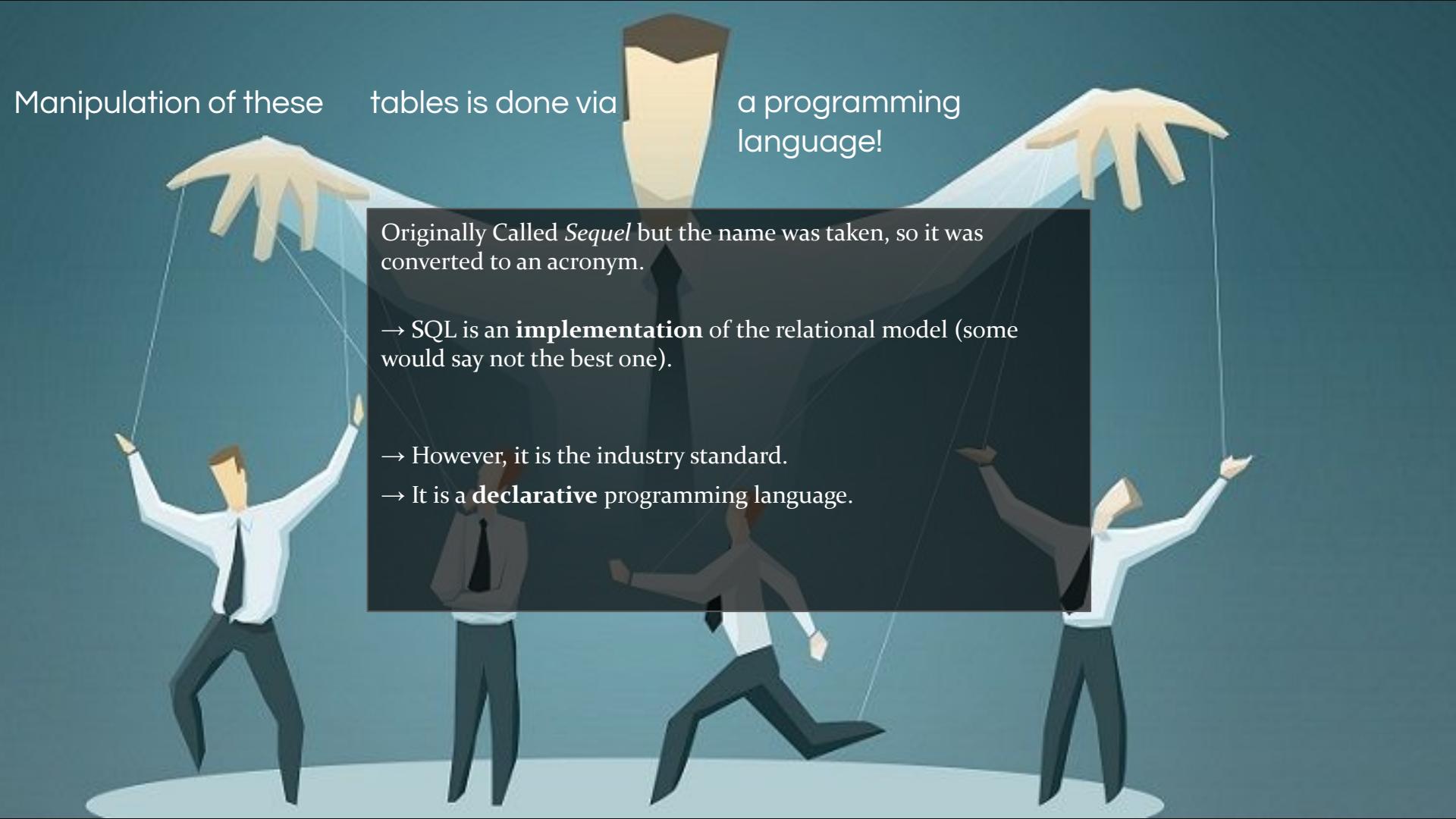
Entity **type** properties to be stored must be pre-specified and **values** entered for all entity occurrences.

The same facts must be "known" for all entities!

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

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

We'll talk about the consequences of NULL values later

A stylized illustration of a man in a suit being controlled by large yellow hands from above via thin white strings. He is looking up at the hands. In the background, other men are visible, one running and another looking up.

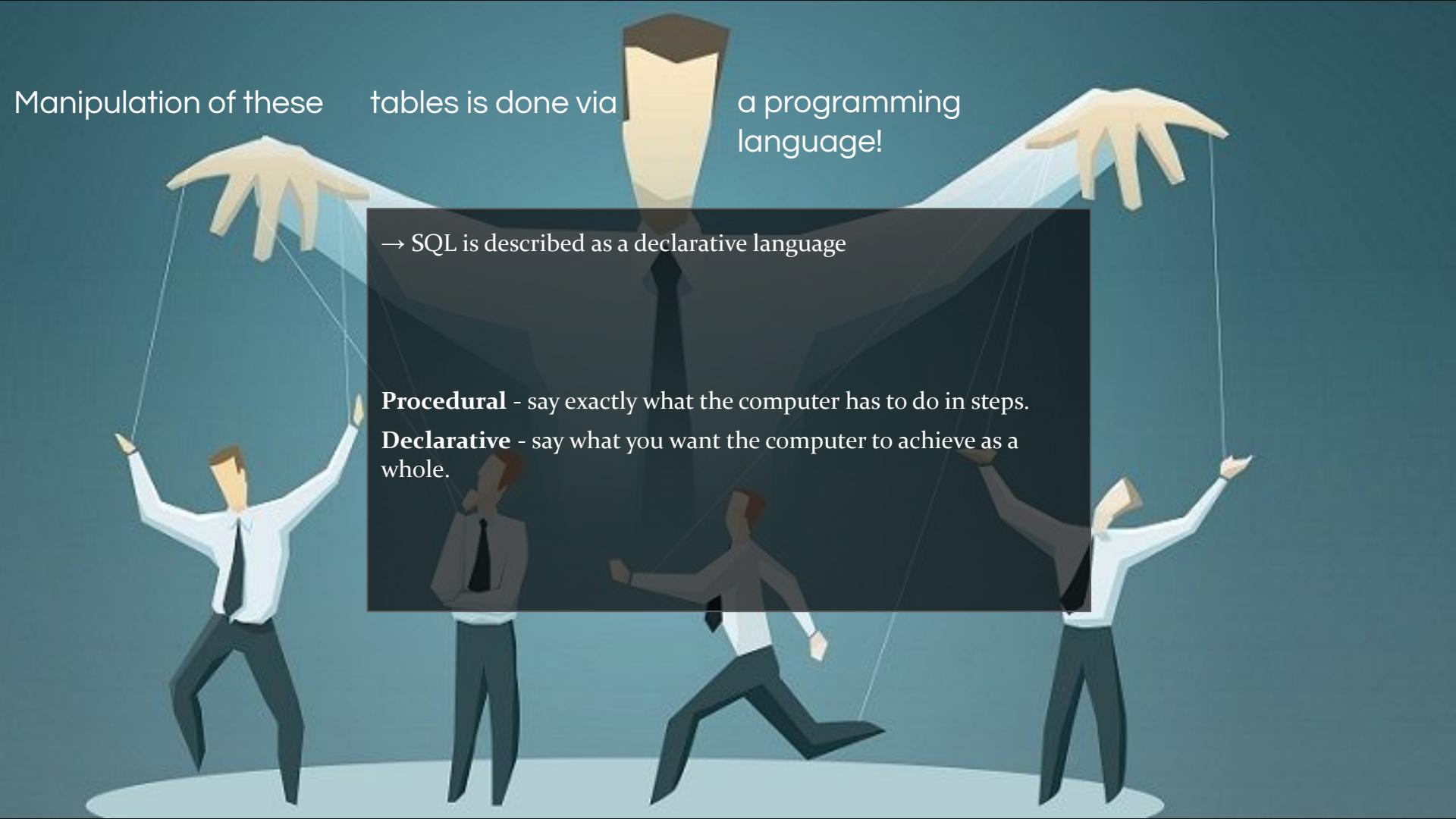
Manipulation of these tables is done via a programming language!

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...

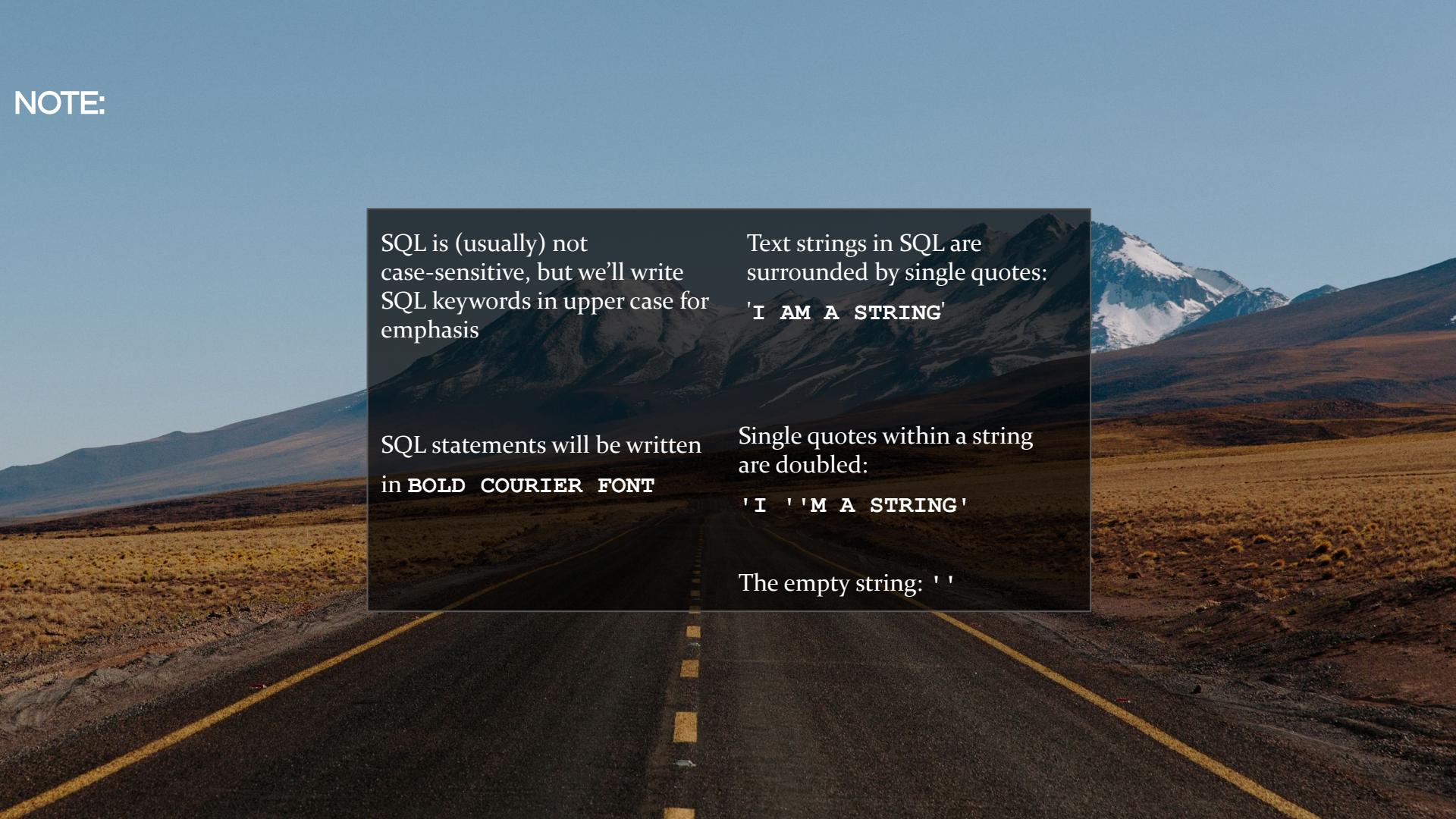
→ SQL only has a few basic commands.

→ 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...

Two types:

- Integer
- Floating-point

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

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**

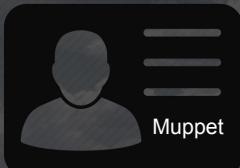
Numbers → use **INTEGER** or
NUMERIC

Temporal → use

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

SQL: CREATE TABLE

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



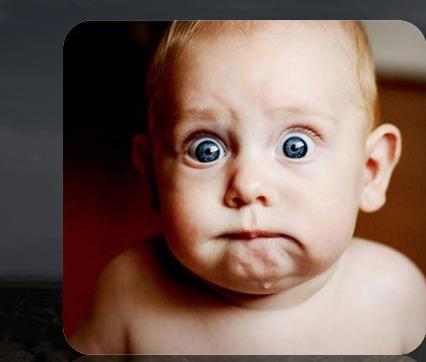
Name

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
```

student

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

```
UPDATE student  
SET year = year + 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
```

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

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

id
S103
S104
S107

SQL: WHERE EXERCISE FOR YOU

HINT:

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

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

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

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:

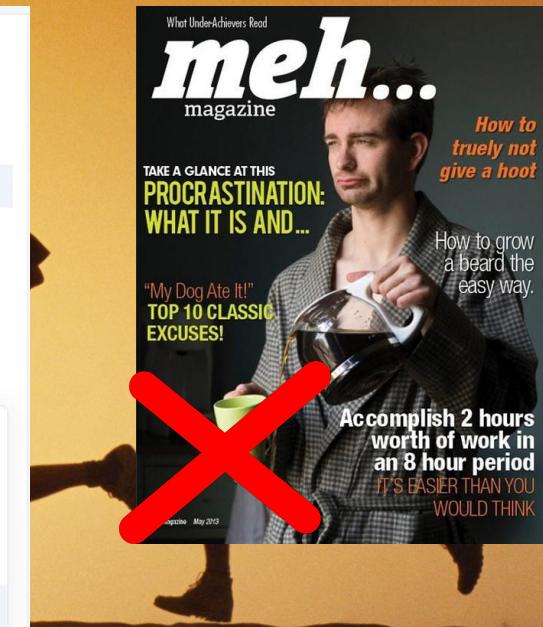
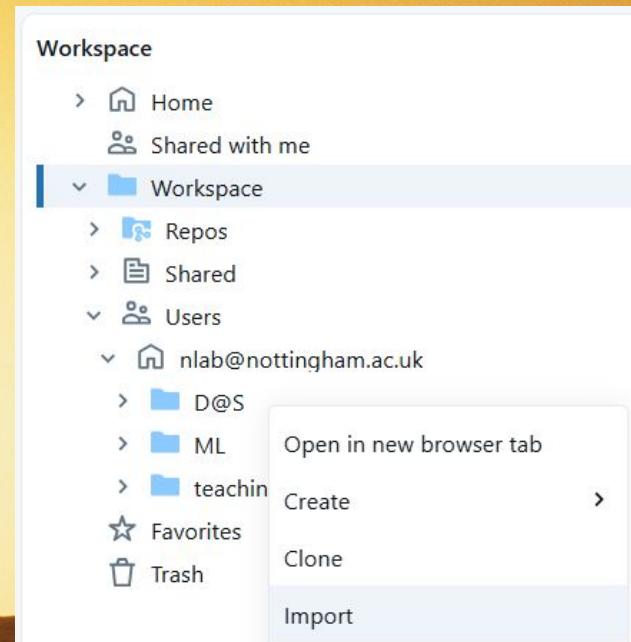
- What are the advantages of breaking entities into multiple table?



Time for our
morning exercise...

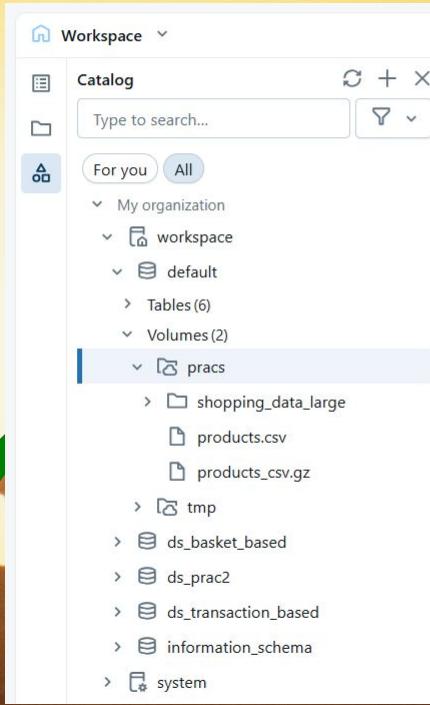


To run in-class examples:

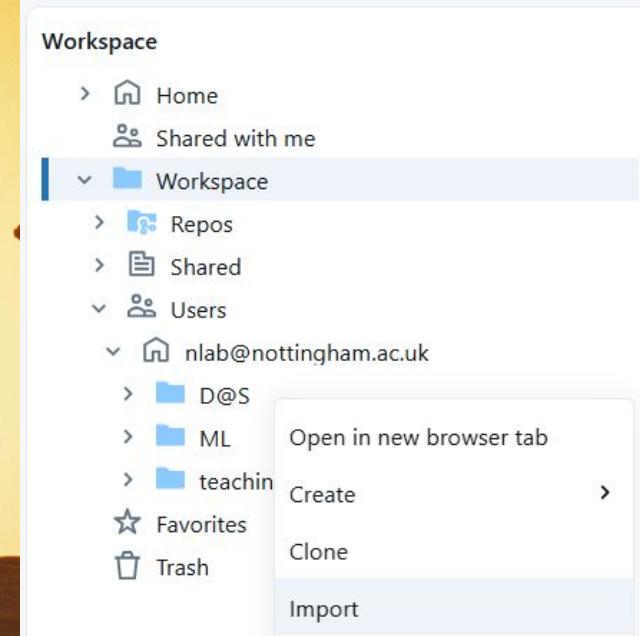


Time for our
morning exercise...

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



To access D@S Prac 2:



Session 6

Relational Databases (Database design for analysts)



Data at Scale

Dr Georgiana Nica-Avram

today.



Data at Scale

Dr Georgiana Nica-Avram

today.

- Overview of relational database design (awareness of process, not how to do it)
- What makes a good relational database design
- What issues you may face if you work with a poorly designed relational database
- ACID properties



Data at Scale

Dr Georgiana Nica-Avram

today.



Data at Scale

Dr Georgiana Nica-Avram

So... Hopefully I've now (or
at least after next week)
convinced you...

Facts provide good
building blocks since

→ Once data is in this
form it's "easy" to
manipulate



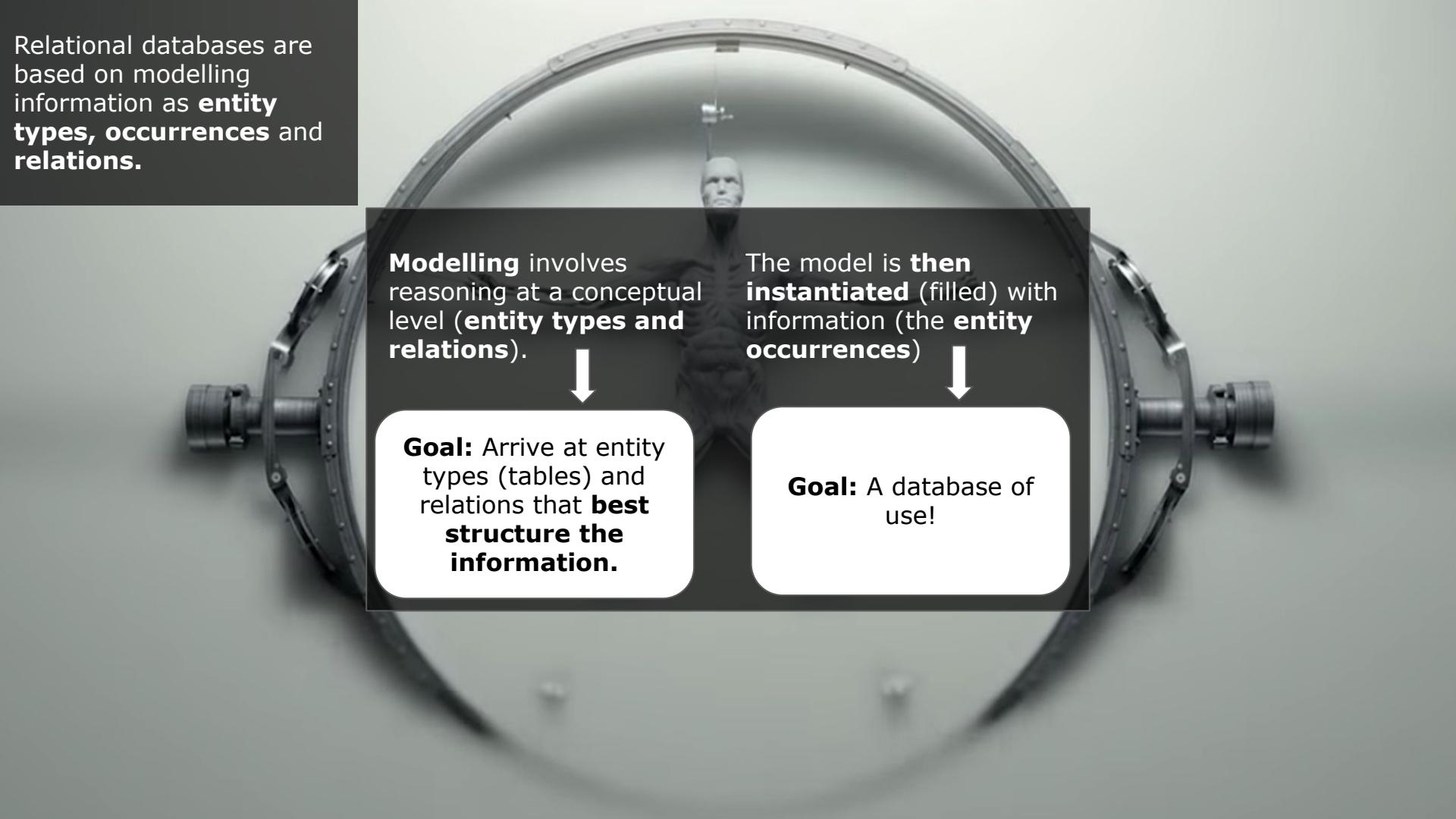
I now need to
convince you...

- That all information can be represented this way
- and show you the extra benefits the representation provides

We'll do this by providing an **overview** of how to **design databases**.

NOTE: *In-depth database design is outside of the scope of this course.*

Relational databases are based on modelling information as **entity types, occurrences** and **relations**.



Modelling involves reasoning at a conceptual level (**entity types and relations**).

The model is **then instantiated** (filled) with information (the **entity occurrences**)

Goal: Arrive at entity types (tables) and relations that **best structure the information.**

Goal: A database of use!

Goal: Arrive at entity types
(tables) and relations that
best structure the
information.

**Organising available
information** into common
structure: **groups of facts.**

Group of facts == Entity

Abstract thing, e.g idea

Event, e.g. purchase

Role, e.g chef, patient, customer

Personal info, e.g birth date,
gender

An interlude...

We are NOT identifying
and storing objects and
properties of objects!



This is Bob.

Where should Bob be
stored?

An interlude...

We are NOT identifying
and storing objects and
properties of objects!



This is Bob.

Where should Bob be
stored?



Object Database:
In a "Person Store" as
Bob (obviously)!!

An interlude...

We are NOT identifying and storing objects and properties of objects!



This is Bob.

Where should Bob be stored?



Object Database:
In a "Person Store" as Bob (obviously)!!



Relational Database:

But Bob is a Chef **and** a Patient.



The hospital doesn't care that Bob is a chef...

An interlude...

We are NOT identifying and storing objects and properties of objects!



This is Bob.

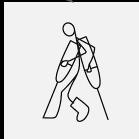
Where should Bob be stored?



Object Database:
In a "Person Store" as Bob (obviously)!!



Chef table



Patient table



Relational Database:

But Bob is a Chef **and** a Patient.

The hospital doesn't care that Bob is a chef...



Group information for a purpose:

Create "views" on Bob that can be linked back together if required.

Also helps prevent data duplication...

An interlude...

We are NOT identifying and storing objects and properties of objects!



This is Bob.

Where should Bob be stored?



Object Database:
In a "Person Store" as Bob (obviously)!!



Chef table



Patient table



Relational Database:

But Bob is a Chef **and** a Patient.

The hospital doesn't care that Bob is a chef...



Group information for a purpose:

Create "views" on Bob that can be linked back together if required.

Also helps prevent data duplication...



Dynamically create new "views" for different purposes via SQL.

More flexible modelling.

An interlude...

We are NOT identifying and storing objects and properties of objects!



This is Bob.

Where should Bob be stored?



Object Database:
In a "Person Store" as Bob (obviously)!!



Chef table



Patient table



Relational Database:

But Bob is a Chef **and** a Patient.

The hospital doesn't care that Bob is a chef...



Group information for a purpose:

Create "views" on Bob that can be linked back together if required.

Also helps prevent data duplication...



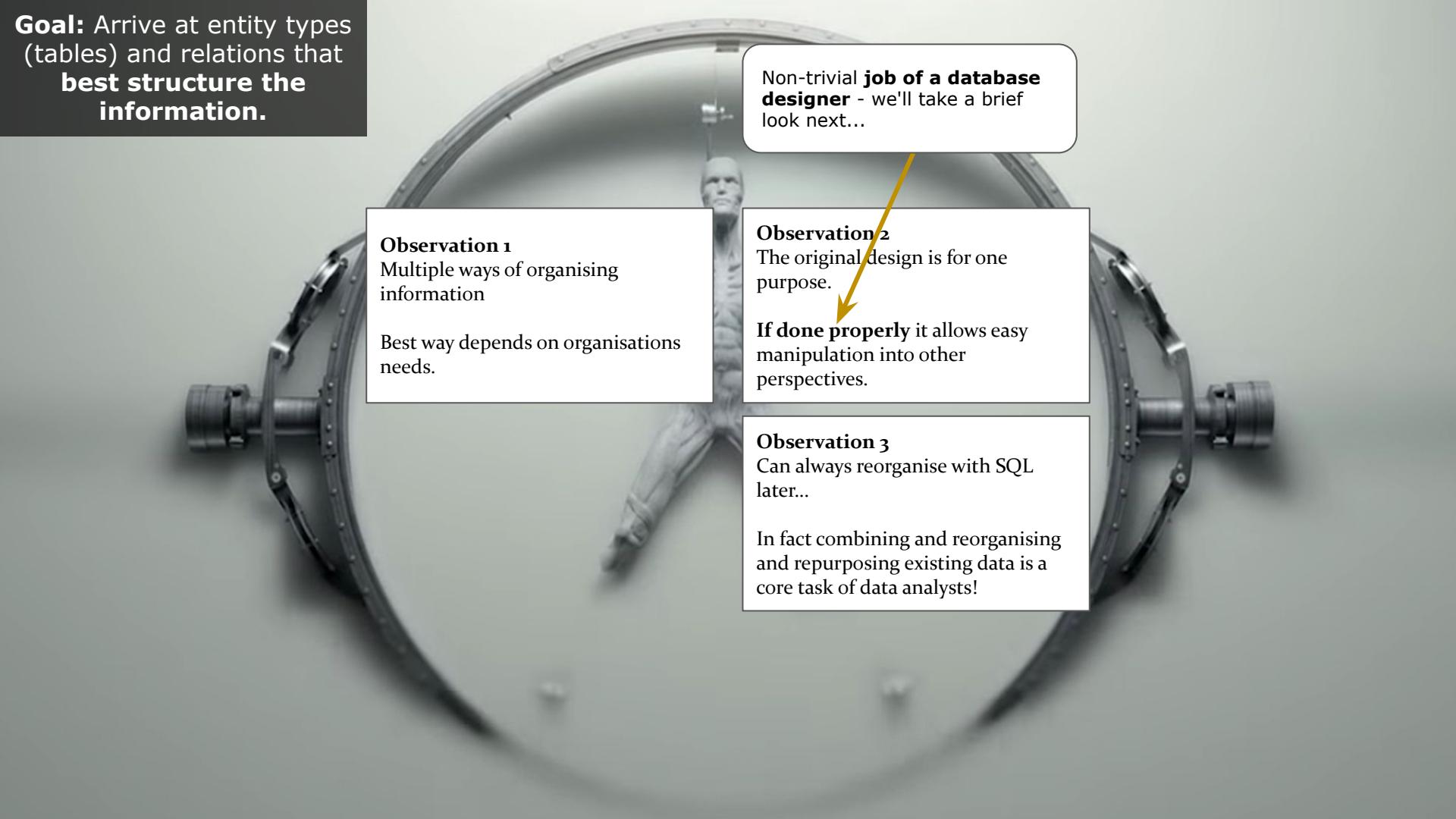
Dynamically create new "views" for different purposes via SQL.

More flexible modelling.

Added benefit:

Less arguments on the philosophy of what an object is and how it should be stored (is a laptop an object? to Lenovo? to Intel?)

Goal: Arrive at entity types
(tables) and relations that
best structure the
information.



Non-trivial **job of a database designer** - we'll take a brief look next...

Observation 1

Multiple ways of organising information

Best way depends on organisations needs.

Observation 2

The original design is for one purpose.

If done properly it allows easy manipulation into other perspectives.

Observation 3

Can always reorganise with SQL later...

In fact combining and reorganising and repurposing existing data is a core task of data analysts!

Designing databases.

Or modelling
information...

Draw a picture!

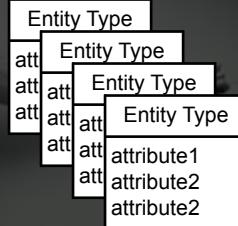
Repeat and refine,
identifying attributes
and cardinality of
relationships

Step 1: Group information into entity types (identify entities)

Step 2: Identify relationships

Step 3: Follow rules to make the entity types more atomic ("better lego pieces"). *Normalization*.

Step 4: Repeat until you (stakeholders) are happy or run out of time



1..1 relationship o..*

Relationships

One-to-one

Relationships between individual entities are **associations** (links) between one or more **entities** (rows) from one entity type (table) to one or more entities (rows) of another entity type (table).

Relationships between **entity types** (tables) describe these relationships generally.

Entity type: student

first	last	gender
Mark	Jones	M
Victoria	Smith	F
Jane	Doe	F

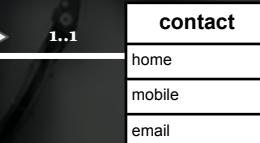
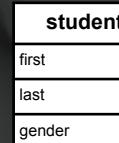
Entity type: contact

home	mobile	email
0115 84475	07586655810	m@x.com
0115 84786	07613399785	v@y.co.uk
0115 96375	07694411220	d@z.co.uk

student
first
last
gender

has

contact
home
mobile
email



Relationships

One-to-one

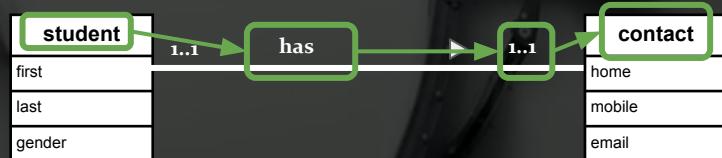
Relationships between individual entities are **associations** (links) between one or more **entities** (rows) from one entity type (table) to one or more entities (rows) of another entity type (table).

Relationships between **entity types** (tables) describe these relationships generally.

Entity type: student

first	last	gender	home	mobile	email
Mark	Jones	M	0115 84475	07586655810	m@x.com
Victoria	Smith	F	0115 84786	07613399785	v@y.co.uk
Jane	Doe	F	0115 96375	07694411220	d@z.co.uk

Entity type: contact



Relationships

One-to-one

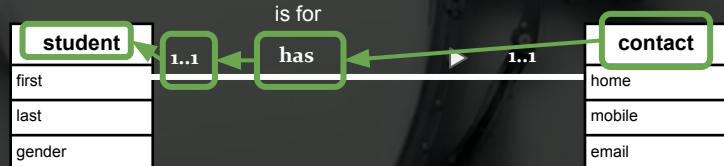
Relationships between individual entities are **associations** (links) between one or more **entities** (rows) from one entity type (table) to one or more entities (rows) of another entity type (table).

Relationships between **entity types** (tables) describe these relationships generally.

Entity type: student

first	last	gender	home	mobile	email
Mark	Jones	M	0115 84475	07586655810	m@x.com
Victoria	Smith	F	0115 84786	07613399785	v@y.co.uk
Jane	Doe	F	0115 96375	07694411220	d@z.co.uk

Entity type: contact



Relationships

One-to-many

Relationships between individual entities are **associations** (links) between one or more **entities** (rows) from one entity type (table) to one or more entities (rows) of another entity type (table).

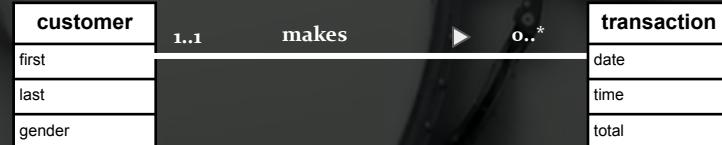
Relationships between **entity types** (tables) describe these relationships generally.

Entity type: customer

first	last	gender
Mark	Jones	M
Victoria	Smith	F
Jane	Doe	F

Entity type: transaction

date	time	total
2017-08-09	08:05:32	£12.00
2017-08-10	13:00:00	£6.50
2017-08-10	11:00:01	£3.00



Relationships

Many-to-many

Relationships between individual entities are **associations** (links) between one or more **entities** (rows) from one entity type (table) to one or more entities (rows) of another entity type (table).

Relationships between **entity types** (tables) describe these relationships generally.

Entity type: basket

time	loyalty_id	total
11am	7896	£3.70
1pm	9934	£3.25
2pm	3900	£0.07

Entity type: items

desc.	cost
Doritos	£3.00
Chocolate bar	£0.70
Sandwich	£3.25

basket
time
loyalty_id
total

contains

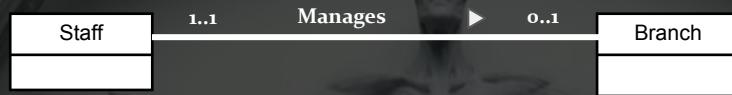
o..*

items
desc.
cost

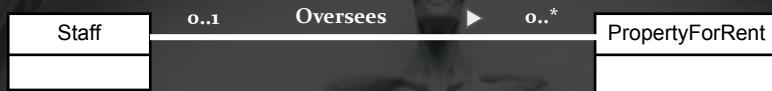
1..*

NOTE: An item can appear in multiple baskets as in the item table instances denote the generic item type. If item rows defined a specific physical item then the relationship would be one-to-many.

Some textbook examples...



Some textbook examples...



Some textbook examples...



Call Detail Record (CDR) Data

(More event data...)



Calls



start time
tower id
caller id
called id
tariff type
prepaid balance
product id
...

duration
charge
serviceflow
roaming
result code
incoming
...

SMS



start time
tower id
caller id
called id
tariff type
prepaid balance
product id
...

charge
serviceflow
roaming
result code
incoming
sms length
...

Data

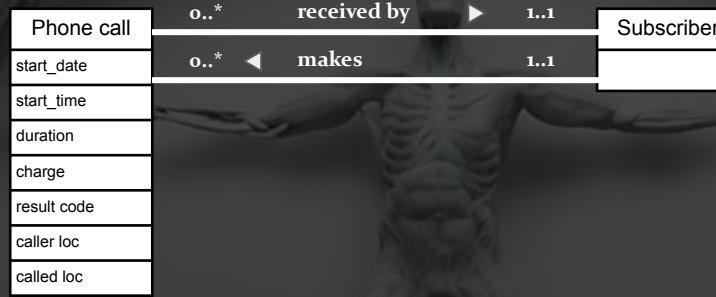


start time
tower id
caller id
tariff type
prepaid balance
charge
...

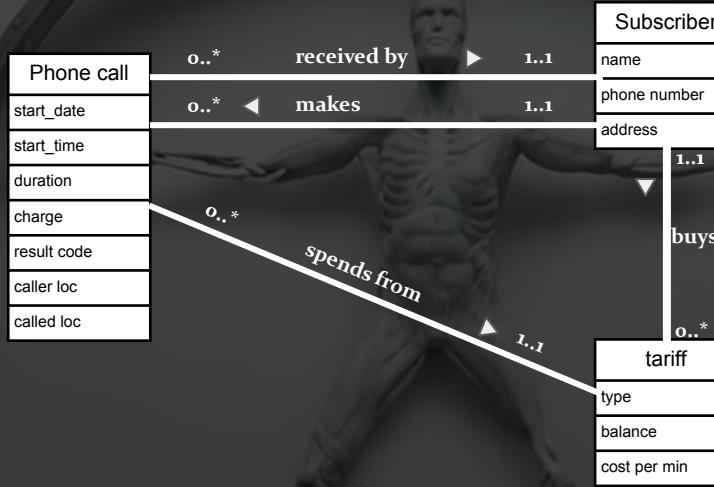
product id
duration
data up
data down
...

start time
tower id
caller id
called id
tariff type
prepaid balance
product id

duration
charge
roaming
result code
...



start time
tower id
caller id
called id
tariff type
prepaid balance
product id
...
duration
charge
roaming
result code

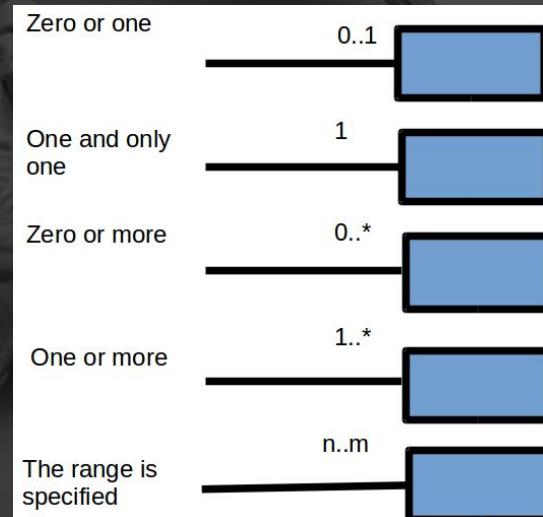


Nota bene

Among other functions, Unified Modeling Language (UML) is used to visualise, specify and document relationships between entity types (tables).

Relationships are expressed in terms of cardinality:

- one to one
- one to many
- many to many



Keys



In order **to utilise relationships** to manipulate data we need to be able to **uniquely identify entity occurrences**.

Keys:

Do not need to be IDs

(but very often are..)

Candidate key: The minimal set of attributes that uniquely identifies each occurrence of an entity type.

Entity type: basket

staff_id	date	time	shop	till	total
6	2017-01-01	11:00:00	123	4	£3.00
78	2017-01-02	12:00:00	849	3	£2.15
9	2017-01-02	13:00:01	837	2	£15.00

Primary key: The candidate key selected to uniquely identify each entity occurrence.

Keys:

Do not need to be IDs

(but very often are..)

Candidate key: The minimal set of attributes that uniquely identifies each occurrence of an entity type.

Primary key: The candidate key selected to uniquely identify each entity occurrence.

Entity type: basket

staff_id	date	time	shop	till	total
6	2017-01-01	11:00:00	123	4	£3.00
78	2017-01-02	12:00:00	849	3	£2.15
9	2017-01-02	13:00:01	837	2	£15.00

Entity type: basket

staff_id	date	time	shop	till	total
6	2017-01-01	11:00:00	123	4	£3.00
78	2017-01-02	12:00:00	849	3	£2.15
9	2017-01-02	13:00:01	837	2	£15.00

Keys should have:

- minimal attributes
- temporal invariance
- future certainty of uniqueness

Keys:

Do not need to be IDs

(but very often are..)

Candidate key: The minimal set of attributes that uniquely identifies each occurrence of an entity type.

Primary key: The candidate key selected to uniquely identify each entity occurrence.

Entity type: basket

staff_id	date	time	shop	till	total
6	2017-01-01	11:00:00	123	4	£3.00
78	2017-01-02	12:00:00	849	3	£2.15
9	2017-01-02	13:00:01	837	2	£15.00

Entity type: basket

staff_id	date	time	shop	till	total
6	2017-01-01	11:00:00	123	4	£3.00
78	2017-01-02	12:00:00	849	3	£2.15
9	2017-01-02	13:00:01	837	2	£15.00

Keys should have:

- minimal attributes
- temporal invariance
- future certainty of uniqueness

Minimize data redundancy
(more soon)

Keys:

Do not need to be IDs

(but very often are..)

Candidate key: The minimal set of attributes that uniquely identifies each occurrence of an entity type.

Primary key: The candidate key selected to uniquely identify each entity occurrence.

Entity type: basket

staff_id	date	time	shop	till	total
6	2017-01-01	11:00:00	123	4	£3.00
78	2017-01-02	12:00:00	849	3	£2.15
9	2017-01-02	13:00:01	837	2	£15.00

Entity type: basket

staff_id	date	time	shop	till	total
6	2017-01-01	11:00:00	123	4	£3.00
78	2017-01-02	12:00:00	849	3	£2.15
9	2017-01-02	13:00:01	837	2	£15.00

Keys should have:

- minimal attributes
- temporal invariance
- future certainty of uniqueness



Required to prevent errors

Keys:

Do not need to be IDs

(but very often are..)

Candidate key: The minimal set of attributes that uniquely identifies each occurrence of an entity type.

Primary key: The candidate key selected to uniquely identify each entity occurrence.

Entity type: basket

staff_id	date	time	shop	till	total
6	2017-01-01	11:00:00	123	4	£3.00
78	2017-01-02	12:00:00	849	3	£2.15
9	2017-01-02	13:00:01	837	2	£15.00

Entity type: basket

staff_id	date	time	shop	till	total
6	2017-01-01	11:00:00	123	4	£3.00
78	2017-01-02	12:00:00	849	3	£2.15
9	2017-01-02	13:00:01	837	2	£15.00

Keys should have:

- minimal attributes
- temporal invariance
- future certainty of uniqueness

Often this means artificially creating an ID for this purpose.

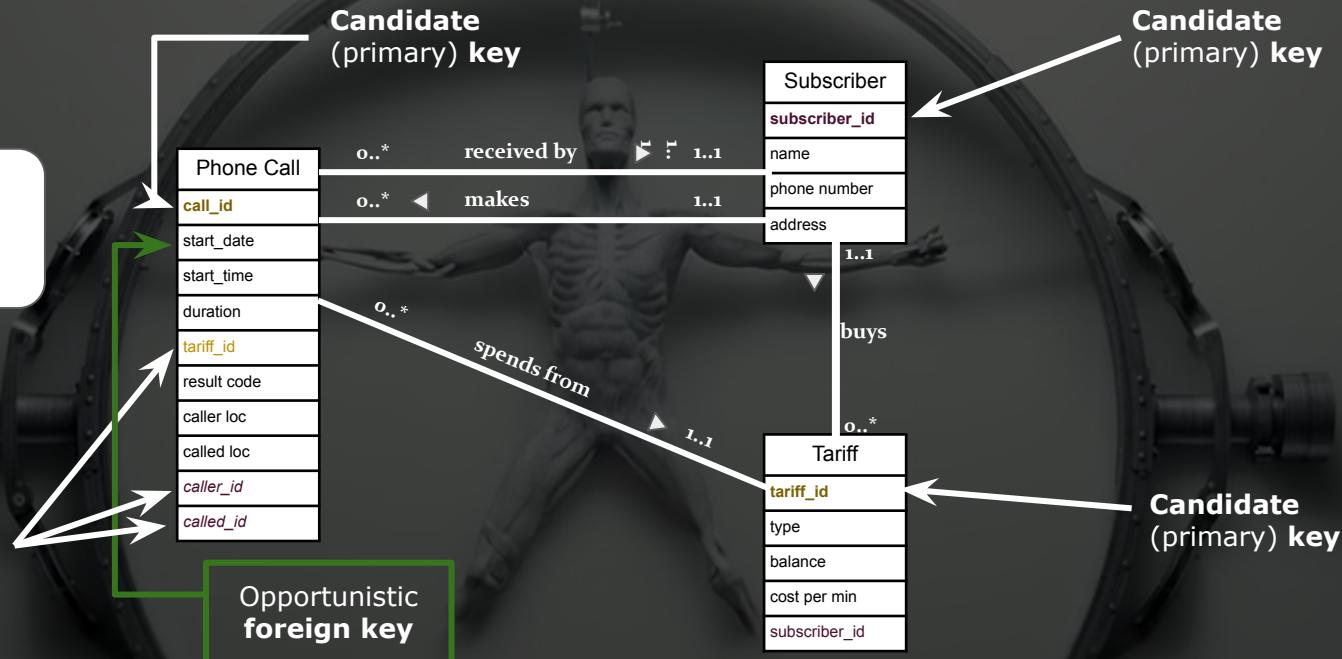
How are they used?

Implicitly define relationships!!

"keys should have minimal attributes"

→ minimise data redundancy

Foreign key:
attribute (or set of)
that uniquely
identifies a row in
another table



How are they used?

Implicitly define relationships!!

What you need to know:

- 1) Entity types (tables) are linked together by relationships.
- 2) Relationships are implicitly defined through **candidate** and **foreign keys**.
- 3) These can exist **by design** or **opportunistically**.
- 4) How to read entity relationship diagrams

Subscriber
subscriber_id

Candidate
(primary) key

Candidate
(primary) key

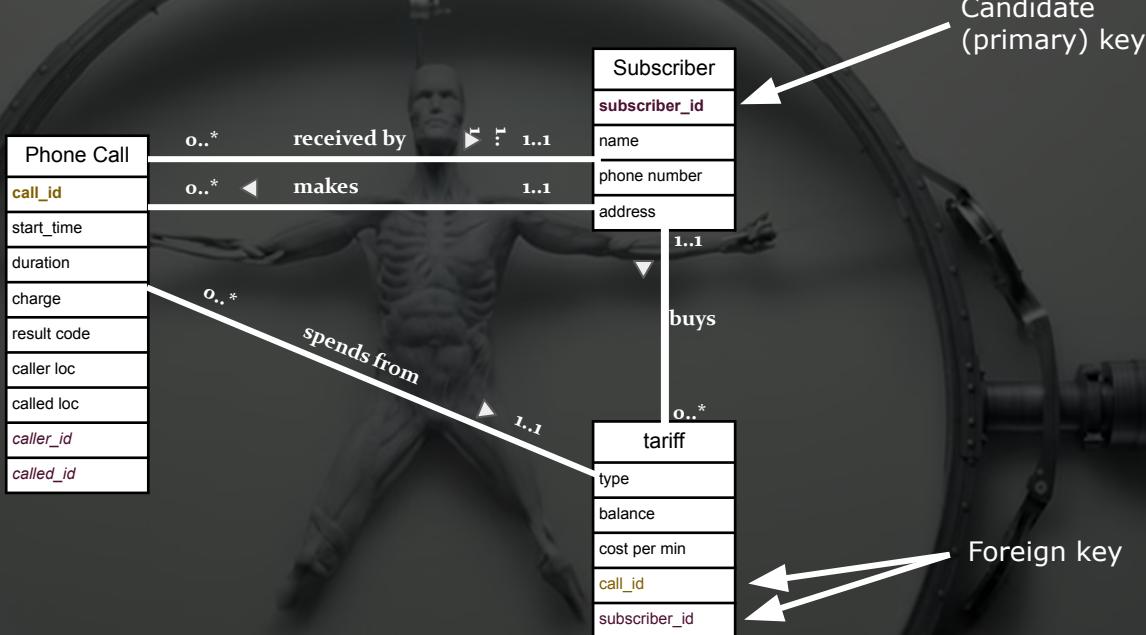
cost per min
call_id
subscriber_id

Foreign key

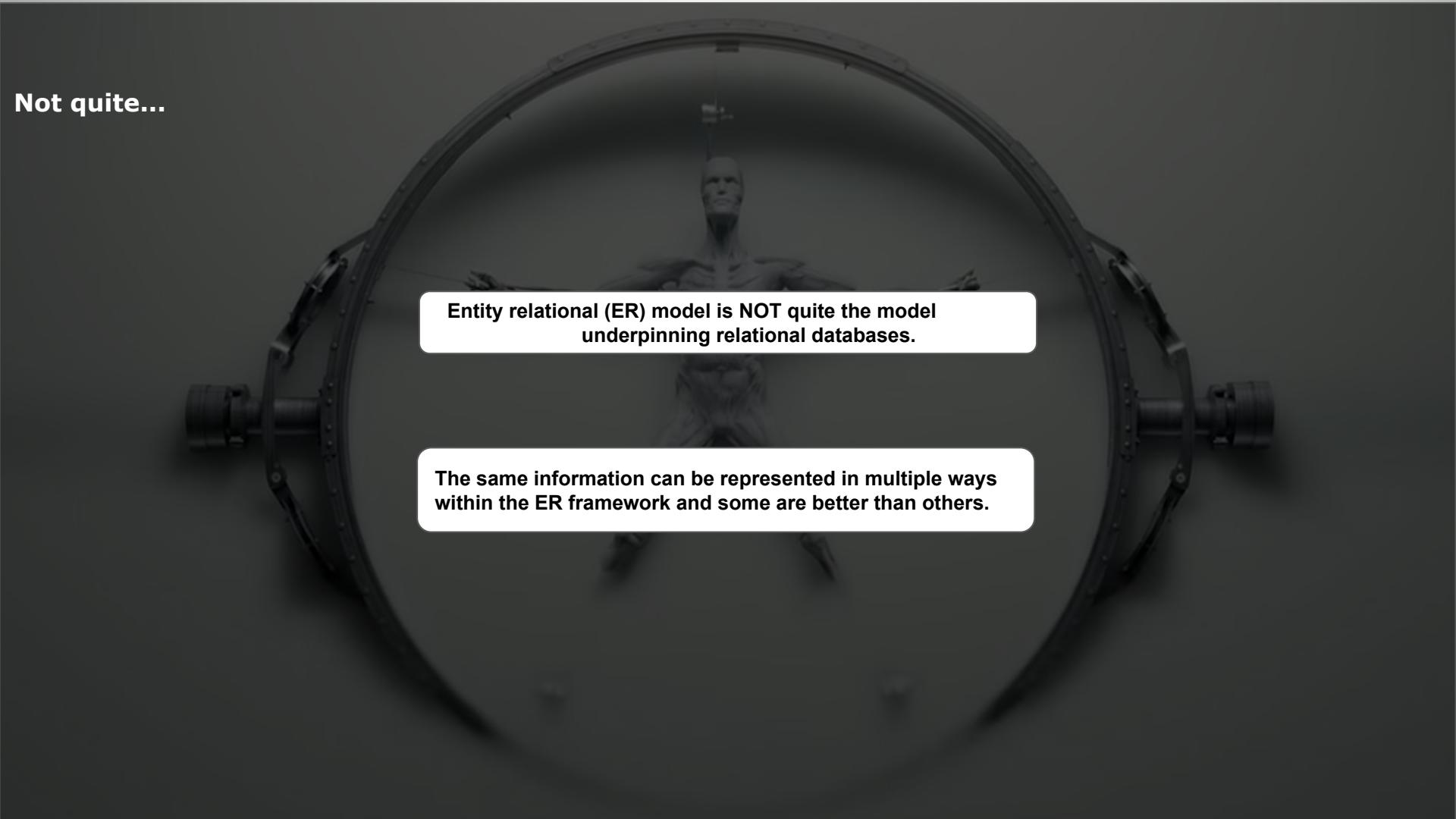


Hurray! We now have a model of the world we can use!

Candidate (primary) key

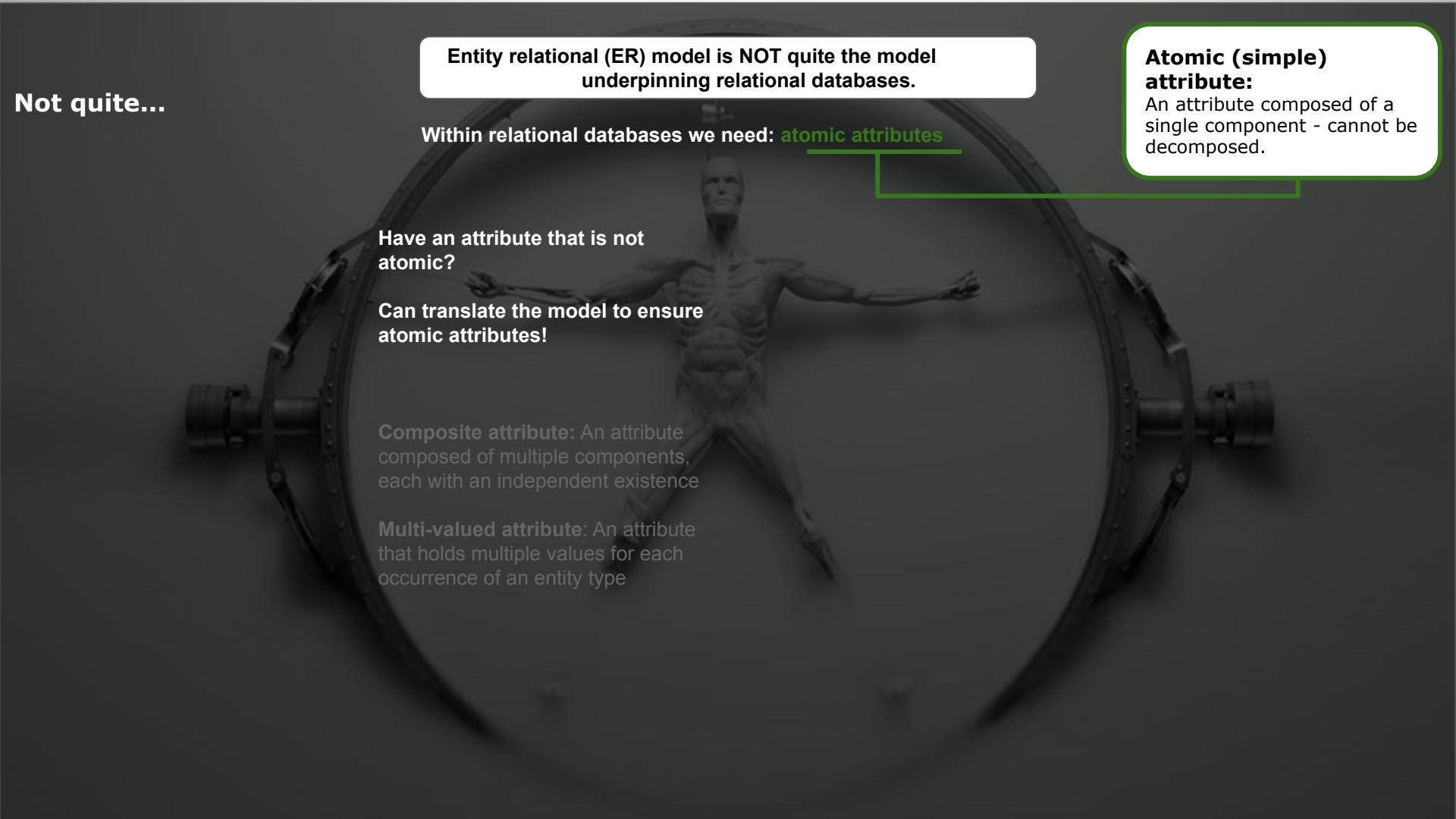


Not quite...



Entity relational (ER) model is NOT quite the model
underpinning relational databases.

The same information can be represented in multiple ways
within the ER framework and some are better than others.



Atomic (simple) attribute:

An attribute composed of a single component - cannot be decomposed.

Entity relational (ER) model is NOT quite the model underpinning relational databases.

Within relational databases we need: **atomic attributes**

Have an attribute that is not atomic?

Can translate the model to ensure atomic attributes!

Composite attribute: An attribute composed of multiple components, each with an independent existence

Multi-valued attribute: An attribute that holds multiple values for each occurrence of an entity type

Not quite...

Entity relational (ER) model is NOT quite the model
Underpinning relational databases.

Not quite...

Within relational databases we need: **atomic attributes**

Have an attribute that is not
atomic?

Can translate the model to ensure
atomic attributes!

Composite attribute: An attribute
composed of multiple components,
each with an independent exists

Multi-valued attribute: An attribute
that holds multiple values for each
occurrence of an entity type

Subscriber
subscriber_id
name
phone number
address

might* change to

*If we want to reason about
address parts in SQL.

Subscriber
subscriber_id
name
phone number
house_number
street
town
postcode
country

Entity relational (ER) model is NOT quite the model underpinning relational databases.

Not quite...

Within relational databases we need: **atomic attributes**

Have an attribute that is not atomic?

Can translate the model to ensure atomic attributes!

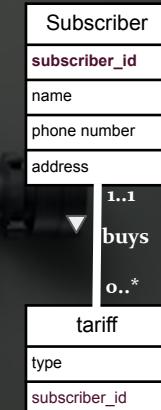
Composite attribute: An attribute composed of multiple components, each with an independent exists

Multi-valued attribute: An attribute that holds multiple values for each occurrence of an entity type

Subscriber
subscriber_id
name
phone number
address
tariffs

would have* to change to

*If we want to reason about tariffs in SQL.



Entity relational (ER) model is NOT quite the model underpinning relational databases.

Not quite...

Within relational databases we need: **atomic attributes**

Have an attribute that is not atomic

Can't have attributes that are

What you need to know:

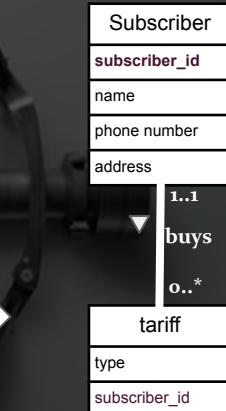
- 1) Entity types (tables) only have atomic attributes
- 2) **This does not limit what can be modelled.**

Multi-

that holds multiple values for each occurrence of an entity type

phone number
address
tariffs

*If we want to reason about tariffs in SQL.





The same information can be represented in multiple ways within the ER framework and some are better than others.

Not quite...

Good database design has minimal data redundancy

Redundant data

- Is data that already exists elsewhere in the database
- Redundant data leads to various subtle, but important problems:
 - INSERT anomalies
 - UPDATE anomalies
 - DELETE anomalies

Good database design has minimal data redundancy

Real world example of one transaction (basket) being paid for by **card + cash**

Entity Type: till_payments

basket_id	till_id	store	total	amount paid	credit card num
B1	T12	Nottingham	£30.00	£20.00	C1
B1	T12	Nottingham	£30.00	£10.00	C2
B2	T11	Nottingham	£15.00	£5.00	C1
B2	T11	Nottingham	£15.00	£10.00	C2
B3	T33	York	£2.00	£2.00	C4
B4	T41	Bath	£60.00	£55.00	C5
B4	T41	Bath	£60.00	£5.00	C6
B5	T51	Bath	£44.00	£44.00	C7

- **INSERT anomalies**
Can't add a basket with no credit card number (without NULLs)

For each basket we must correctly enter both the till_id and store even though the store can only have one value once the till_id is known.

- **UPDATE anomalies**
To change store for T12, we have to change two rows

- **DELETE anomalies**
If we remove B3, we remove any trace of the York store as well

Good database design has minimal data redundancy

Entity Type: till_payments

basket_id	till_id	store	total	amount paid	credit card num
B1	T12	Nottingham	£30.00	£20.00	C1
B1	T12	Nottingham	£30.00	£10.00	C2
B2	T11	Nottingham	£15.00	£5.00	C1
B2	T11	Nottingham	£15.00	£10.00	C2
B3	T33	York	£2.00	£2.00	C4
B4	T41	Bath	£60.00	£55.00	C5
B4	T41	Bath	£60.00	£5.00	C6
B5	T51	Bath	£44.00	£44.00	C7

- **INSERT anomalies**

Can't add a basket with no credit card number (without NULLs)

For each basket we must correctly enter both the till_id and store even though they always appear together.
- **UPDATE anomalies**

To change store for T12, we have to change two rows
- **DELETE anomalies**

If we remove B3, we remove any trace of the York store as well

Good database design has minimal data redundancy

Entity Type: till_payments

basket_id	till_id	store	total	amount paid	credit card num
B1	T12	Nottingham	£30.00	£20.00	C1
B1	T12	Nottingham	£30.00	£10.00	C2
B2	T11	Nottingham	£15.00	£5.00	C1
B2	T11	Nottingham	£15.00	£10.00	C2
B3	T33	York	£2.00	£2.00	C4
B4	T41	Bath	£60.00	£55.00	C5
B4	T41	Bath	£60.00	£5.00	C6
B5	T51	Bath	£44.00	£44.00	C7

- **INSERT anomalies**
Can't add a basket with no credit card number (without NULLs)

For each basket we must correctly enter both the till_id and store even though they always appear together.
- **UPDATE anomalies**
To change store for T12, we have to change two rows
- **DELETE anomalies**
If we remove B3, we remove any trace of the York store as well

Good database design has minimal data redundancy

Entity Type: till_payments

basket_id	till_id	store	total	amount paid	credit card num
B1	T12	Nottingham	£30.00	£20.00	C1
B1	T12	Nottingham	£30.00	£10.00	C2
B2	T11	Nottingham	£15.00	£5.00	C1
B2	T11	Nottingham	£15.00	£10.00	C2
B3	T33	York	£2.00	£2.00	C4
B4	T41	Bath	£60.00	£55.00	C5
B4	T41	Bath	£60.00	£5.00	C6
B5	T51	Bath	£44.00	£44.00	C7

Bad news: Identifying entities and relationships is not enough :(

Good news: Maths underpinning = rules to do this. Equivalent representation.

Bad news: It's complicated, based on analysing functional dependencies between attributes.

(i.e. if you know the till_id you know the store)

Good news: This is the role of database designers not business analytics.

Good news: We get to avoid these issues by design.

**For fun let's
convince
ourselves this
is true...**

Entity Type: till_payments

basket_id	till_id	store	total	amount paid	credit card num
B1	T12	Nottingham	£30.00	£20.00	C1
B1	T12	Nottingham	£30.00	£10.00	C2
B2	T11	Nottingham	£15.00	£5.00	C1
B2	T11	Nottingham	£15.00	£10.00	C2
B3	T33	York	£2.00	£2.00	C4
B4	T41	Bath	£60.00	£55.00	C5
B4	T41	Bath	£60.00	£5.00	C6
B5	T51	Bath	£44.00	£44.00	C7

Entity Type: baskets

basket_id	till_id	total
B1	T12	£30.00
B2	T11	£15.00
B3	T33	£2.00
B4	T41	£60.00
B5	T51	£44.00

Entity Type: tills

till_id	store
T12	Nottingham
T11	Nottingham
T33	York
T41	Bath
T51	Bath



Entity Type: credit card transactions

basket_id	amount paid	credit card num
B1	£20.00	C1
B1	£10.00	C2
B2	£5.00	C1
B2	£10.00	C2
B3	£2.00	C4
B4	£55.00	C5
B4	£5.00	C6
B5	£44.00	C7

- **INSERT anomalies**

Can't add a basket with no credit card number (without NULLs)

For each basket we must correctly enter both the till_id and store even though the store can only have one value once the till_id is known.

- **UPDATE anomalies**

To change store for T12, we have to change two rows

- **DELETE anomalies**

If we remove B3, we remove any trace of the York store as well

**For fun let's
convince
ourselves this
is true...**

Entity Type: till_payments

basket_id	till_id	store	total	amount paid	credit card num
B1	T12	Nottingham	£30.00	£20.00	C1
B1	T12	Nottingham	£30.00	£10.00	C2
B2	T11	Nottingham	£15.00	£5.00	C1
B2	T11	Nottingham	£15.00	£10.00	C2
B3	T33	York	£2.00	£2.00	C4
B4	T41	Bath	£60.00	£55.00	C5
B4	T41	Bath	£60.00	£5.00	C6
B5	T51	Bath	£44.00	£44.00	C7

Entity Type: baskets

basket_id	till_id	total
B1	T12	£30.00
B2	T11	£15.00
B3	T33	£2.00
B4	T41	£60.00
B5	T51	£44.00

Entity Type: tills

till_id	store
T12	Nottingham
T11	Nottingham
T33	York
T41	Bath
T51	Bath

Entity Type: credit card transactions

basket_id	amount paid	credit card num
B1	£20.00	C1
B1	£10.00	C2
B2	£5.00	C1
B2	£10.00	C2
B3	£2.00	C4
B4	£55.00	C5
B4	£5.00	C6
B5	£44.00	C7



- **INSERT anomalies**

Can't add a basket with no credit card number (without NULLs)

For each basket we must correctly enter both the till_id and store even though they always appear together.

- **UPDATE anomalies**

To change store for T12, we have to change two rows

- **DELETE anomalies**

If we remove B3, we remove any trace of the York store as well

**For fun let's
convince
ourselves this
is true...**

Entity Type: till_payments

basket_id	till_id	store	total	amount paid	credit card num
B1	T12	Nottingham	£30.00	£20.00	C1
B1	T12	Nottingham	£30.00	£10.00	C2
B2	T11	Nottingham	£15.00	£5.00	C1
B2	T11	Nottingham	£15.00	£10.00	C2
B3	T33	York	£2.00	£2.00	C4
B4	T41	Bath	£60.00	£55.00	C5
B4	T41	Bath	£60.00	£5.00	C6
B5	T51	Bath	£44.00	£44.00	C7

Entity Type: baskets

basket_id	till_id	total
B1	T12	£30.00
B2	T11	£15.00
B3	T33	£2.00
B4	T41	£60.00
B5	T51	£44.00

Entity Type: tills

till_id	store
T12	Nottingham
T11	Nottingham
T33	York
T41	Bath
T51	Bath

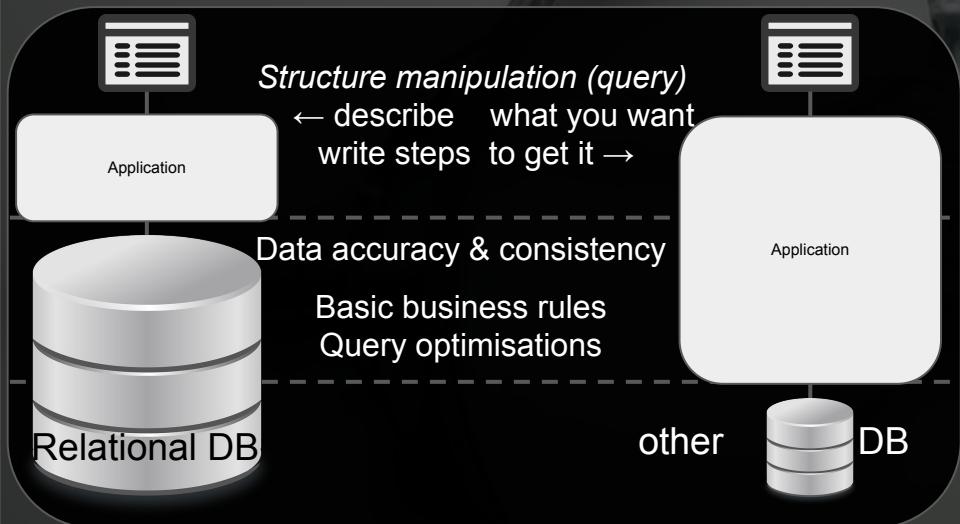
Entity Type: credit card transactions

basket_id	amount paid	credit card num
B1	£20.00	C1
B1	£10.00	C2
B2	£5.00	C1
B2	£10.00	C2
B3	£2.00	C4
B4	£55.00	C5
B4	£5.00	C6
B5	£44.00	C7



- **INSERT anomalies**
Can't add a basket with no credit card number (without NULLs)
- For each basket we must correctly enter both the till_id and store even though they always appear together.
- **UPDATE anomalies**
To change store for T12, we have to change two rows
- **DELETE anomalies**
If we remove B3, we remove any trace of the York store as well

What if we don't have this by design?



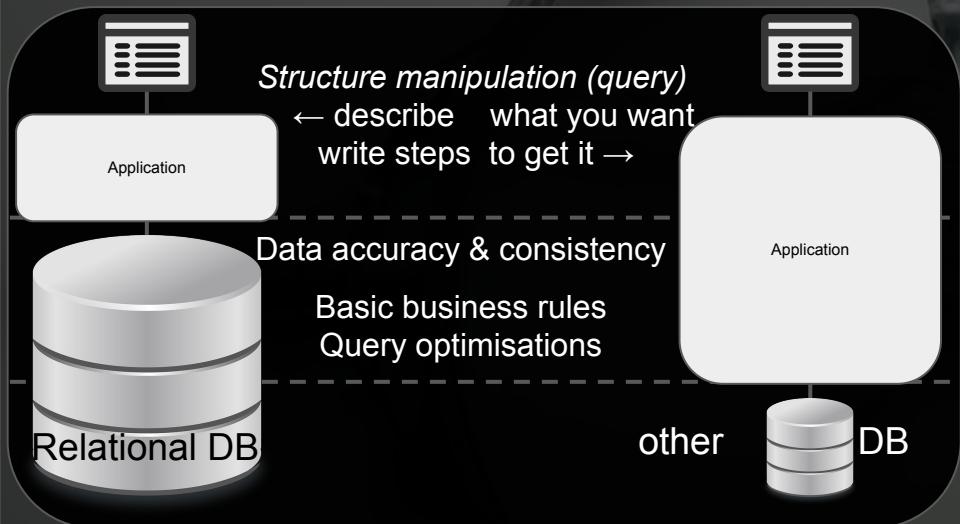
In this form we must write code to enable statistics (and business logic) with NULLs....

- **INSERT anomalies**
Can't add a basket with no credit card number (without NULLs)

For each basket we must correctly enter both the till_id and store even though they always appear together.

- **UPDATE anomalies**
To change store for T12, we have to change two rows
- **DELETE anomalies**
If we remove B3, we remove any trace of the York store as well

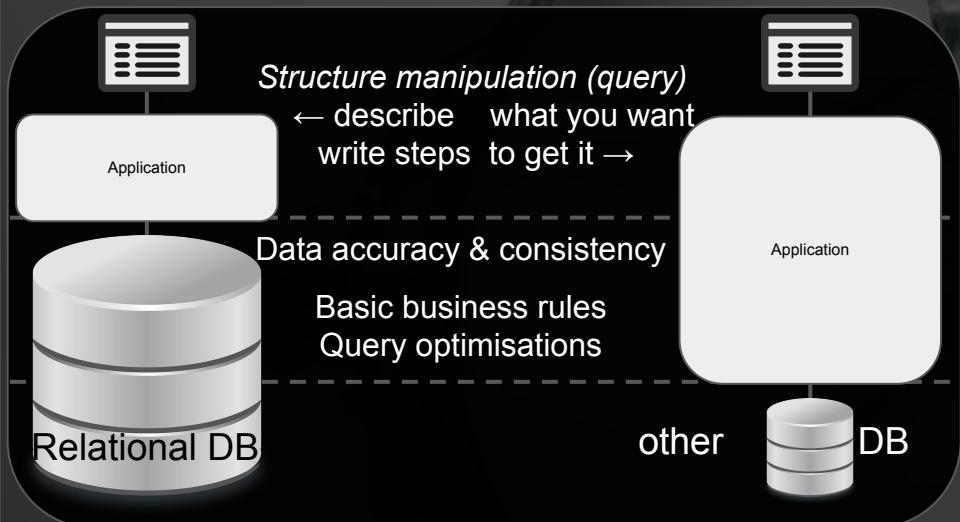
What if we don't have this by design?



In this form, we must write code to enforce / fix consistency issues...

- **INSERT anomalies**
 - Can't add a basket with no credit card number (without NULLs)
 - For each basket we must correctly enter both the till_id and store even though they always appear together.
- **UPDATE anomalies**
 - To change store for T12, we have to change two rows
- **DELETE anomalies**
 - If we remove B3, we remove any trace of the York store as well

What if we don't have this by design?



In this form we potentially lose data... must account for this somehow...

- **INSERT anomalies**
Can't add a basket with no credit card number (without NULLs)

For each basket we must correctly enter both the till_id and store even though they always appear together.
- **UPDATE anomalies**
To change store for T12, we have to change two rows
- **DELETE anomalies**
If we remove B3, we remove any trace of the York store as well

Summary

What you need to know:

- 1) Entity types (tables) are linked together by relationships.
- 2) Relationships are implicitly defined through **candidate** and **foreign keys**.
- 3) These can exist **by design** or **inherently**.
- 4) How to read entity relationship diagrams.

What you need to know:

- 1) Entity types (tables) are linked by keys.
- 2) **This does not limit what can be modelled.**

What you need to know:

- 1) The relational model can model all information*
- 2) There is a fixed, mathematical approach to arrive at tables that prevents unnecessary data replication.
- 3) This enables data consistency **by design** without application code.
- 4) When you want to find information, you'll generally only find it in one table...

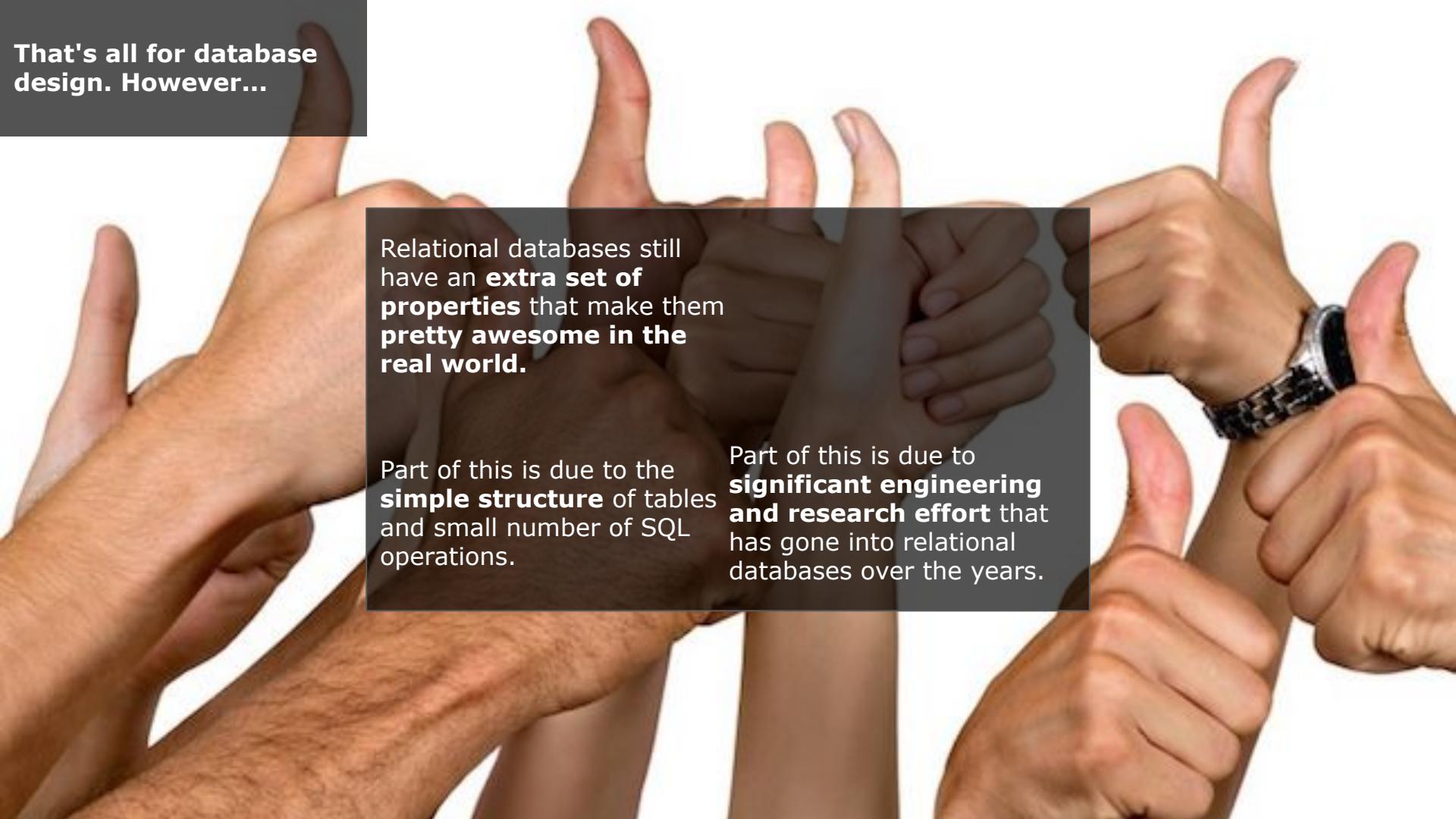
Want to know more about database design: Chapters 14 and 15 of Database Systems by Connolly & Begg

That's all for database design. However...



Relational databases still have an **extra set of properties** that make them **pretty awesome in the real world.**

That's all for database design. However...



Relational databases still have an **extra set of properties** that make them **pretty awesome in the real world.**

Part of this is due to the **simple structure** of tables and small number of SQL operations.

Part of this is due to **significant engineering and research effort** that has gone into relational databases over the years.

That's all for database design. However...



Specifically, relational databases have well defined methods to deal with **concurrency**.

That's all for database design. However...



Specifically, relational databases have well defined methods to deal with **concurrency**.

This enables **multiple people/machines** to update, read, add and delete data while **keeping the data consistent and correct**.

Prevents:

- Lost (overridden) updates
- Uncommitted update
- Inconsistent analysis
(i.e. data point changes partway through an algorithm, different values for the same thing used)

That's all for database design. However...



Specifically, relational databases have well defined methods to deal with **concurrency**.

Magic of databases is that transactions are executed concurrently unless absolutely required to run sequentially - **high efficiency**.

That's all for database design. However...

More information, see Chapter 14:
Database System Concepts, Silberschatz,
Korth & Sudarshan



Relational databases define and implement **transactions**.

Transactions (statements) **Transactions** have a guaranteed set of properties: ACID...

are defined as a "logical unit of work".

That's all for database design. However...

More information, see Chapter 14:
Database System Concepts, Silberschatz,
Korth & Sudarshan



ACID: Atomicity,
Consistency, Isolation,
Durability.

That's all for database design. However...

More information, see Chapter 14:
Database System Concepts, Silberschatz,
Korth & Sudarshan



Atomicity: Everything in a transaction either happens or it doesn't.

(all pieces of information are committed or none)

That's all for database design. However...

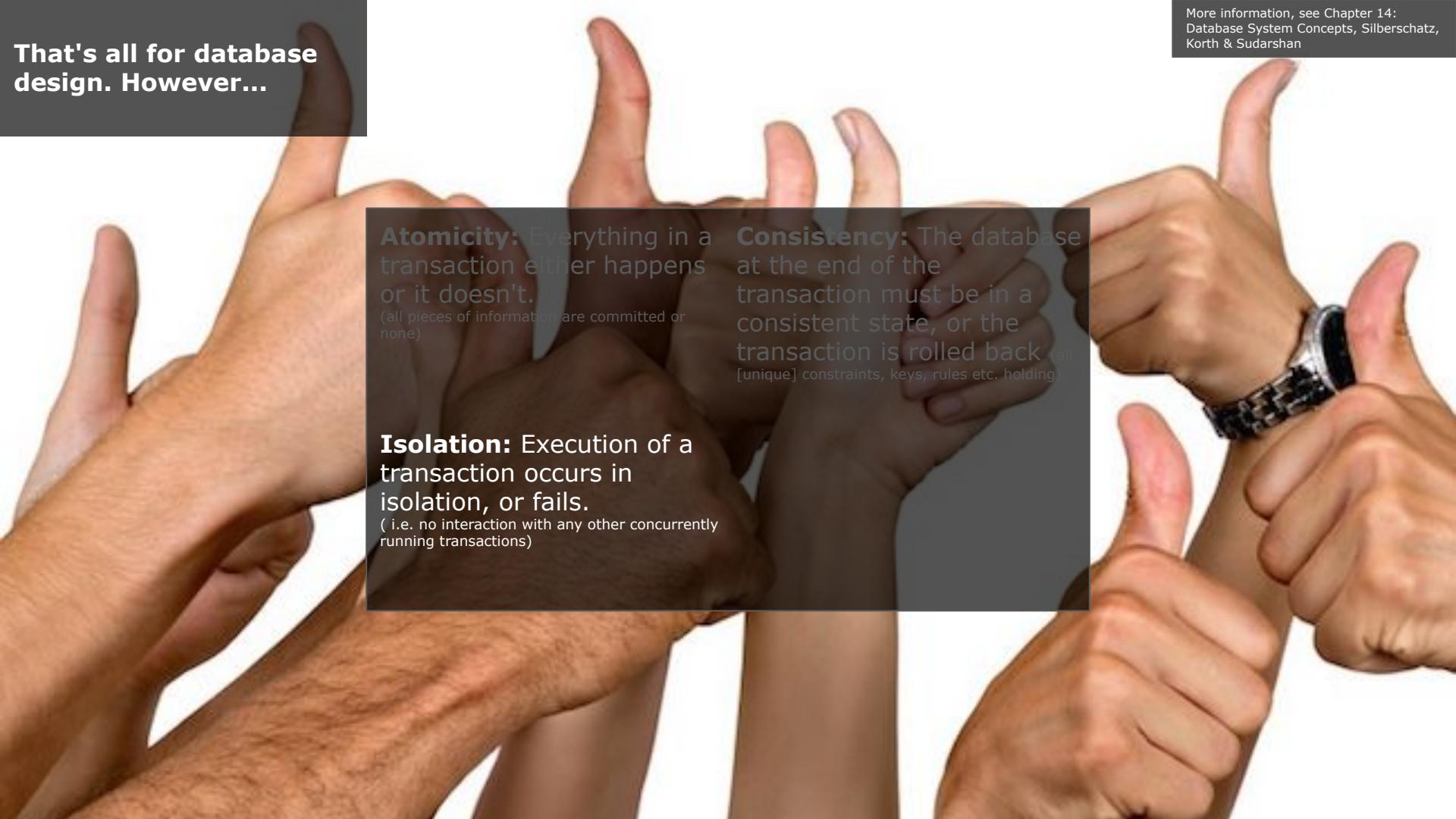
More information, see Chapter 14:
Database System Concepts, Silberschatz,
Korth & Sudarshan



Atomicity: Everything in a transaction either happens or it doesn't.

(all pieces of information are committed or none)

Consistency: The database at the end of the transaction must be in a consistent state, or the transaction is rolled back (all [unique] constraints, keys, rules etc. holding).



That's all for database design. However...

More information, see Chapter 14:
Database System Concepts, Silberschatz,
Korth & Sudarshan

Atomicity: Everything in a transaction either happens or it doesn't.

(all pieces of information are committed or none)

Isolation: Execution of a transaction occurs in isolation, or fails.

(i.e. no interaction with any other concurrently running transactions)

Consistency: The database at the end of the transaction must be in a consistent state, or the transaction is rolled back (all [unique] constraints, keys, rules etc. holding).



That's all for database design. However...

More information, see Chapter 14:
Database System Concepts, Silberschatz,
Korth & Sudarshan

Atomicity: Everything in a transaction either happens or it doesn't.

(all pieces of information are committed or none)

Isolation: Execution of a transaction occurs in isolation, or fails.

(i.e. no interaction with any other concurrently running transactions)

Consistency: The database at the end of the transaction must be in a consistent state, or the transaction is rolled back (all [unique] constraints, keys, rules etc. holding).

Durability: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.



That's all for database design. However...

More information, see Chapter 14:
Database System Concepts, Silberschatz,
Korth & Sudarshan

Atomicity: Everything in a transaction either happens or it doesn't.

(all pieces of information are committed or none)

Isolation: Execution of a transaction occurs in isolation, or fails.

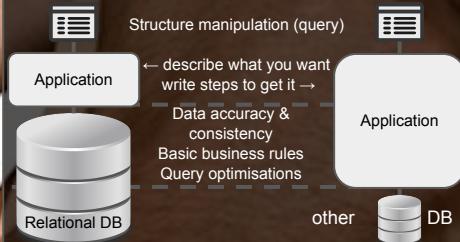
(i.e. no interaction with any other concurrently running transactions)

Consistency: The database at the end of the transaction must be in a consistent state, or the transaction is rolled back (all [unique] constraints, keys, rules etc. holding).

Durability: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

That's all for database design. However...

So that's nice. What's the catch?



Turns out it is **hard to scale** these mechanisms up to distributed environments. **noSQL** (i.e. most object databases) **explicitly "traded this off"**.

Time for our
morning exercises...

o..* 1..1 1..*

action

Vehicles	
Attribute	Data type
Vehicle ID	

Parts	
Attribute	Data type
Parts ID	

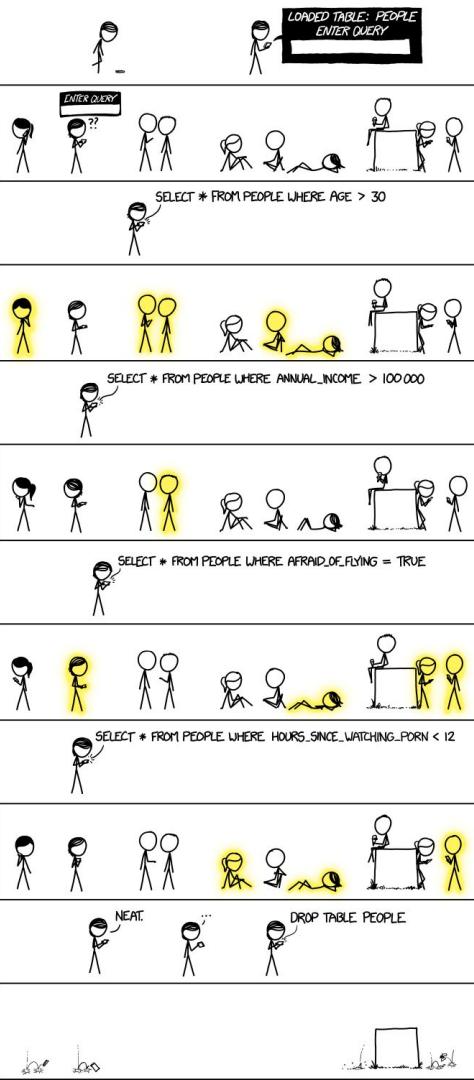
Pieces	
Attribute	Data type
Piece ID	
Type	
Parts ID	
Mechanics ID	

Mechanics	
Attribute	Data type
Job ID	
Factory address	



Session 7

Relational Databases (SQL II)



Remember back
9,900 minutes ago...

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

All operate on a single table.

All return a single table.

Think in tables, entities & attributes...

Remember back
9,900 minutes ago...



Remember back
9,900 minutes ago...

student

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

Only ask about students

Before:

Operated on **one table**.

Only could ask questions about one set
of entities.



Remember back
9,900 minutes ago...

student

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

Only ask about students

course

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

Only ask about courses

Before:

Operated on one table.

Only could ask questions about one set of entities.



Remember back
9,900 minutes ago...

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

Only ask about grades

student

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

S106 Mark Jones
Only ask about courses

course

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

Only ask about students

Before:

Operated on one table.

Only could ask questions about one set of entities.



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

Now: Joining tables together.

Run operators (ask questions) on new set of composite entities.



SQL: JOINING TABLES

(the traditional way)

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

Facts about two different entities types can be joined together to obtain facts about pairs of entity occurrences (one from each type).

We're going to learn the mechanism first, then interpretation.

Joining student occurrence: S103, John, Smith

With grade occurrence: S103, DBS, 72

Creates a new "Graded student" entity. (more on interpretation later)

Joining grade occurrence: S103, DBS, 72

With course occurrence: DBS, Database Systems

SQL: JOINING TABLES

(the traditional way)

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

Facts about two different entities types can be joined together to obtain facts about pairs of entity occurrences (one from each type).

We're going to learn the mechanism first, then interpretation.

Joining student occurrence: S103, John, Smith

With grade occurrence: S103, DBS, 72

Creates a new "Graded student" entity. (more on interpretation later)

Joining grade occurrence: S103, DBS, 72

With course occurrence: DBS, Database Systems

One-to-many relationship:

student 1...1 → has → 0..* grades

→ Grade table is appended / extended with its unique match in student

→ Creates a new, more detailed, grade entity.

SQL: JOINING TABLES

(the traditional way)

student

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

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
			S103	IAI	58
			S104	PR1	68
			S104	IAI	65
			S106	PR2	43
			S107	PR1	76
			S107	PR2	60
			S107	IAI	35
S104	Mary	Jones	S103	DBS	72
			S103	IAI	58
			S104	PR1	68
			S104	IAI	65
			S106	PR2	43
			S107	PR1	76
			S107	PR2	60
			S107	IAI	35
S105	Jane	Brown	S103	DBS	72
			S103	IAI	58
			S104	PR1	68
			S104	IAI	65
			S106	PR2	43
			S107	PR1	76
			S107	PR2	60
			S107	IAI	35
			S104	IAI	65
			...		

Joining tables joins **every row** in the first table to **every row** in the second.

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 student, grade
```

SQL: JOINING TABLES

(the traditional way)

student

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

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
			S103	IAI	58
			S104	PR1	68
			S104	IAI	65
			S106	PR2	43
			S107	PR1	76
			S107	PR2	60
			S107	IAI	35
S104	Mary	Jones	S103	DBS	72
			S103	IAI	58
			S104	PR1	68
			S104	IAI	65
			S106	PR2	43
			S107	PR1	76
			S107	PR2	60
			S107	IAI	35
S105	Jane	Brown	S103	DBS	72
			S103	IAI	58
			S104	PR1	68
			S104	IAI	65
			S106	PR2	43
			S107	PR1	76
			S107	PR2	60
			S107	IAI	35

Joining tables joins **every row** in the first table to **every row** in the second.

Database does not understand what resultant rows are **true and should be kept**.

```
SELECT *  
FROM student, grade
```

Database does not understand
what resultant rows are **true and**
should be kept.

YOU must tell it.

student

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

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

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S103	John	Smith	S104	PR1	68
S103	John	Smith	S104	IAI	65
S103	John	Smith	S106	PR2	43
S103	John	Smith	S107	PR1	76
S103	John	Smith	S107	PR2	60
S103	John	Smith	S107	IAI	35
S104	Mary	Jones	S103	DBS	72
S104	Mary	Jones	S103	IAI	58
S104	Mary	Jones	S104	PR1	68
S104	Mary	Jones	S104	IAI	65
S104	Mary	Jones	S106	PR2	43
S104	Mary	Jones	S107	PR1	76
S104	Mary	Jones	S107	PR2	60
S104	Mary	Jones	S107	IAI	35
S105	Jane	Brown	S103	DBS	72
S105	Jane	Brown	S103	IAI	58
S105	Jane	Brown	S104	PR1	68
S105	Jane	Brown	S104	IAI	65
S105	Jane	Brown	S106	PR2	43
S105	Jane	Brown	S107	PR1	76
S105	Jane	Brown	S107	PR2	60
S105	Jane	Brown	S107	IAI	35

```
SELECT *
FROM student, grade
WHERE student.id = grade.id
```

Database does not understand what resultant rows are **true and should be kept**.

YOU must tell it.

student

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

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

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S103	John	Smith	S104	PR1	68
S103	John	Smith	S104	IAI	65
S103	John	Smith	S106	PR2	43
S103	John	Smith	S107	PR1	76
S103	John	Smith	S107	PR2	60
S103	John	Smith	S107	IAI	35
S104	Mary	Jones	S103	DBS	72
S104	Mary	Jones	S103	IAI	58
S104	Mary	Jones	S104	PR1	68
S104	Mary	Jones	S104	IAI	65
S104	Mary	Jones	S106	PR2	43
S104	Mary	Jones	S107	PR1	76
S104	Mary	Jones	S107	PR2	60
S104	Mary	Jones	S107	IAI	35
S105	Jane	Brown	S103	DBS	72
S105	Jane	Brown	S103	IAI	58
S105	Jane	Brown	S104	PR1	68
S105	Jane	Brown	S104	IAI	65
S105	Jane	Brown	S106	PR2	43
S105	Jane	Brown	S107	PR1	76
S105	Jane	Brown	S107	PR2	60
S105	Jane	Brown	S107	IAI	35

```
SELECT *  
FROM student, grade  
WHERE student.id = grade.id
```

Database does not understand what resultant rows are **true and should be kept**.

YOU must tell it.

student

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

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

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S103	John	Smith	S104	PR1	68
S103	John	Smith	S104	IAI	65
S103	John	Smith	S106	PR2	43
S103	John	Smith	S107	PR1	76
S103	John	Smith	S107	PR2	60
S103	John	Smith	S107	IAI	35
S104	Mary	Jones	S103	DBS	72
S104	Mary	Jones	S103	IAI	58
S104	Mary	Jones	S104	PR1	68
S104	Mary	Jones	S104	IAI	65
S104	Mary	Jones	S106	PR2	43
S104	Mary	Jones	S107	PR1	76
S104	Mary	Jones	S107	PR2	60
S104	Mary	Jones	S107	IAI	35
S105	Jane	Brown	S103	DBS	72
S105	Jane	Brown	S103	IAI	58
S105	Jane	Brown	S104	PR1	68
S105	Jane	Brown	S104	IAI	65
S105	Jane	Brown	S106	PR2	43
S105	Jane	Brown	S107	PR1	76
S105	Jane	Brown	S107	PR2	60
S105	Jane	Brown	S107	IAI	35

```
SELECT *  
FROM student, grade  
WHERE student.id = grade.id
```

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S104	Mary	Jones	S104	PR1	68
S104	Mary	Jones	S104	IAI	65
S106	Mark	Jones	S106	PR2	43
S107	John	Brown	S107	PR1	76
S107	John	Brown	S107	PR2	60
S107	John	Brown	S107	IAI	35

After you have a single table!

And you know how to deal with those...

```
SELECT *
FROM student, grade
WHERE (student.id = grade.id)
```

Graded Students

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S104	Mary	Jones	S104	PR1	68
S104	Mary	Jones	S104	IAI	65
S106	Mark	Jones	S106	PR2	43
S107	John	Brown	S107	PR1	76
S107	John	Brown	S107	PR2	60
S107	John	Brown	S107	IAI	35

After you have a single table!

And you know how to deal with those...

```
SELECT *  
FROM student, grade  
WHERE (student.id = grade.id)
```

```
SELECT first, last, code, mark  
FROM student, grade  
WHERE (student.id = grade.id)
```

Graded Students

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S104	Mary	Jones	S104	PR1	68
S104	Mary	Jones	S104	IAI	65
S106	Mark	Jones	S106	PR2	43
S107	John	Brown	S107	PR1	76
S107	John	Brown	S107	PR2	60
S107	John	Brown	S107	IAI	35



first	last	code	mark
John	Smith	DBS	72
John	Smith	IAI	58
Mary	Jones	PR1	68
Mary	Jones	IAI	65
Mark	Jones	PR2	43
John	Brown	PR1	76
John	Brown	PR2	60
John	Brown	IAI	35

After you have a single table!

And you know how to deal with those...

```
SELECT *  
FROM student, grade  
WHERE (student.id = grade.id)
```

```
SELECT first, last, code, mark  
FROM student, grade  
WHERE (student.id = grade.id)
```

```
SELECT first, last  
FROM student, grade  
WHERE (student.id = grade.id)  
AND code = 'PR1'  
AND mark > 70
```

Graded Students

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S104	Mary	Jones	S104	PR1	68
S104	Mary	Jones	S104	IAI	65
S106	Mark	Jones	S106	PR2	43
S107	John	Brown	S107	PR1	76
S107	John	Brown	S107	PR2	60
S107	John	Brown	S107	IAI	35

first	last	code	mark
John	Smith	DBS	72
John	Smith	IAI	58
Mary	Jones	PR1	68
Mary	Jones	IAI	65
Mark	Jones	PR2	43
John	Brown	PR1	76
John	Brown	PR2	60
John	Brown	IAI	35

first	last
John	Brown

After we exhaustively join first two tables, let's exhaustively joint the third...

```
SELECT *  
FROM student, grade, course
```

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S103	John	Smith	S104	PR1	68
S103	John	Smith	S104	IAI	65
S103	John	Smith	S106	PR2	43
S103	John	Smith	S107	PR1	76
S103	John	Smith	S107	PR2	60
S103	John	Smith	S107	IAI	35
S104	Mary	Jones	S103	DBS	72
S104	Mary	Jones	S103	IAI	58
S104	Mary	Jones	S104	PR1	68
S104	Mary	Jones	S104	IAI	65
S104	Mary	Jones	S106	PR2	43
S104	Mary	Jones	S107	PR1	76
S104	Mary	Jones	S107	PR2	60
S104	Mary	Jones	S107	IAI	35



After we exhaustively join first two tables exhaustively joint the third...

```
SELECT *  
FROM student, grade, course
```

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S103	John	Smith	S104	PR1	68
S103	John	Smith	S104	IAI	65
S103	John	Smith	S106	PR2	43
S103	John	Smith	S107	PR1	76
S103	John	Smith	S107	PR2	60
S103	John	Smith	S107	IAI	35
S104	Mary	Jones	S103	DBS	72
S104	Mary	Jones	S103	IAI	58
S104	Mary	Jones	S104	PR1	68
S104	Mary	Jones	S104	IAI	65
S104	Mary	Jones	S106	PR2	43
S104	Mary	Jones	S107	PR1	76
S104	Mary	Jones	S107	PR2	60
S104	Mary	Jones	S107	IAI	35

id	first	last	id	code	mark	code	title
S103	John	Smith	S103	DBS	72	DBS	Database Systems
						PR1	Programming 1
						PR2	Programming 2
						IAI	Intro to AI
S103	John	Smith	S103	IAI	58	DBS	Database Systems
						PR1	Programming 1
						PR2	Programming 2
						IAI	Intro to AI
S103	John	Smith	S104	PR1	68	DBS	Database Systems
						PR1	Programming 1
						PR2	Programming 2
						IAI	Intro to AI
S103	John	Smith	S104	IAI	65	DBS	Database Systems
						PR1	Programming 1
						PR2	Programming 2
						IAI	Intro to AI
S103	John	Smith	S106	PR2	43	DBS	Database Systems
						PR1	Programming 1
						PR2	Programming 2
						IAI	Intro to AI
S103	John	Smith	S107	PR1	76	DBS	Database Systems
						PR1	Programming 1
						PR2	Programming 2
						IAI	Intro to AI

course

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

SQL: JOINING TABLES

(the traditional way)

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

What about three tables?

Same deal, facts "new" entity.

Joining student: S103, John, Smith

With grade: S103, DBS, 72

With course: DBS, Database Systems

Creates a new, "**detailed graded student**" entity.

id	first	last	id	code	mark	code	title
S103	John	Smith	S103	DBS	72	DBS	Database Systems
S103	John	Smith	S103	DBS	72	PR1	Programming 1
S103	John	Smith	S103	DBS	72	PR2	Programming 2
S103	John	Smith	S103	DBS	72	IAI	Intro to AI
S103	John	Smith	S103	IAI	58	DBS	Database Systems
S103	John	Smith	S103	IAI	58	PR1	Programming 1
S103	John	Smith	S103	IAI	58	PR2	Programming 2
S103	John	Smith	S103	IAI	58	IAI	Intro to AI
S103	John	Smith	S104	PR1	68	DBS	Database Systems
S103	John	Smith	S104	PR1	68	PR1	Programming 1
S103	John	Smith	S104	PR1	68	PR2	Programming 2
S103	John	Smith	S104	PR1	68	IAI	Intro to AI
S103	John	Smith	S104	IAI	65	DBS	Database Systems
S103	John	Smith	S104	IAI	65	PR1	Programming 1
S103	John	Smith	S104	IAI	65	PR2	Programming 2
S103	John	Smith	S104	IAI	65	IAI	Intro to AI
S103	John	Smith	S106	PR2	43	DBS	Database Systems
S103	John	Smith	S106	PR2	43	PR1	Programming 1
S103	John	Smith	S106	PR2	43	PR2	Programming 2
S103	John	Smith	S106	PR2	43	IAI	Intro to AI
S103	John	Smith	S107	PR1	76	DBS	Database Systems

Again, we must filter to retain only rows that are **true**

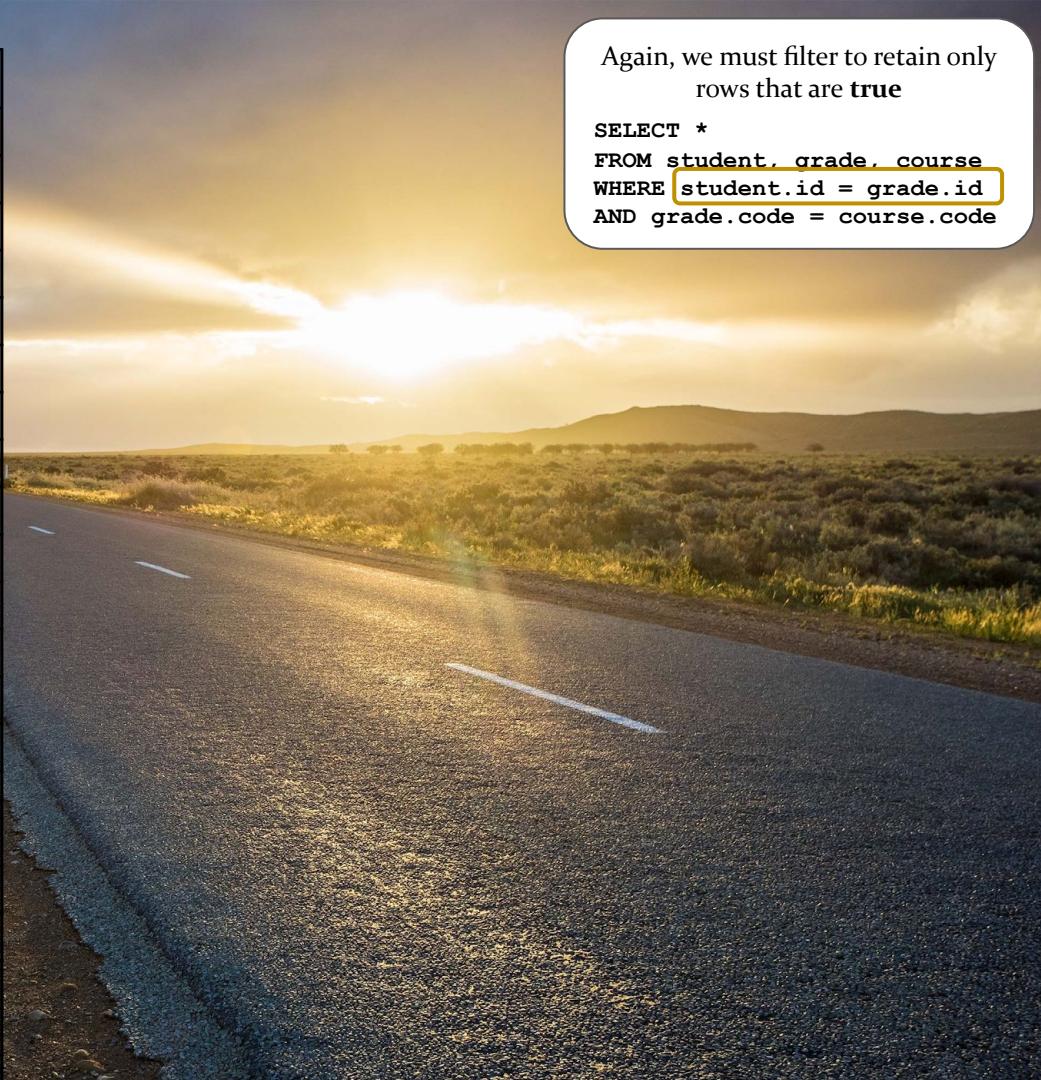
```
SELECT *
FROM student, grade, course
WHERE student.id = grade.id
AND grade.code = course.code
```



id	first	last	id	code	mark	code	title
S103	John	Smith	S103	DBS	72	DBS	Database Systems
S103	John	Smith	S103	DBS	72	PR1	Programming 1
S103	John	Smith	S103	DBS	72	PR2	Programming 2
S103	John	Smith	S103	DBS	72	IAI	Intro to AI
S103	John	Smith	S103	IAI	58	DBS	Database Systems
S103	John	Smith	S103	IAI	58	PR1	Programming 1
S103	John	Smith	S103	IAI	58	PR2	Programming 2
S103	John	Smith	S103	IAI	58	IAI	Intro to AI
S103	John	Smith	S104	PR1	68	DBS	Database Systems
S103	John	Smith	S104	PR1	68	PR1	Programming 1
S103	John	Smith	S104	PR1	68	PR2	Programming 2
S103	John	Smith	S104	PR1	68	IAI	Intro to AI
S103	John	Smith	S104	IAI	65	DBS	Database Systems
S103	John	Smith	S104	IAI	65	PR1	Programming 1
S103	John	Smith	S104	IAI	65	PR2	Programming 2
S103	John	Smith	S104	IAI	65	IAI	Intro to AI
S103	John	Smith	S106	PR2	43	DBS	Database Systems
S103	John	Smith	S106	PR2	43	PR1	Programming 1
S103	John	Smith	S106	PR2	43	PR2	Programming 2
S103	John	Smith	S106	PR2	43	IAI	Intro to AI
S103	John	Smith	S107	PR1	76	DBS	Database Systems

Again, we must filter to retain only rows that are **true**

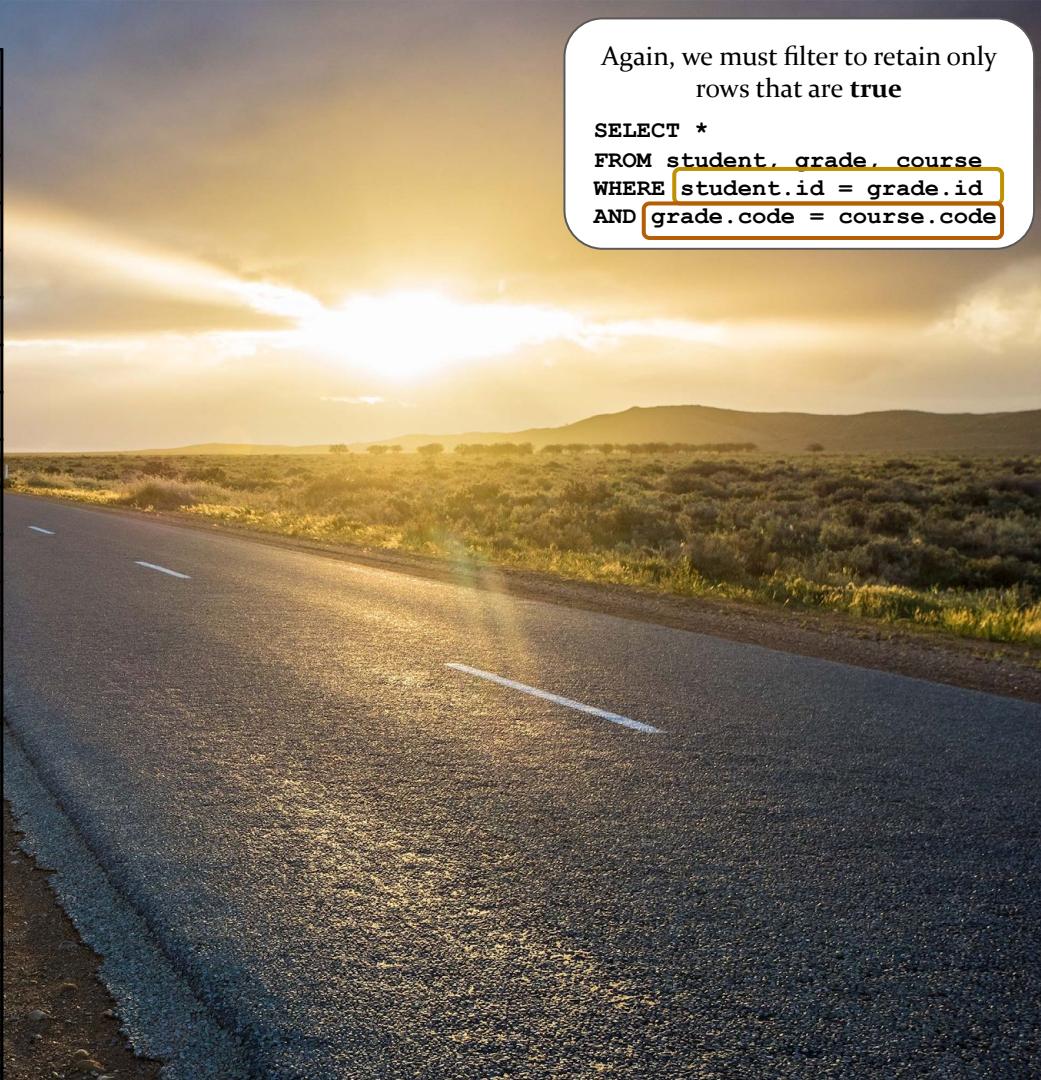
```
SELECT *
FROM student, grade, course
WHERE student.id = grade.id
AND grade.code = course.code
```



id	first	last	id	code	mark	code	title
S103	John	Smith	S103	DBS	72	DBS	Database Systems
S103	John	Smith	S103	DBS	72	PR1	Programming 1
S103	John	Smith	S103	DBS	72	PR2	Programming 2
S103	John	Smith	S103	DBS	72	IAI	Intro to AI
S103	John	Smith	S103	IAI	58	DBS	Database Systems
S103	John	Smith	S103	IAI	58	PR1	Programming 1
S103	John	Smith	S103	IAI	58	PR2	Programming 2
S103	John	Smith	S103	IAI	58	IAI	Intro to AI
S103	John	Smith	S104	PR1	68	DBS	Database Systems
S103	John	Smith	S104	PR1	68	PR1	Programming 1
S103	John	Smith	S104	PR1	68	PR2	Programming 2
S103	John	Smith	S104	PR1	68	IAI	Intro to AI
S103	John	Smith	S104	IAI	65	DBS	Database Systems
S103	John	Smith	S104	IAI	65	PR1	Programming 1
S103	John	Smith	S104	IAI	65	PR2	Programming 2
S103	John	Smith	S104	IAI	65	IAI	Intro to AI
S103	John	Smith	S106	PR2	43	DBS	Database Systems
S103	John	Smith	S106	PR2	43	PR1	Programming 1
S103	John	Smith	S106	PR2	43	PR2	Programming 2
S103	John	Smith	S106	PR2	43	IAI	Intro to AI
S103	John	Smith	S107	PR1	76	DBS	Database Systems

Again, we must filter to retain only rows that are **true**

```
SELECT *
FROM student, grade, course
WHERE student.id = grade.id
AND grade.code = course.code
```



id	first	last	id	code	mark	code	title
S103	John	Smith	S103	DBS	72	DBS	Database Systems
S103	John	Smith	S103	DBS	72	PR1	Programming 1
S103	John	Smith	S103	DBS	72	PR2	Programming 2
S103	John	Smith	S103	DBS	72	IAI	Intro to AI
S103	John	Smith	S103	IAI	58	DBS	Database Systems
S103	John	Smith	S103	IAI	58	PR1	Programming 1
S103	John	Smith	S103	IAI	58	PR2	Programming 2
S103	John	Smith	S103	IAI	58	IAI	Intro to AI
S103	John	Smith	S104	PR1	68	DBS	Database Systems
S103	John	Smith	S104	PR1	68	PR1	Programming 1
S103	John	Smith	S104	PR1	68	PR2	Programming 2
S103	John	Smith	S104	PR1	68	IAI	Intro to AI
S103	John	Smith	S104	IAI	65	DBS	Database Systems
S103	John	Smith	S104	IAI	65	PR1	Programming 1
S103	John	Smith	S104	IAI	65	PR2	Programming 2
S103	John	Smith	S104	IAI	65	IAI	Intro to AI
S103	John	Smith	S106	PR2	43	DBS	Database Systems
S103	John	Smith	S106	PR2	43	PR1	Programming 1
S103	John	Smith	S106	PR2	43	PR2	Programming 2
S103	John	Smith	S106	PR2	43	IAI	Intro to AI
S103	John	Smith	S107	PR1	76	DBS	Database Systems

✓ ✗

Again, we must filter to retain only rows that are **true**

```
SELECT *
FROM student, grade, course
WHERE student.id = grade.id
AND grade.code = course.code
```

id	first	last	id	code	mark	code	title
S103	John	Smith	S103	DBS	72	DBS	Database Systems
S103	John	Smith	S103	DBS	72	PR1	Programming 1
S103	John	Smith	S103	DBS	72	PR2	Programming 2
S103	John	Smith	S103	DBS	72	IAI	Intro to AI
S103	John	Smith	S103	IAI	58	DBS	Database Systems
S103	John	Smith	S103	IAI	58	PR1	Programming 1
S103	John	Smith	S103	IAI	58	PR2	Programming 2
S103	John	Smith	S103	IAI	58	IAI	Intro to AI
S103	John	Smith	S104	PR1	68	DBS	Database Systems
S103	John	Smith	S104	PR1	68	PR1	Programming 1
S103	John	Smith	S104	PR1	68	PR2	Programming 2
S103	John	Smith	S104	PR1	68	IAI	Intro to AI
S103	John	Smith	S104	IAI	65	DBS	Database Systems
S103	John	Smith	S104	IAI	65	PR1	Programming 1
S103	John	Smith	S104	IAI	65	PR2	Programming 2
S103	John	Smith	S104	IAI	65	IAI	Intro to AI
S103	John	Smith	S106	PR2	43	DBS	Database Systems
S103	John	Smith	S106	PR2	43	PR1	Programming 1
S103	John	Smith	S106	PR2	43	PR2	Programming 2
S103	John	Smith	S106	PR2	43	IAI	Intro to AI
S103	John	Smith	S107	PR1	76	DBS	Database Systems

Again, we must filter to retain only rows that are **true**

```
SELECT *
FROM student, grade, course
WHERE student.id = grade.id
AND grade.code = course.code
```

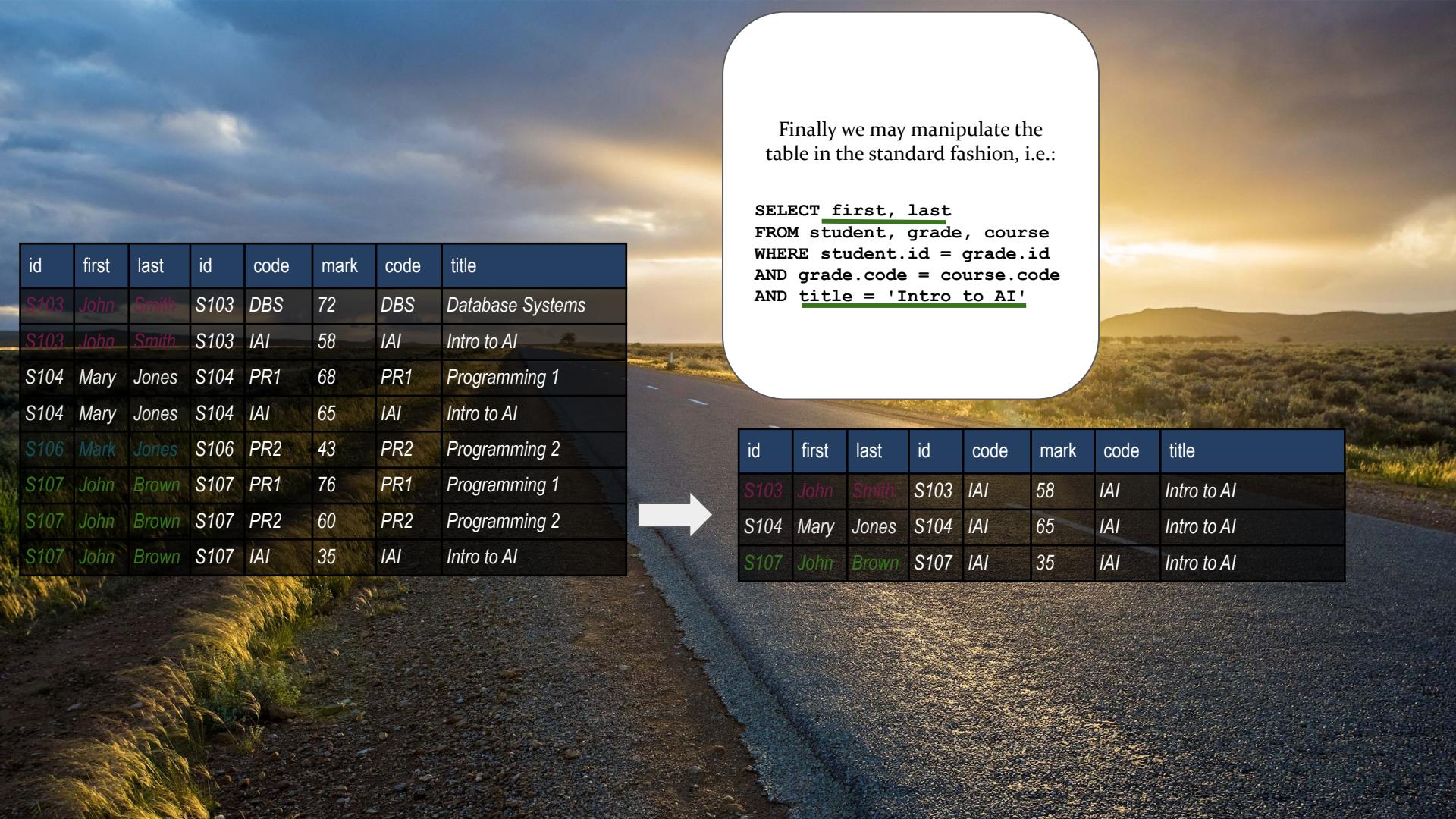
Again, we must filter to retain only rows that are **true**

```
SELECT *
FROM student, grade, course
WHERE student.id = grade.id
AND grade.code = course.code
```

id	first	last	id	code	mark	code	title
S103	John	Smith	S103	DBS	72	DBS	Database Systems
S103	John	Smith	S103	DBS	72	PR1	Programming 1
S103	John	Smith	S103	DBS	72	PR2	Programming 2
S103	John	Smith	S103	DBS	72	IAI	Intro to AI
S103	John	Smith	S103	IAI	58	DBS	Database Systems
S103	John	Smith	S103	IAI	58	PR1	Programming 1
S103	John	Smith	S103	IAI	58	PR2	Programming 2
S103	John	Smith	S103	IAI	58	IAI	Intro to AI
S103	John	Smith	S104	PR1	68	DBS	Database Systems
S103	John	Smith	S104	PR1	68	PR1	Programming 1
S103	John	Smith	S104	PR1	68	PR2	Programming 2
S103	John	Smith	S104	PR1	68	IAI	Intro to AI
S103	John	Smith	S104	IAI	65	DBS	Database Systems
S103	John	Smith	S104	IAI	65	PR1	Programming 1
S103	John	Smith	S104	IAI	65	PR2	Programming 2
S103	John	Smith	S104	IAI	65	IAI	Intro to AI
S103	John	Smith	S106	PR2	43	DBS	Database Systems
S103	John	Smith	S106	PR2	43	PR1	Programming 1
S103	John	Smith	S106	PR2	43	PR2	Programming 2
S103	John	Smith	S106	PR2	43	IAI	Intro to AI
S103	John	Smith	S107	PR1	76	DBS	Database Systems



id	first	last	id	code	mark	code	title
S103	John	Smith	S103	DBS	72	DBS	Database Systems
S103	John	Smith	S103	IAI	58	IAI	Intro to AI
S104	Mary	Jones	S104	PR1	68	PR1	Programming 1
S104	Mary	Jones	S104	IAI	65	IAI	Intro to AI
S106	Mark	Jones	S106	PR2	43	PR2	Programming 2
S107	John	Brown	S107	PR1	76	PR1	Programming 1
S107	John	Brown	S107	PR2	60	PR2	Programming 2
S107	John	Brown	S107	IAI	35	IAI	Intro to AI



Finally we may manipulate the table in the standard fashion, i.e.:

```
SELECT first, last
FROM student, grade, course
WHERE student.id = grade.id
AND grade.code = course.code
AND title = 'Intro to AI'
```

id	first	last	id	code	mark	code	title
S103	John	Smith	S103	DBS	72	DBS	Database Systems
S103	John	Smith	S103	IAI	58	IAI	Intro to AI
S104	Mary	Jones	S104	PR1	68	PR1	Programming 1
S104	Mary	Jones	S104	IAI	65	IAI	Intro to AI
S106	Mark	Jones	S106	PR2	43	PR2	Programming 2
S107	John	Brown	S107	PR1	76	PR1	Programming 1
S107	John	Brown	S107	PR2	60	PR2	Programming 2
S107	John	Brown	S107	IAI	35	IAI	Intro to AI

id	first	last	id	code	mark	code	title
S103	John	Smith	S103	IAI	58	IAI	Intro to AI
S104	Mary	Jones	S104	IAI	65	IAI	Intro to AI
S107	John	Brown	S107	IAI	35	IAI	Intro to AI

Some observations...

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

students

id	first	last	age
S103	John	Smith	22
S104	Mary	Jones	27
S105	Jane	Brown	21
S106	Mark	Jones	26
S107	John	Brown	30

course

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

Joining tables gives us:

New set of entities

(row: "Graded student", new set of jointly true facts)



Some observations...

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

students

id	first	last	age
S103	John	Smith	22
S104	Mary	Jones	27
S105	Jane	Brown	21
S106	Mark	Jones	26
S107	John	Brown	30

course

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

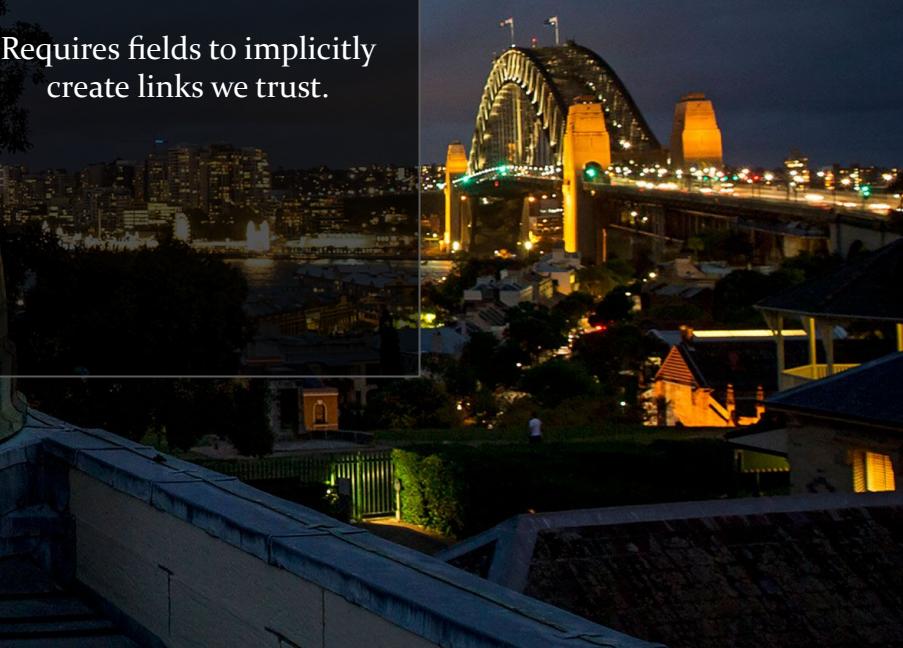
Joining tables gives us:

New set of entities

(row: "Graded student", new set of jointly true facts)

Does the joint entity you're making make sense?
Will the each **rows facts** be **jointly true?**

Requires fields to implicitly create links we trust.



Some observations...

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

students

id	first	last	age
S103	John	Smith	22
S104	Mary	Jones	27
S105	Jane	Brown	21
S106	Mark	Jones	26
S107	John	Brown	30

course

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

Joining tables gives us:

New set of entities

(row: "Graded student", new set of jointly true facts)

Does the joint entity you're making make sense?
Will the each **rows** facts be **jointly true**?

Requires fields to implicitly create links we trust.

Links considered at **design-time**: unambiguous

(e.g. Bob's ID in hospital & chef table
e.g. transaction_id in basket and line_items table)

Other links (e.g. linking by date): Opportunities! But think carefully...



Understanding when to link: Documentation

Documentation describes
fields that link data and
tell us **the relationship**
(how to interpret the result after linking)

Recall:

Candidate/Primary keys:
Uniquely identify a entity
occurrence (row)

Foreign keys:
Record an identifier (link) to
a candidate key in another
table

Understanding when to link: Documentation

Documentation describes
fields that link data and
tell us **the relationship**
(how to interpret the result after linking)

How we **interpret** the resulting **entity instances** is guided by the relationship information (documented = enforced) (non-documented = at our own risk)

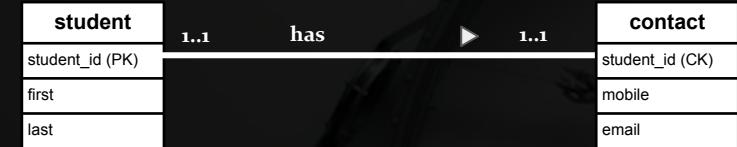
Recall:

Candidate/Primary keys:
Uniquely identify a entity occurrence (row)

Foreign keys:
Record an identifier (link) to a candidate key in another table

One-to-one

Relationship is always implicit,
and is realised due to the JOIN
and the constraints (enforced by
design or opportunistic) on to the
linking fields.



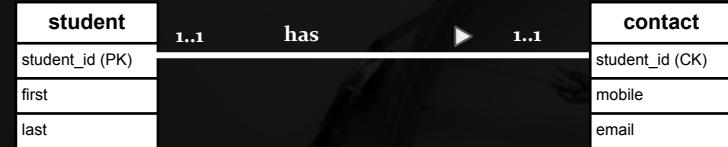
id	first	last
1	Gav	Smith
2	Bob	Brown

id	mobile	email
1	123	g@s
2	586	b@b

One-to-one

Relationship is always implicit, and is realised due to the JOIN and the constraints (enforced by design or opportunistic) on to the linking fields.

Consider the join of the tables in this example. All rows from **student** are joined with all rows from **contact**.



id	first	last
1	Gav	Smith
2	Bob	Brown

id	mobile	email
1	123	g@s
2	586	b@b

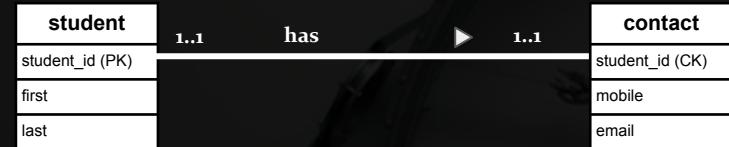
id	first	last	id	mobile	email
1	Gav	Smith	1	123	g@s
1	Gav	Smith	2	586	b@b
2	Bob	Brown	1	123	g@s
2	Bob	Brown	2	586	b@b

One-to-one

Relationship is always implicit, and is realised due to the JOIN and the constraints (enforced by design or opportunistic) on to the linking fields.

Consider the join of the tables in this example. All rows from **student** are joined with all rows from **contact**.

Due to the PK and CK constraints on the linking fields, only one row per student possible in the output table.



id	first	last
1	Gav	Smith
2	Bob	Brown

id	mobile	email
1	123	g@s
2	586	b@b

id	first	last	id	mobile	email
1	Gav	Smith	1	123	g@s
1	Gav	Smith	2	586	b@b
2	Bob	Brown	1	123	g@s
2	Bob	Brown	2	586	b@b

id	first	last	mobile	email
1	Gav	Smith	123	g@s
2	Bob	Brown	486	b@b

One-to-one

(enforced by cust_id being primary/candidate keys in each table)

Relationship is always implicit, and is realised due to the JOIN and the constraints (enforced by design or opportunistic) on to the linking fields.

Consider the join of the tables in this example. All rows from **student** are joined with all rows from **contact**.

Due to the PK and CK constraints on the linking fields, only one row per student possible in the output table.



Interpretation is simple. Equivalently:

- Each **contact** record is **extended** with **student** information.
- Each **student** record is **extended** with **contact** information.

id	first	last	mobile	email
1	Gav	Smith	123	g@s
2	Bob	Brown	486	b@b

Most Logical (maybe) Interpretation

(but not the easiest to use to "think" in SQL as it is a conceptual nested structure - not the basis of SQL!)

Each customer has a list of transactions.

Could view this as a customer "entity" with multiple transactions.

Mark

→	2017-08-09	08:05:32	£12.00
→	2017-08-10	13:00:00	£6.50

Jane

→	2017-08-10	11:00:01	£3.00
---	------------	----------	-------

One-to-many

(enforced by cust_id being primary key in the "one end" of the table and a foreign key in the "many end" of the relationship)



Entity type: customer

cust_id	first	last
1	Mark	Jones
2	Victoria	Smith
3	Jane	Doe

Entity type: transaction

cust_id	date	time	total
1	2017-08-09	08:05:32	£12.00
1	2017-08-10	13:00:00	£6.50
3	2017-08-10	11:00:01	£3.00

One-to-many

(enforced by `cust_id` being primary key in the "one end" of the table and a foreign key in the "many end" of the relationship)



Most Logical (maybe) Interpretation

(but not the easiest to use to "think" in SQL as it is a conceptual nested structure - not the basis of SQL!)

Each customer has a list of transactions.

Could view this as a customer "entity" with multiple transactions.

Mark

→	2017-08-09	08:05:32	£12.00
→	2017-08-10	13:00:00	£6.50

Jane

→	2017-08-10	11:00:01	£3.00
---	------------	----------	-------

Entity type: customer			Entity type: transaction			
cust_id	first	last	cust_id	date	time	total
1	Mark	Jones	1	2017-08-09	08:05:32	£12.00
2	Victoria	Smith	1	2017-08-10	13:00:00	£6.50
3	Jane	Doe	3	2017-08-10	11:00:01	£3.00

Relationship is always implicit, and is realised due to the JOIN and the constraints (enforced by design or opportunistic) on to the linking fields.

Enforced by

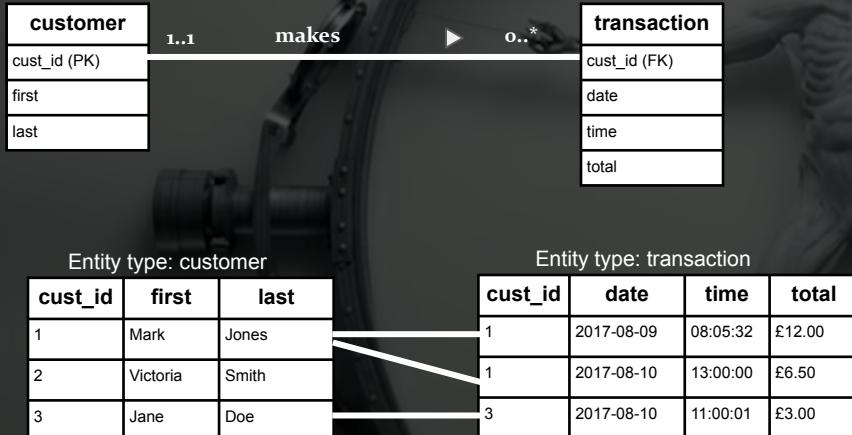
→ **`cust_id` being primary key in the "one end" of the table**

→ **a foreign key in the "many end" of the relationship**
(records the `cust_id` of the transaction)

→ When all rows from each table are compared, only true rows are those where the customer id's match.

I.e. where the transaction record has been extended with the right customer information.

One-to-many



Potentially easier
if we stay "thinking
in tables"

Alternate interpretation
View as a transaction
entity augmented with
customer information.

cust_id	first	last	date	time	total
1	Mark	Jones	2017-08-09	08:05:32	£12.00
1	Mark	Jones	2017-08-10	13:00:00	£6.50
3	Jane	Doe	2017-08-10	11:00:01	£3.00

One-to-many

Remember:

The result of joining two tables is a new table.

**Always think:
What is the entity type represented?**

Relationships (particularly if they are enforced) guide our interpretation.

Each customer has a list of transactions.

Could view this as a customer "entity" with multiple transactions.

Mark	→	2017-08-09	08:05:32	£12.00
	→	2017-08-10	13:00:00	£6.50
Jane	→	2017-08-10	11:00:01	£3.00

A
V
E
C

Identical structure information. But we have a table. **Keeping things simple for SQL.**

cust_id	first	last	date	time	total
1	Mark	Jones	2017-08-09	08:05:32	£12.00
			2017-08-10	13:00:00	£6.50
3	Jane	Doe	2017-08-10	11:00:01	£3.00

Many-to-many

Entity type: basket

b_id	time	loyalty_id	total
b1	11am	7896	£3.70
b2	1pm	9934	£3.25
b3	2pm	3900	£0.07

Entity type: items

i_id	desc.	cost
i1	Doritos	£3.00
i2	Chocolate bar	£0.70
i3	Sandwich	£3.25



Should not exist in a well designed relational database.

Interpretation

A basket entity (row) would need to store many items. A row must, however, only be atomic entities.

Entity type: basket

b_id	time	loyalty_id	total
b1	11am	7896	£3.70
b2	1pm	9934	£3.25
b3	2pm	3900	£0.07

Entity type: items

i_id	desc.	cost
i1	Doritos	£3.00
i2	Chocolate bar	£0.70
i3	Sandwich	£3.25



Many-to-many

Interpretation

Should not exist in a well designed relational database.

If it did, the items table is no longer "items" as we'd conceptualize it (one item is one entity occurrence).

Entity type: basket

b_id	time	loyalty_id	total
b1	11am	7896	£3.70
b2	1pm	9934	£3.25
b3	2pm	3900	£0.07

Entity type: items

i_id	desc.	cost
i1	Doritos	£3.00
i2	Chocolate bar	£0.70
i3	Sandwich	£3.25

basket

time
loyalty_id
total

items

desc.
cost

o..*



1..*

Same information, corrected to being actual "tables". Becomes three tables and two one-to-many relationships.

basket

basket_id
time
loyalty_id
total

line_items

basket_id
item_id

*

o..*

▲

listed in

items

item_id
desc.
cost

AGGREGATE FUNCTIONS

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

OK, so joins are fun...
But let's step back...

And look at operations on
a single (or post join)
table.

Question: What is the
maximum mark per
course?

Answer in SQL:

```
SELECT code, MAX(mark)
FROM grade
GROUP BY code
```

AGGREGATE FUNCTIONS

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

OK, so joins are fun...
But let's step back...

And look at operations on
a single (or post join)
table.

Question: What is the
maximum mark per
course?

Answer in SQL:

```
SELECT code, MAX(mark)
FROM grade
GROUP BY code
```

Step 1: Group (put) the
rows into buckets, one
bucket per distinct module
code value

AGGREGATE FUNCTIONS

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

OK, so joins are fun...
But let's step back...

Question: What is the maximum mark per course?

And look at operations on a single (or post join) table.

Answer in SQL:

```
SELECT code, MAX(mark)  
FROM grade  
GROUP BY code
```

Step 2: Per group (bucket) compute the aggregate function (in this case max) over all values in the bucket

Step 1: Group (put) the rows into buckets, one bucket per distinct module code value

AGGREGATE FUNCTIONS

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

OK, so joins are fun...
But let's step back...

Question: What is the maximum mark per course?

And look at operations on a single (or post join) table.

Answer in SQL:

```
SELECT code, MAX(mark)  
FROM grade  
GROUP BY code
```

Step 3: Return the bucket label and the aggregate values.

NOTE: Bucket labels need not be included. Fields not part of a bucket label can't be.

WHY? Ambiguous, what value of the many should have been picked per row?

Step 2: Per group (bucket) compute the aggregate function (in this case max) over all values in the bucket

Step 1: Group (put) the rows into buckets, one bucket per distinct module code value

AGGREGATE FUNCTIONS

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

What about if we want just
one big bucket?

(group over everything)
(max over all rows)

Question: What is the
maximum mark?

Answer in SQL:

```
SELECT MAX(mark)  
FROM grade
```

AGGREGATE FUNCTIONS

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

What about if we want just one big bucket?

(group over everything)
(max over all rows)

Question: What is the maximum mark?

Answer in SQL:

```
SELECT MAX(mark)  
FROM grade
```

Step 0: Don't specify bucket labels.

AGGREGATE FUNCTIONS

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

What about if we want just one big bucket?

(group over everything)
(max over all rows)

Question: What is the maximum mark?

Answer in SQL:

```
SELECT  
FROM grade
```

MAX (mark)

Step 2: The aggregate function (in this case max) is computed over all values.

Step 0: Don't specify bucket labels.

AGGREGATE FUNCTIONS

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

What about if we want just one big bucket?

(group over everything)
(max over all rows)

Question: What is the maximum mark?

Answer in SQL:

```
SELECT MAX(mark)  
FROM grade
```

Step 3: Return the aggregate values

NOTE: No bucket labels, no non-aggregate terms.

WHY? Ambiguous, what value of the many should have been picked per row?

Step 2: The aggregate function (in this case max) is computed over all values.

Step 1: Don't specify bucket labels.

AGGREGATE FUNCTIONS

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

There are many aggregate functions.

You can write your own
(in Python even!)

Standard aggregate functions

count avg variance
max sum median
min stdev

AGGREGATE FUNCTIONS

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

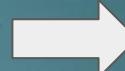
All operate under the same principal.

Work on the bucket of data given by the group by.

More than one aggregate can be run over the buckets at once.

```
SELECT code,  
       MIN(mark),  
       MAX(mark),  
       AVG(mark),  
FROM grade  
GROUP BY code
```

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



code	min	max	avg
DBS	72	72	72
IAI	35	65	52.6
PR1	68	76	72
PR2	43	60	51.5

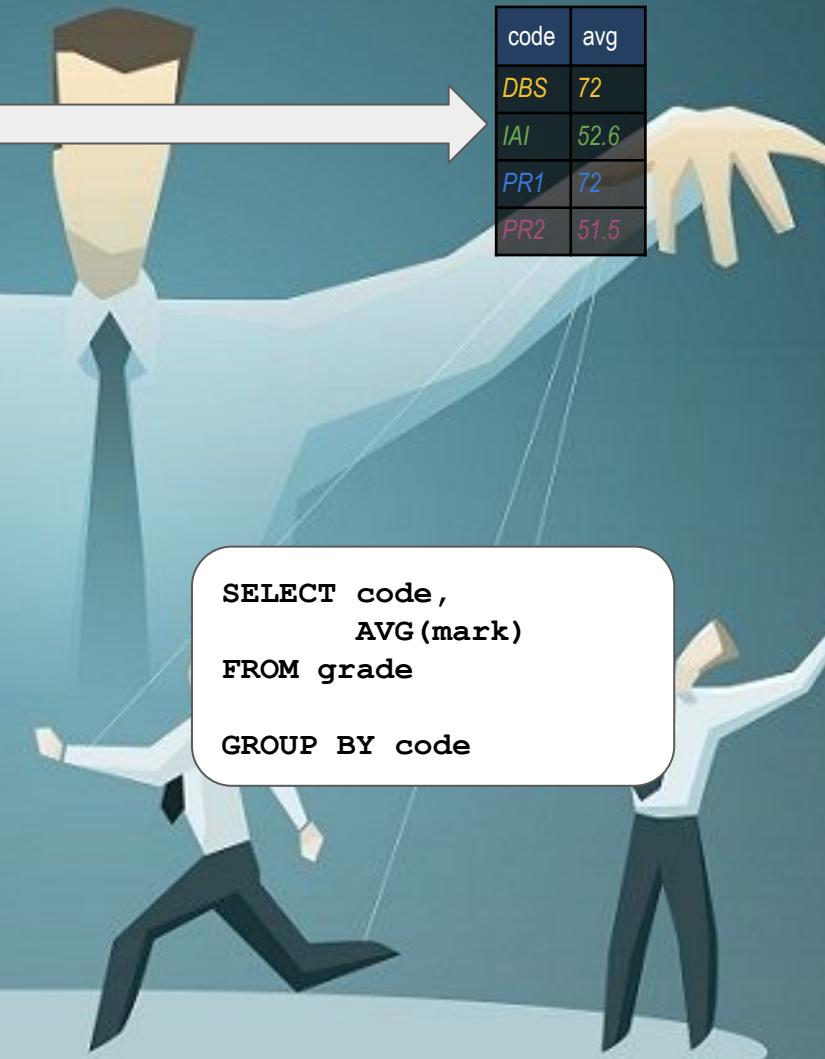
Conditions on AGGREGATE FUNCTIONS

Aggregate function
without conditions

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

code	avg
DBS	72
IAI	52.6
PR1	72
PR2	51.5

```
SELECT code,  
       AVG(mark)  
  FROM grade  
  
 GROUP BY code
```



Conditions on AGGREGATE FUNCTIONS

Where conditions operate **BEFORE** the group by.

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

id	code	mark
S104	IAI	65
S107	IAI	35
S107	PR1	76
S104	PR1	68
S107	PR2	60
S106	PR2	43

code	avg
IAI	48.5
PR1	72
PR2	51.5

```
SELECT code,  
       AVG(mark)  
  FROM grade  
 WHERE id != 'S103'  
 GROUP BY code
```

Conditions on AGGREGATE FUNCTIONS

New keyword **HAVING**
operates **after** the group
by.

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

id	code	mark
S104	IAI	65
S107	IAI	35
S107	PR1	76
S104	PR1	68
S107	PR2	60
S106	PR2	43

code	avg
IAI	48.5
PR1	72
PR2	51.5

code	avg
PR1	72
PR2	51.5

```
SELECT code,  
       AVG(mark)  
  FROM grade  
 WHERE id != 'S103'  
 GROUP BY code  
 HAVING AVG(mark) > 50
```

We now know a lot...

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
- **JOIN** - create new tables of "composite entities"
- **GROUP BY** - creates new entities (rows) by grouping entities and computing specified group (aggregate) facts.
- **HAVING** - filters the result of group by

All operate on a single table.

All return a single table.

Think in tables, entities & attributes...

We now know a lot...

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
- **JOIN** - create new tables of "composite entities"
- **GROUP BY** - creates new entities (rows) by grouping entities and computing specified group (aggregate) facts.
- **HAVING** - filters the result of group by

All operate on a single table.

All return a single table.

Think in tables, entities & attributes...

Combining this with joins...

1st we can join two (or more) tables.

We then have **a single table**.

Then we can filter them.

Then we can group them.

Then the required columns are extracted and **aggregates** calculated.

```
SELECT student.id,  
       AVG(grade.mark)  
  FROM student, grade  
 WHERE student.id = grade.id  
 GROUP BY student.id
```

Combining this with joins...

1st we can join two (or more) tables.

We then have **a single table**.

Then we can filter them.

Then we can group them.

Then the required columns are extracted and **aggregates** calculated.

```
SELECT student.id,  
       AVG(grade.mark)  
  FROM student, grade  
 WHERE student.id = grade.id  
 GROUP BY student.id
```

Combining this with joins...

1st we can join two (or more) tables.

We then have **a single table**.

Then we can filter them.

Then we can group them.

Then the required columns are extracted and **aggregates** calculated.

```
SELECT student.id,  
       AVG(grade.mark)  
  FROM student, grade  
 WHERE student.id = grade.id  
 GROUP BY student.id
```

Combining this with joins...

1st we can join two (or more) tables.

We then have **a single table**.

Then we can filter them.

Then we can group them.

Then the required columns are extracted and **aggregates** calculated.

```
SELECT student.id,  
       AVG(grade.mark)  
  FROM student, grade  
 WHERE student.id = grade.id  
 GROUP BY student.id
```

```
SELECT student.id,  
       AVG(grade.mark)  
  FROM student, grade  
 WHERE student.id = grade.id  
 GROUP BY student.id
```

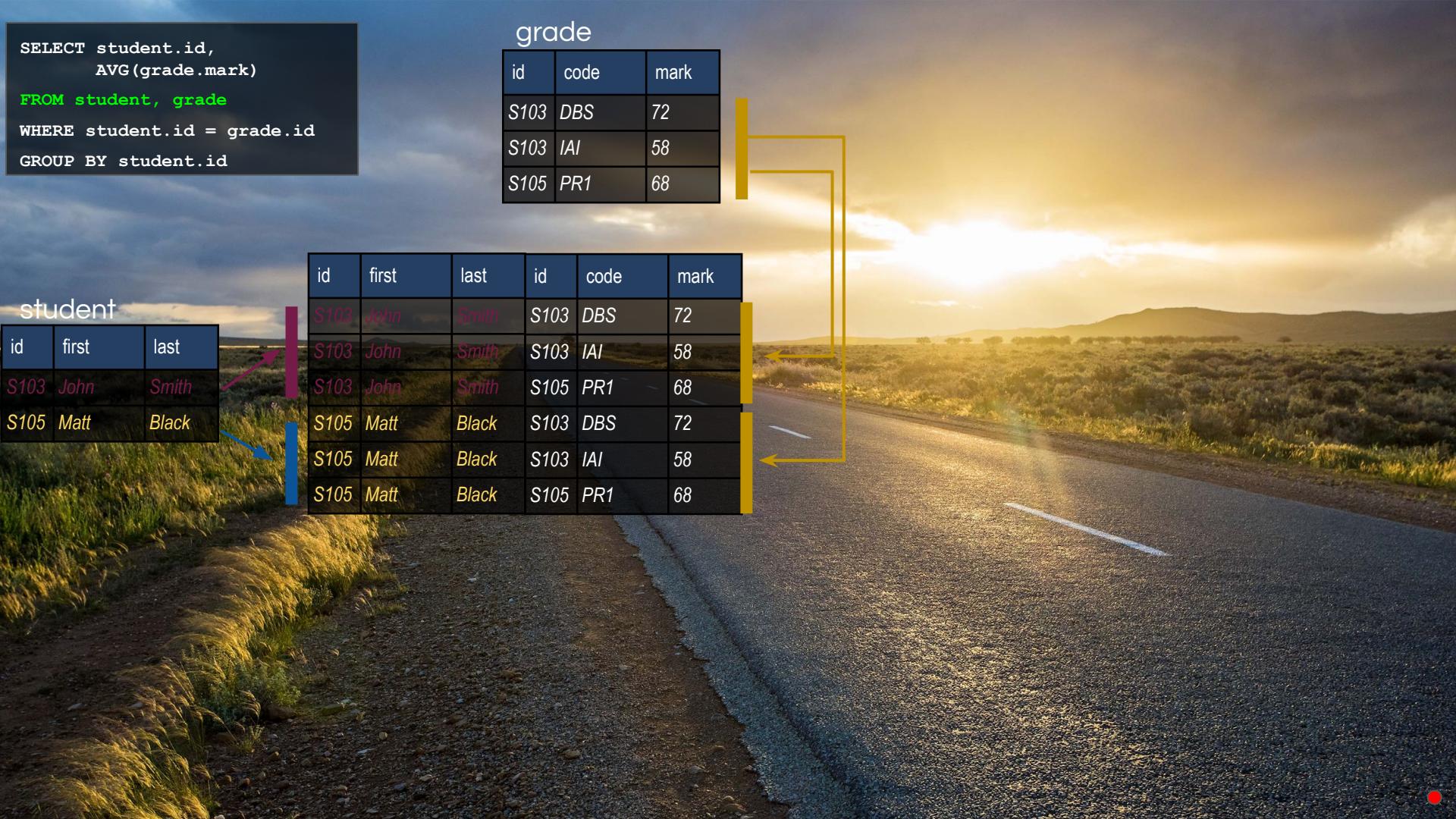
grade

id	code	mark
S103	DBS	72
S103	IAI	58
S105	PR1	68

student

id	first	last
S103	John	Smith
S105	Matt	Black

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S103	John	Smith	S105	PR1	68
S105	Matt	Black	S103	DBS	72
S105	Matt	Black	S103	IAI	58
S105	Matt	Black	S105	PR1	68



```
SELECT student.id,  
       AVG(grade.mark)  
  FROM student, grade  
 WHERE student.id = grade.id  
 GROUP BY student.id
```

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S105	PR1	68

```
SELECT *
```

```
FROM student, grade
```

```
WHERE (student.id = grade.id)
```



student

id	first	last
S103	John	Smith
S105	Matt	Black

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S103	John	Smith	S105	PR1	68
S105	Matt	Black	S103	DBS	72
S105	Matt	Black	S103	IAI	58
S105	Matt	Black	S105	PR1	68

```
SELECT *
```

```
FROM student, grade
```

```
WHERE (student.id = grade.id)
```



```
SELECT student.id,  
       AVG(grade.mark)  
  FROM student, grade  
 WHERE student.id = grade.id  
 GROUP BY student.id
```

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S105	PR1	68

```
SELECT *
```

```
FROM student, grade
```

```
WHERE (student.id = grade.id)
```



id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S105	Matt	Black	S105	PR1	68

student

id	first	last
S103	John	Smith
S105	Matt	Black

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S103	John	Smith	S105	PR1	68
S105	Matt	Black	S103	DBS	72
S105	Matt	Black	S103	IAI	58
S105	Matt	Black	S105	PR1	68

```
SELECT student.id,  
       AVG(grade.mark)  
  FROM student, grade  
 WHERE student.id = grade.id  
 GROUP BY student.id
```

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S105	PR1	68

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S105	Matt	Black	S105	PR1	68

student

id	first	last
S103	John	Smith
S105	Matt	Black

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S103	John	Smith	S105	PR1	68
S105	Matt	Black	S103	DBS	72
S105	Matt	Black	S103	IAI	58
S105	Matt	Black	S105	PR1	68



S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58



S105	Matt	Black	S105	PR1	68
------	------	-------	------	-----	----

```

SELECT student.id,
       AVG(grade.mark)
FROM student, grade
WHERE student.id = grade.id
GROUP BY student.id

```

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S105	PR1	68

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S105	Matt	Black	S105	PR1	68

student

id	first	last
S103	John	Smith
S105	Matt	Black

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S103	John	Smith	S105	PR1	68
S105	Matt	Black	S103	DBS	72
S105	Matt	Black	S103	IAI	58
S105	Matt	Black	S105	PR1	68

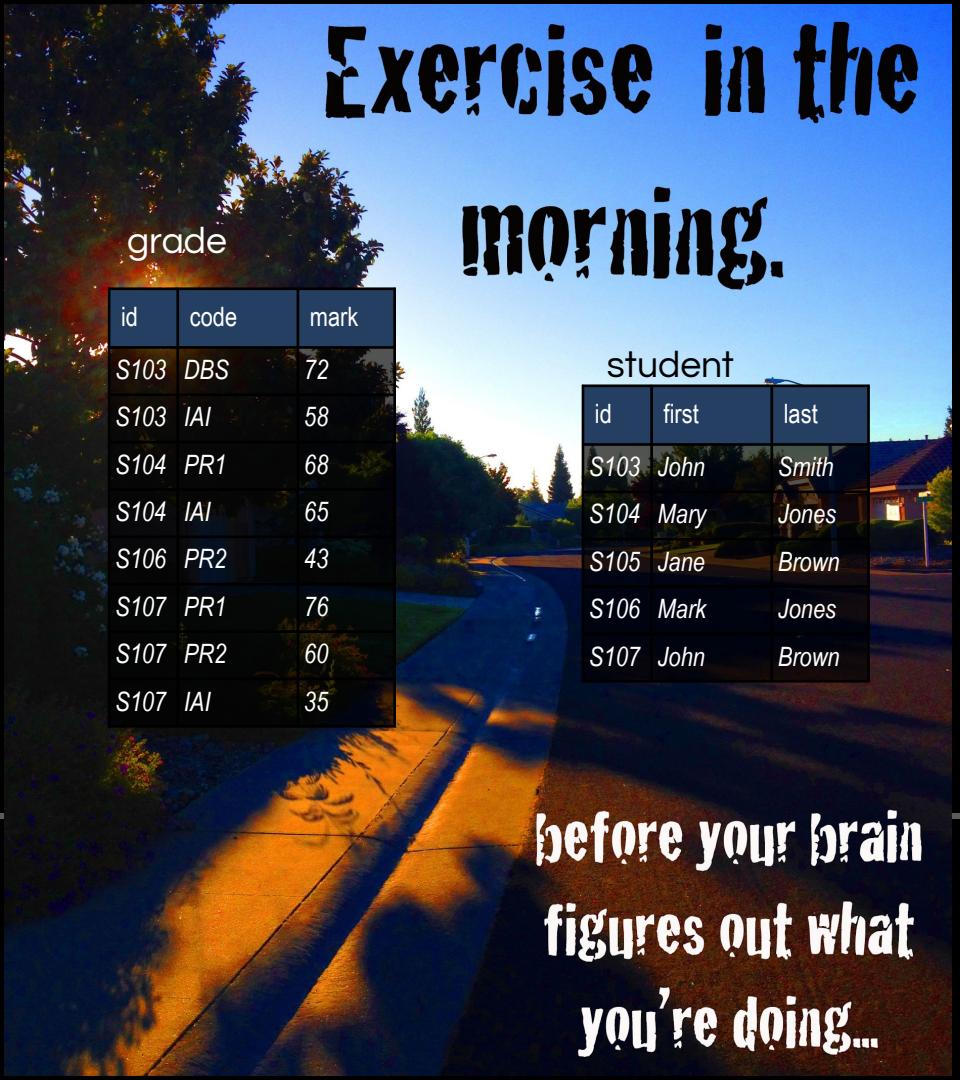


S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58



S105	Matt	Black	S105	PR1	68
------	------	-------	------	-----	----

id	AVG(mark)
S103	65
S105	68



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

morning.

student

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

before your brain
figures out what
you're doing...

**SELECT id
FROM student**

id

S103

S104

S105

S106

S107

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

id

S103

S104

S104

S107

S107

code

DBS

PR1

IAI

PR1

PR2

mark

72

68

65

76

60

**SELECT code, mark
FROM student, grade
WHERE first = 'John' and
last = 'Smith'**

code	mark
DBS	72
IAI	58

SELECT code, AVG(mark)

**FROM grade
GROUP BY code**

code

DBS

72

IAI

52.667

PR1

72

PR2

53

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

- JOINS via multiple tables in the FROM statement
- Filter to keep only "correct" rows

INNER JOIN

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

student

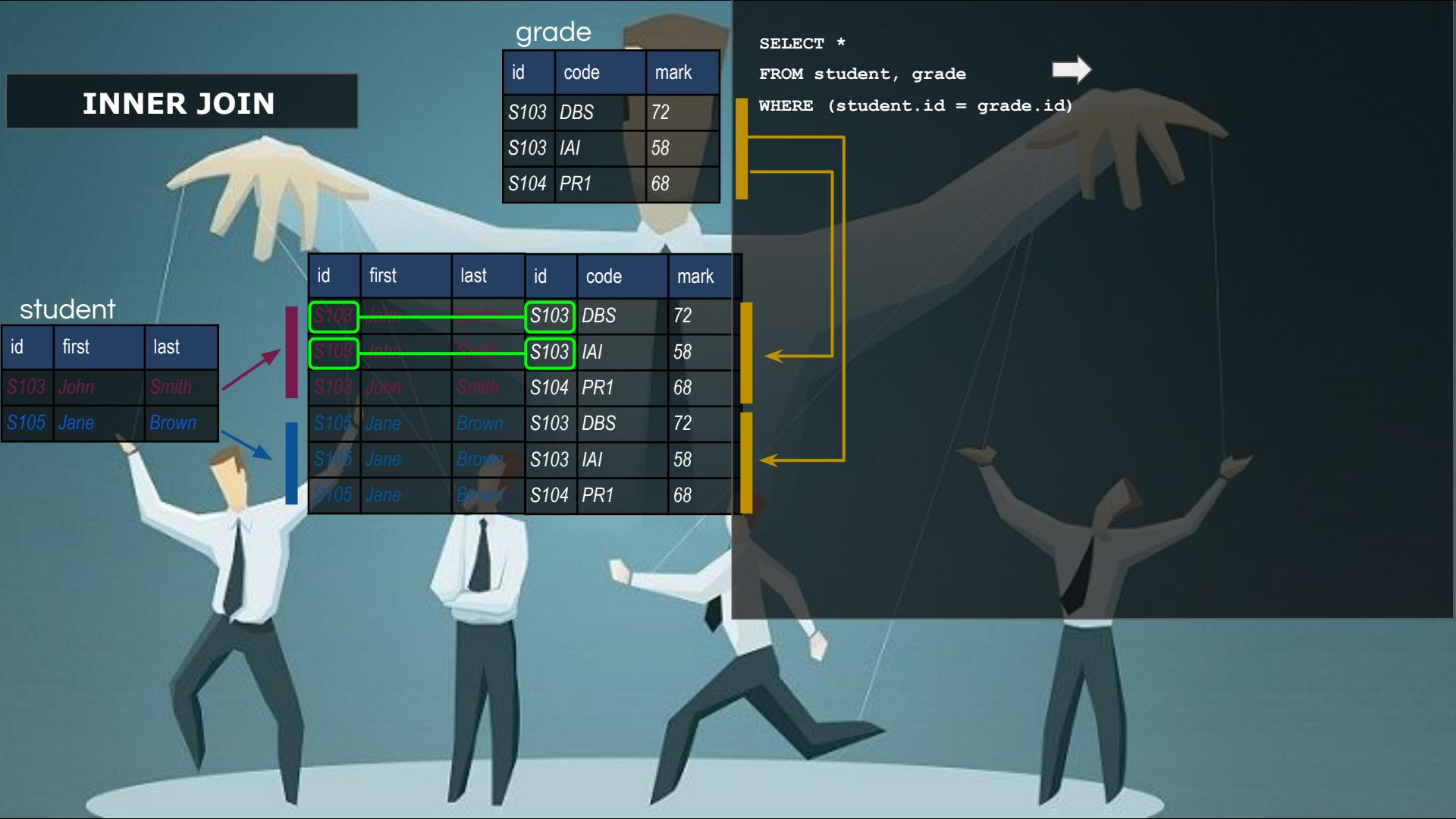
id	first	last
S103	John	Smith
S105	Jane	Brown

SELECT *

FROM student, grade

WHERE (student.id = grade.id)

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S103	John	Smith	S104	PR1	68
S105	Jane	Brown	S103	DBS	72
S105	Jane	Brown	S103	IAI	58
S105	Jane	Brown	S104	PR1	68



INNER JOIN

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

SELECT *

FROM student, grade

WHERE (student.id = grade.id)



id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58

student

id	first	last
S103	John	Smith
S105	Jane	Brown

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S103	John	Smith	S104	PR1	68
S105	Jane	Brown	S103	DBS	72
S105	Jane	Brown	S103	IAI	58
S105	Jane	Brown	S104	PR1	68



INNER JOIN

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

student

id	first	last
S103	John	Smith
S105	Jane	Brown

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S103	John	Smith	S104	PR1	68
S105	Jane	Brown	S103	DBS	72
S105	Jane	Brown	S103	IAI	58
S105	Jane	Brown	S104	PR1	68

```
SELECT *  
FROM student, grade  
WHERE (student.id = grade.id)
```

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58

What's wrong?

Lost information regarding:

- (1) S105, Jane, Brown
- (2) S104, PR1, 68

Might want to know that join did not match anything!

Select the students that have not recorded any grades.

student

id	first	last
S103	John	Smith
S105	Jane	Brown

```
SELECT *
FROM student, grade
WHERE (student.id = grade.id)
```

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

Let's consider rows in the original tables.

After the join we have...

**NLAB:***Data at Scale*

student

id	first	last
S103	John	Smith
S105	Jane	Brown

```
SELECT *
FROM student, grade
WHERE (student.id = grade.id)
```

Rows that matched and
were kept (1 or more times)

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

Let's consider rows in
the original tables.

After the join we
have...



id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58



NLAB:

Data at Scale

student

id	first	last
S103	John	Smith
S105	Jane	Brown

```
SELECT *
FROM student, grade
WHERE (student.id = grade.id)
```

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

Let's consider rows in the original tables.

After the join we have...

Rows in **student** that were never matched.



id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58



NLAB:

Data at Scale

student

id	first	last
S103	John	Smith
S105	Jane	Brown

```
SELECT *
FROM student, grade
WHERE (student.id = grade.id)
```

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

Let's consider rows in the original tables.

After the join we have...

Rows in
never

grade that were
matched.

id	code	mark
S104	PR1	68

id	first	last
S105	Jane	Brown

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58



NLAB:

Data at Scale

student

id	first	last
S103	John	Smith
S105	Jane	Brown

```
SELECT *
FROM student, grade
WHERE (student.id = grade.id)
```

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

Let's explicitly encode a "did not match anything"

id	first	last
S105	Jane	Brown

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58

id	code	mark
S104	PR1	68



NLAB:

Data at Scale

student

id	first	last
S103	John	Smith
S105	Jane	Brown

```
SELECT *
FROM student, grade
WHERE (student.id = grade.id)
```

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

Let's explicitly encode a "did not match anything"

By matching the line to a set of NULL* values.

id	first	last	id	code	mark
S105	Jane	Brown	NULL	NULL	NULL

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58

id	first	last	id	code	mark
NULL	NULL	NULL	S104	PR1	68

* Recall NULL ≈ unknown value



NLAB:

Data at Scale

student

id	first	last
S103	John	Smith
S105	Jane	Brown

```
SELECT *
```

```
FROM student, grade  
WHERE (student.id = grade.id)
```

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

Back to JOINS

INNER JOIN

Only including matches is called an inner join.

Join all rows to all rows then filter on a match condition.



NLAB:

Data at Scale

student

id	first	last
S103	John	Smith
S105	Jane	Brown

```
SELECT *
FROM student
JOIN grade
ON (student.id = grade.id)
```

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

New, equivalent syntax!

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58

Back to JOINS INNER JOIN

Only including matches is called an inner join.

Join all rows to all rows then filter on a match condition.



NLAB:

Data at Scale

student

id	first	last
S103	John	Smith
S105	Jane	Brown

```
SELECT *
FROM student
JOIN grade
ON (student.id = grade.id)
```

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

New, equivalent syntax!

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58

Back to JOINS INNER JOIN

Only including matches is called an inner join.

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58

Join all rows to all rows then filter on a match condition.



NLAB:

Data at Scale

student

id	first	last
S103	John	Smith
S105	Jane	Brown

```
SELECT *
FROM student
LEFT OUTER JOIN grade
ON (student.id = grade.id)
```

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

Back to JOINS LEFT OUTER JOIN

Join all rows to all rows then filter on a match condition.

THEN add any missing rows in the left table.

id	first	last	id	code	mark
S105	Jane	Brown	NULL	NULL	NULL

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58

**NLAB:***Data at Scale*

student

id	first	last
S103	John	Smith
S105	Jane	Brown

```
SELECT *
FROM student
LEFT OUTER JOIN grade
ON (student.id = grade.id)
```

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

Back to JOINS LEFT OUTER JOIN

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S105	Jane	Brown	NULL	NULL	NULL

Join all rows to all rows then filter on a match condition.

THEN add any missing rows in the left table.

id	first	last	id	code	mark
S105	Jane	Brown	NULL	NULL	NULL

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58



NLAB:

Data at Scale

student

id	first	last
S103	John	Smith
S105	Jane	Brown

```
SELECT *
FROM student
RIGHT OUTER JOIN grade
ON (student.id = grade.id)
```

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

Back to JOINS
RIGHT OUTER JOIN

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
NULL	NULL	NULL	S104	PR1	68

Join all rows to all rows then
filter on a match condition.

THEN add any
missing rows in the
right table.

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58

id	first	last	id	code	mark
NULL	NULL	NULL	S104	PR1	68



NLAB:

Data at Scale

student

id	first	last
S103	John	Smith
S105	Jane	Brown

```
SELECT *
FROM student
FULL OUTER JOIN grade
ON (student.id = grade.id)
```

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

Back to JOINS
FULL OUTER JOIN

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S105	Jane	Brown	NULL	NULL	NULL
NULL	NULL	NULL	S104	PR1	68

Join all rows to all rows then
filter on a match condition.

THEN add any
missing rows in both
the left and right
tables

id	first	last	id	code	mark
S105	Jane	Brown	NULL	NULL	NULL

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58

id	first	last	id	code	mark
NULL	NULL	NULL	S104	PR1	68

student

id	first	last
S103	John	Smith
S105	Jane	Brown

```
SELECT *
FROM student
FULL OUTER JOIN grade
ON (student.id = grade.id)
```

If the field is the same in the two tables, we can use the syntax USING instead of ON.

This has the advantage that the output will only include one of the fields student.id and grade.id as we have told it is the same!

```
SELECT *
FROM student
FULL OUTER JOIN grade
USING (id)
```

id	first	last	id	code	mark
S105	Jane	Brown	NULL	NULL	NULL

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58

grade

id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

Back to JOINS FULL OUTER JOIN

id	first	last	id	code	mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S105	Jane	Brown	NULL	NULL	NULL
NULL	NULL	NULL	S104	PR1	68

Join all rows to all rows then filter on a match condition.

THEN add any missing rows in both the left and right tables



NLAB:

Dept of Science



**Exercise in the
morning.**

**before your brain
figures out what
you're doing...**



Session 9

Relational Databases (SQL III)

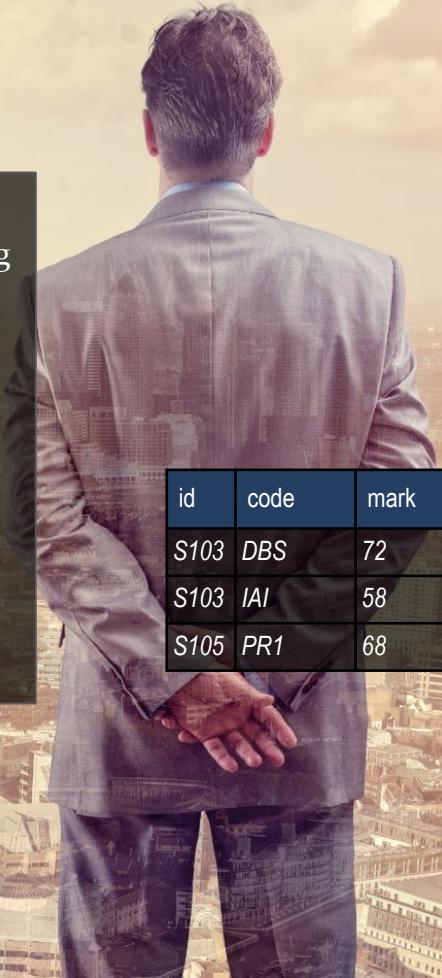


Last week: SQL

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** - return the specified column(s) from a table
- **WHERE** - filters row(s) by condition
- **JOIN... ON... USING...** Join two tables together ON (where) the given condition is true
- **GROUP BY** - creates new entities (rows) by grouping entities and computing specified group (aggregate) facts.
- **HAVING** - filters the result of group by

id	code	mark
S103	DBS	72
S103	IAI	58
S105	PR1	68



This week: Advanced SQL

Advanced

SELECT, WHERE,
GROUP BY, HAVING...
statements

Date and Time
functions

Real world examples

How to write longer
queries...



Advanced SELECT

Let's consider the way a
basic SELECT
SQL statement is executed
by the computer...

grade		
id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68



```
SELECT id, mark  
FROM grade
```

- 1) Computer starts at first record
(where is first? random from your point of view, ordering not guaranteed)

Advanced SELECT

Let's consider the way a basic SELECT SQL statement is executed by the computer...

grade		
id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

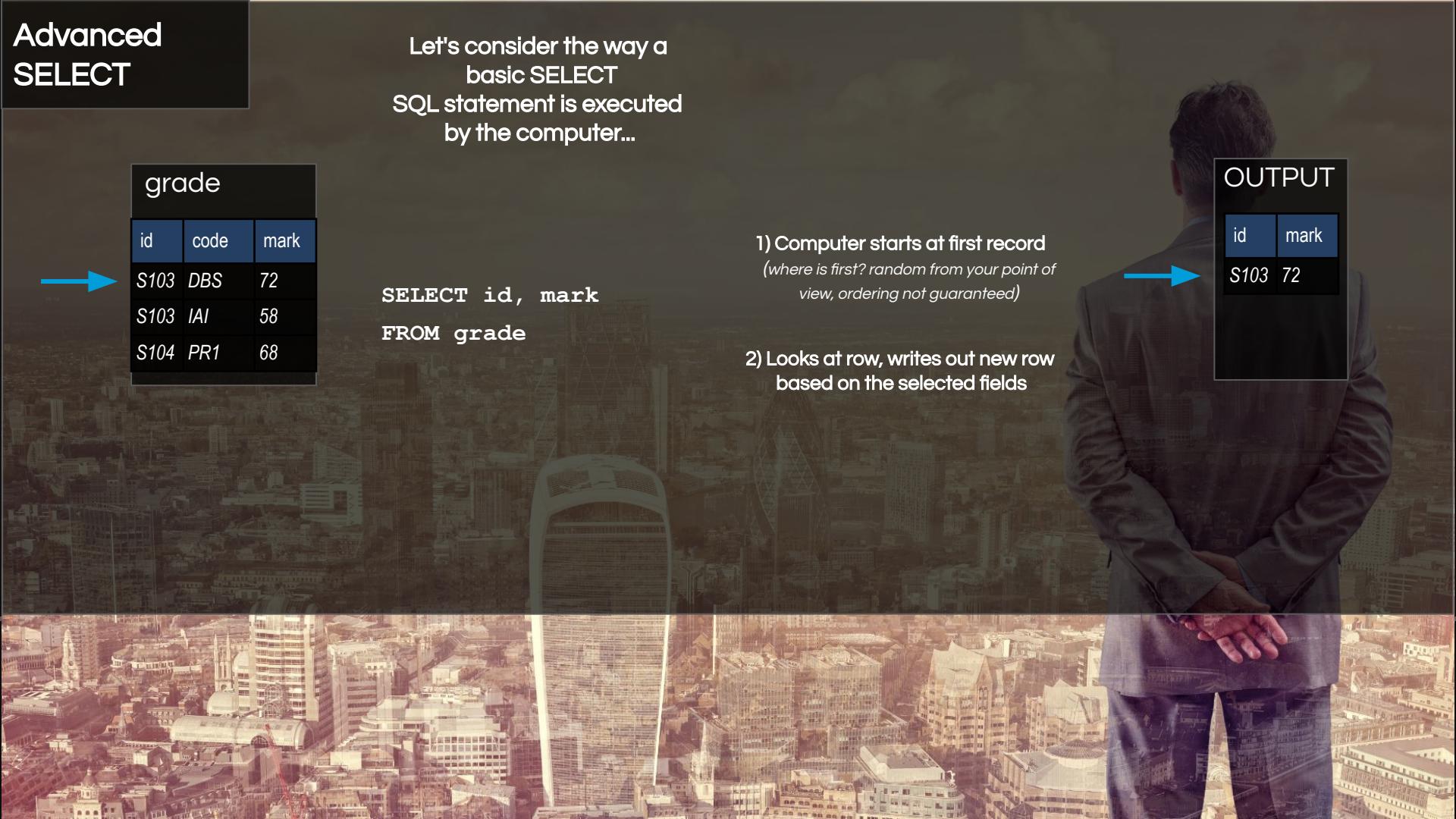
```
SELECT id, mark  
FROM grade
```

- 1) Computer starts at first record
(where is first? random from your point of view, ordering not guaranteed)

- 2) Looks at row, writes out new row based on the selected fields

OUTPUT

id	mark
S103	72



Advanced SELECT

Let's consider the way a basic SELECT SQL statement is executed by the computer...

grade		
id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

```
SELECT id, mark  
FROM grade
```

1) Computer starts at first record
(where is first? random from your point of view, ordering not guaranteed)

2) Looks at row, writes out new row based on the selected fields

OUTPUT

id	mark
S103	72
S103	58

Advanced SELECT

Let's consider the way a basic SELECT SQL statement is executed by the computer...

grade		
id	code	mark
S103	DBS	72
S103	IAI	58
S104	PR1	68

```
SELECT id, mark  
FROM grade
```

Take home message:

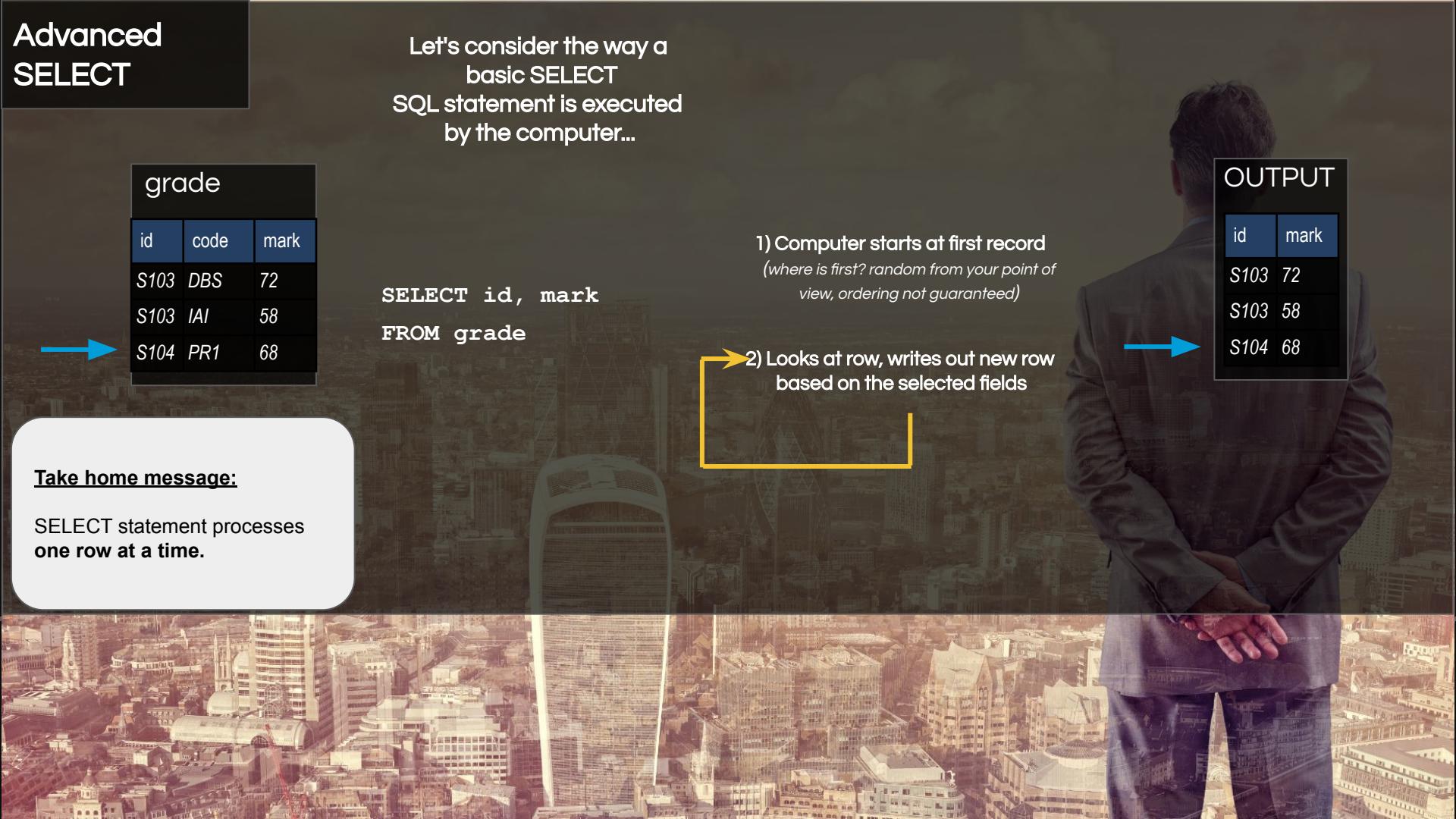
SELECT statement processes one row at a time.

1) Computer starts at first record
(where is first? random from your point of view, ordering not guaranteed)

2) Looks at row, writes out new row based on the selected fields

OUTPUT

id	mark
S103	72
S103	58
S104	68



Advanced SELECT

Take home message:

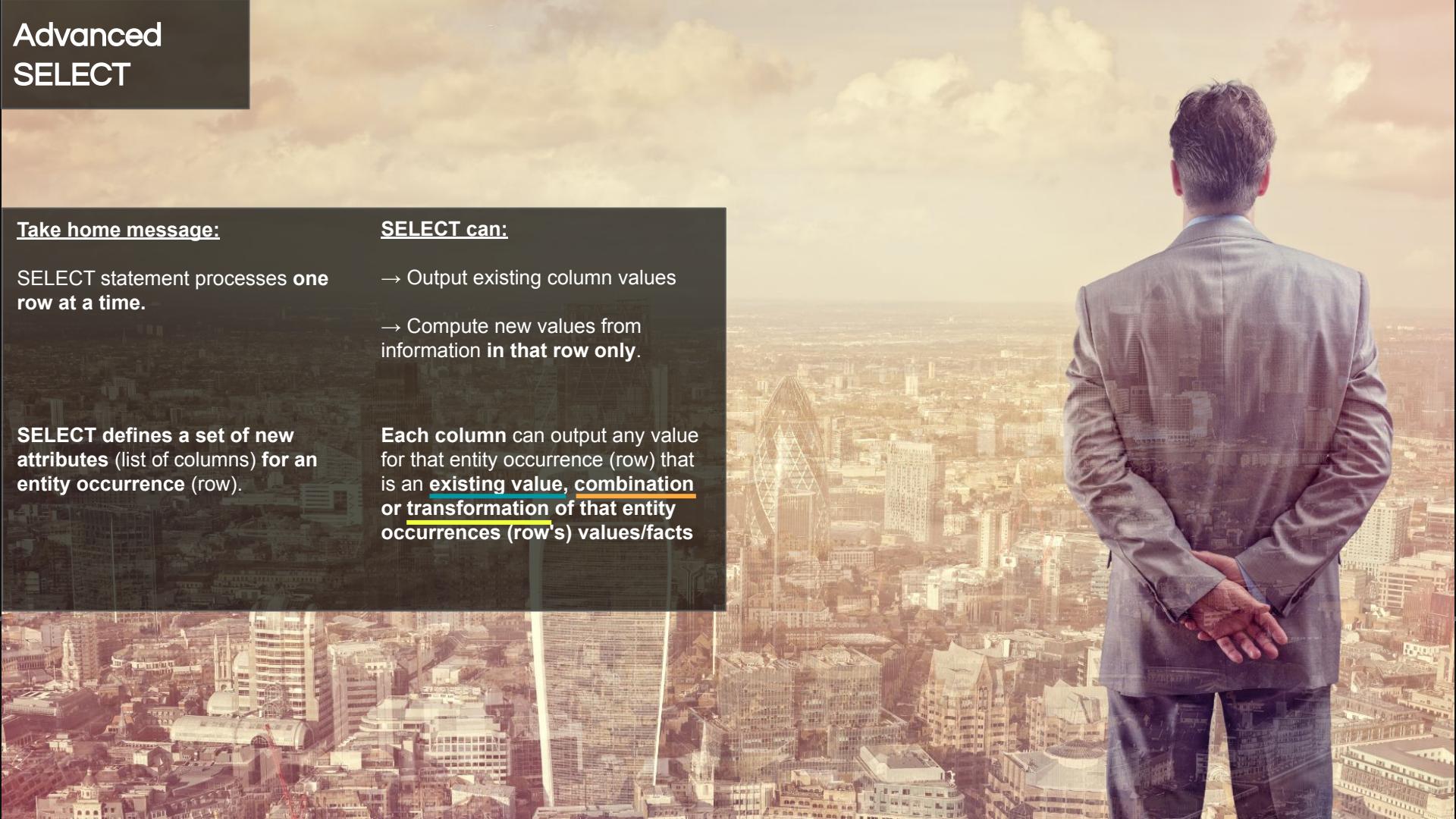
SELECT statement processes **one row at a time.**

SELECT defines a set of new attributes (list of columns) for an entity occurrence (row).

SELECT can:

- Output existing column values
- Compute new values from information **in that row only.**

Each column can output any value for that entity occurrence (row) that is an **existing value, combination or transformation of that entity occurrences (row's) values/facts**



Advanced SELECT

Take home message:

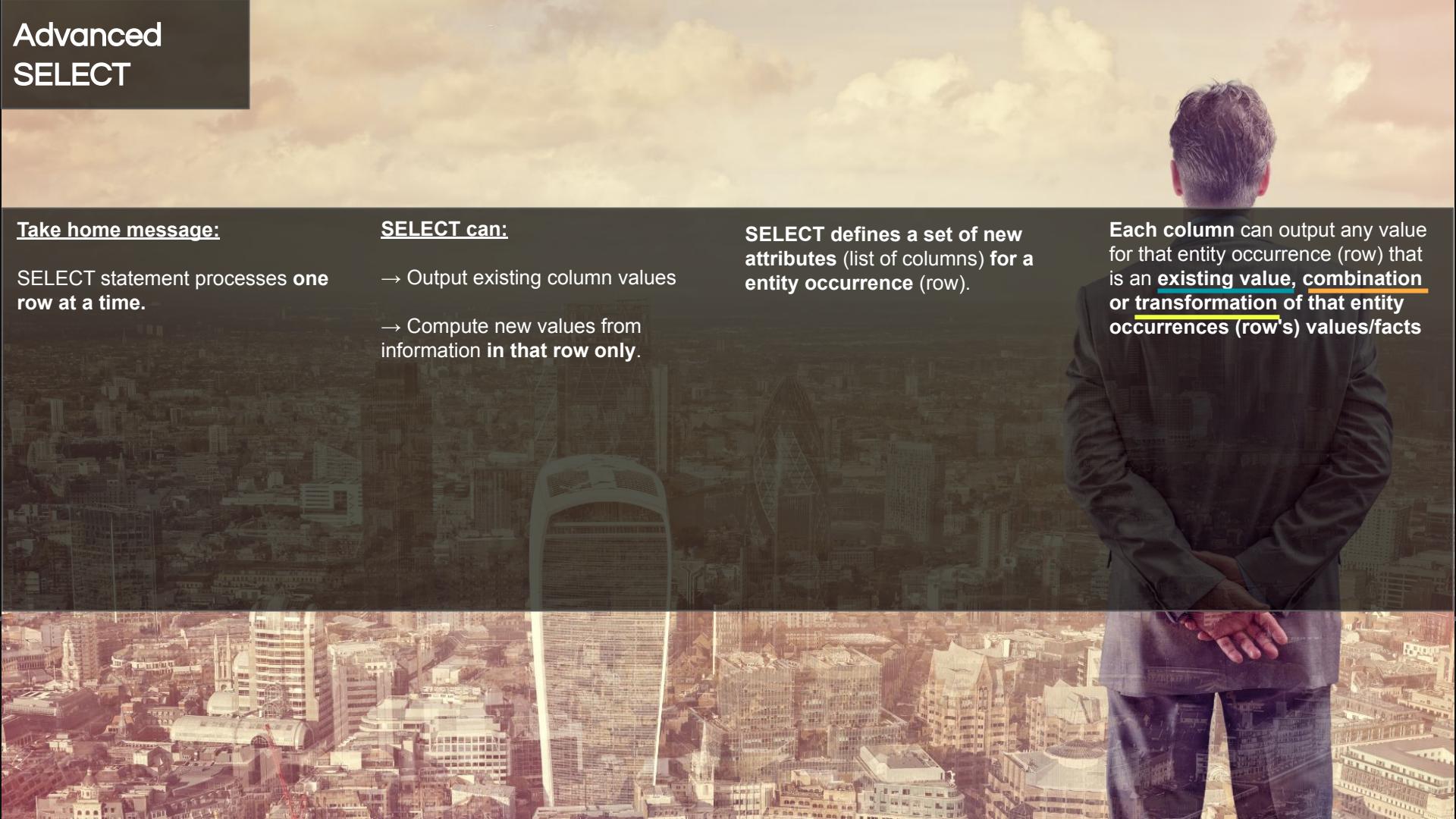
SELECT statement processes **one row at a time**.

SELECT can:

- Output existing column values
- Compute new values from information **in that row only**.

SELECT defines a set of new attributes (list of columns) **for a entity occurrence** (row).

Each column can output any value for that entity occurrence (row) that is an **existing value, combination or transformation of that entity occurrences** (row's) values/facts



Advanced SELECT

Take home message:

SELECT statement processes **one row at a time.**

Can output any value for that entity occurrence (row) that is an existing value, combination or transformation of that entity occurrences (row's) values/facts

Example:

Existing
value

grade

	grade		
	id	code	mark1
→	S103	DBS	72
	S103	IAI	58
	S104	PR1	68
			0

```
SELECT id, mark1  
FROM grade
```

OUTPUT

id	mark1
S103	72



Advanced SELECT

Take home message:

SELECT statement processes **one row at a time.**

Can output any value for that entity occurrence (row) that is an existing value, combination or transformation of that entity occurrences (row's) values/facts

Example:
Combination →

grade			
id	code	mark1	mark2
S103	DBS	72	70
S103	IAI	58	61
S104	PR1	68	0

```
SELECT code || ':' || mark1::STRING AS mylabel  
FROM grade
```

OUTPUT

mylabel
DBS: 72

Advanced SELECT

Take home message:

SELECT statement processes **one row at a time.**

Can output any value for that entity occurrence (row) that is an existing value, combination or transformation of that entity occurrences (row's) values/facts

Example: Combination →

grade			
id	code	mark1	mark2
S103	DBS	72	70
S103	IAI	58	61
S104	PR1	68	0

```
SELECT code || ':' || mark1::STRING AS mylabel  
FROM grade
```

|| is the concatenation operator
(we've seen this before!)

::STRING converts the mark
(which was an INTEGER) to
text before concatenation.

OUTPUT

mylabel
DBS: 72

Also seen this before too!
If you omit this, the computer will
assume you meant this in this
case. Not always the case though.
So good practice to include it.

Advanced SELECT

Take home message:

SELECT statement processes one row at a time.

Can output any value for that entity occurrence (row) that is an existing value, combination or transformation of that entity occurrences (row's) values/facts

Why 2.0 not 2?

Not strictly required for SparkSQL. For others, i.e. PostgreSQL you do... Why?

Since mark1 and mark2 are INTEGER fields, if you divide by 2 (another INTEGER) the computer will do INTEGER division. I.e. $1/2 = 0$ We make one a numeric to ensure normal division occurs.

Example:

Existing value and Combination →

grade			
id	code	mark1	mark2
S103	DBS	72	70
S103	IAI	58	61
S104	PR1	68	0

```
SELECT id, (mark1 + mark2) / 2.0 AS avgmark  
FROM grade
```

OUTPUT

id	avgmark
S103	71.0
S103	59.5

Advanced SELECT

Take home message:

SELECT statement processes one row at a time.

REMEMBER: Each entity occurrence is processed separately.

i.e. we go line-by-line

Example:

Existing value and Combination

grade

	id	code	mark1	mark2
→	S103	DBS	72	70
	S103	IAI	58	61
	S104	PR1	68	0

```
SELECT id, (mark1 + mark2) / 2.0 AS avgmark
FROM grade
```

OUTPUT

	id	avgmark
→	S103	71.0
	S103	59.5

Advanced SELECT

Take home message:

SELECT statement processes one row at a time.

Can output any value for that entity occurrence (row) that is an existing value, combination or transformation of that entity occurrences (row's) values/facts

Transformations are via **FUNCTIONS**.

Functions take a set of values and parameters and return a new value.

postgres define a number of functions, or you can write your own.

A common type of builtin functions are formatting functions.

i.e. we have a date, and want to format it differently.

Again, we've seen this before. There are many more transformations.

student

Example:

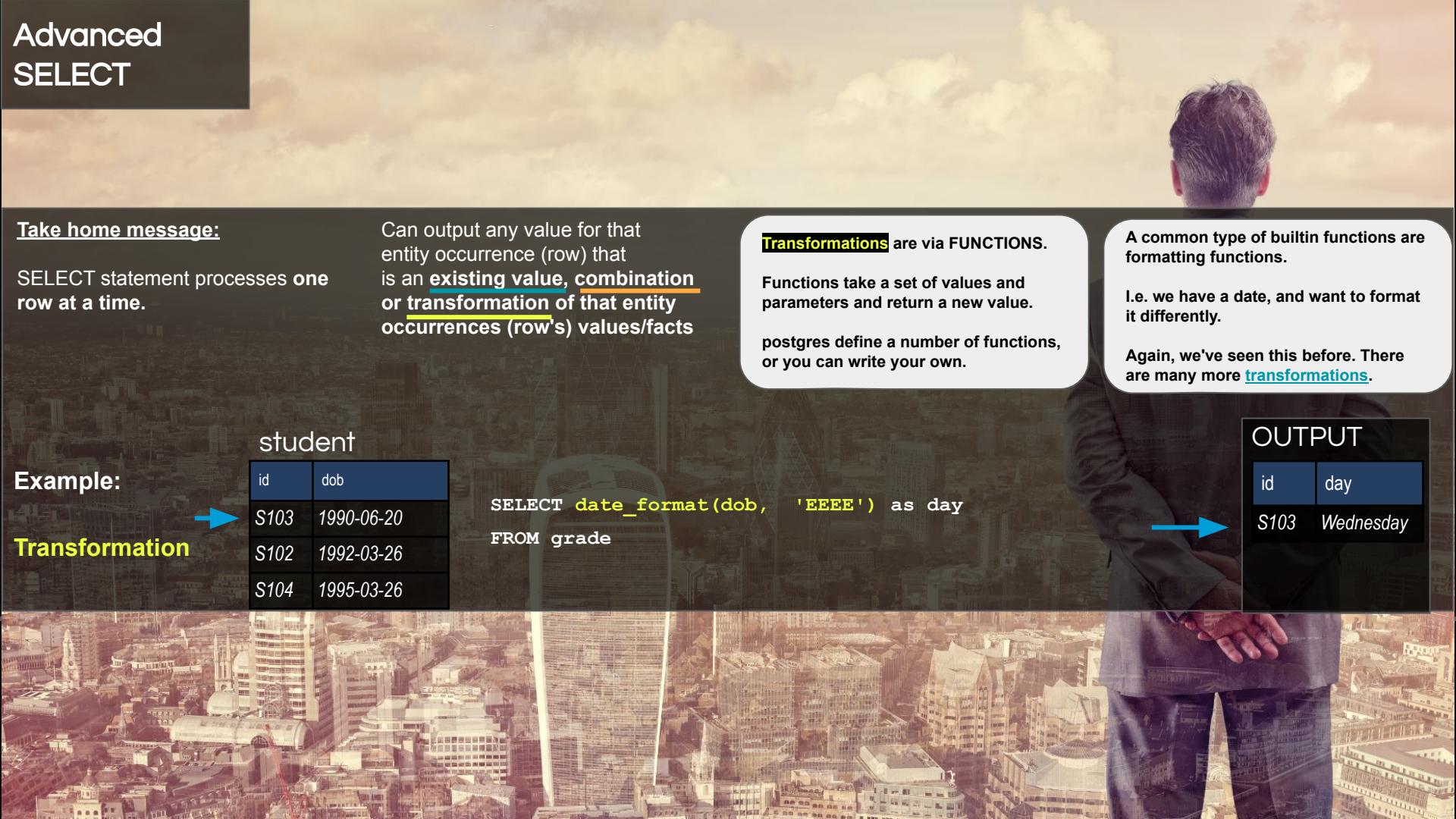
Transformation

	id	dob
→	S103	1990-06-20
	S102	1992-03-26
	S104	1995-03-26

```
SELECT date_format(dob, 'EEEE') as day  
FROM grade
```

OUTPUT

	id	day
→	S103	Wednesday



Advanced SELECT

But I don't know what functions there are... If you think one should exist, then:
→ we try google... e.g. "[SparkSQL format date](#)"
→ we go directly to the [documentation for SparkSQL](#)

Take home message:

SELECT statement processes one

Spark SQL Built-In Functions

Search docs

Functions

!
!=
%
&
*
+
-
/
<
<=

Transformation

date_format

date_format(timestamp, fmt) - Converts timestamp to a value of string in the format specified by the date format fmt.

Arguments:

- timestamp - A date/timestamp or string to be converted to the given format.
- fmt - Date/time format pattern to follow. See [Datetime Patterns](#) for valid date and time format patterns.

Examples:

```
> SELECT date_format('2016-04-08', 'y');  
2016
```

Since: 1.5.0

Can output any value for that entity occurrence (row) that is an **existing value combination**

Transformations are via **FUNCTIONS**.

Functions take a set of values and parameters and return a new value.

postgres define a number of functions, or you can write your own.

A common type of builtin functions are formatting functions.

i.e. we have a date, and want to format it differently

y
cts

date_format(dob, 'EEEE') as day

OUTPUT

id	day
S103	Wednesday

S102	1992-03-26
S104	1995-03-26



Advanced SELECT

Take home message:

SELECT statement processes one row at a time.

Can output any value for that entity occurrence (row) that is an existing value, combination or transformation of that entity occurrences (row's) values/facts

Example:

id	dob
S103	1990-06-20
S102	1992-03-26
S104	1995-03-26

Transformation

```
SELECT date_format(dob, 'EEEE') as day  
FROM grade
```

Datetime Patterns for Formatting and Parsing

There are several common scenarios for datetime usage in Spark:

- CSV/JSON datasources use the pattern string for parsing and formatting datetime content.
- Datetime functions related to convert StringType to/from DateType or TimestampType. For example, unix_timestamp, date_format, to_unix_timestamp, from_unixtime, to_date, to_timestamp, from_utc_timestamp, to_utc_timestamp, etc.

Spark uses pattern letters in the following table for date and timestamp parsing and formatting:

Symbol	Meaning	Presentation	Examples
G	era	text	AD; Anno Domini
y	year	year	2020; 20
D	day-of-year	number(3)	189
M/L	month-of-year	month	7; 07; Jul; July
d	day-of-month	number(2)	28
Q/q	quarter-of-year	number/text	3; Q3; Q3: 3rd quarter
E	day-of-week	text	Tue; Tuesday
F	aligned day of week in month	number(1)	
z	am/pm offset	am/pm	
Z	zone-offset	offset-Z	-0800; -08:00;
'	escape for text	delimiter	
"	single quote	literal	
[optional section start		
]	optional section end		

The count of pattern letters determines the format.

- Text: The text style is determined based on the number of pattern letters used. Less than 4 pattern letters will use the short text form, typically an abbreviation, e.g. day-of-week Monday might output "Mon". Exactly 4 pattern letters will use the full text form, typically the full description, e.g. day-of-week Monday might output "Monday". 5 or more letters will fail.
- Number(n): The n here represents the maximum count of letters this type of datetime pattern can be used.
 - In formatting, if the count of letters is one, then the value is output using the minimum number of digits and without padding otherwise, the count of digits is used as the width of the output field, with the value zero-padded as necessary.
 - In parsing, the exact count of digits is expected in the input field.
- Number/Text: If the count of pattern letters is 3 or greater, use the Text rules above. Otherwise use the Number rules above.
- Fraction: Use one or more (up to 9) contiguous 'S' characters, e.g SSSSSS, to parse and format fraction of second. For parsing, the acceptable fraction length can be [1, the number of contiguous 'S']. For formatting, the fraction length would be padded to the number of contiguous 'S' with zeros. Spark supports datetime of micro-of-second

why
4x
Es?

Advanced SELECT

```
SELECT id,  
       'Born on a ' || date_format(dob, 'EEEE') || ', started studying on a ' || date_format(start_date, 'EEEE') as info  
FROM student
```

Example:
Existing value,
combination &
transformation

student

id	dob	start_date
S103	1990-06-20	2001-07-02
S102	1992-03-26	2010-06-30
S104	1995-03-26	2011-08-03

OUTPUT

id	info
S103	Born on a Wednesday, started studying on a Sunday

Advanced SELECT

REMEMBER: Each entity occurrence is processed separately.

i.e. we go line-by-line

```
SELECT id,  
       'Born on a ' || date_format(dob, 'EEEE') || ', started studying on a ' || date_format(start_date, 'EEEE') as info  
FROM student
```

Example:
**Existing value,
combination &
transformation** →

student

id	dob	start_date
S103	1990-06-20	2001-07-02
S102	1992-03-26	2010-06-30
S104	1995-03-26	2011-08-03

OUTPUT

id	info
S103	Born on a Wednesday, started studying on a Monday
S102	Born on a Thursday, started studying on a Wednesday

Advanced SELECT

REMEMBER: Each entity occurrence is processed separately.

i.e. we go line-by-line

```
SELECT id,  
       'Born on a ' || date_format(dob, 'EEEE') || ', started studying on a ' || date_format(start_date, 'EEEE') as info  
FROM student
```

Example:
Existing value,
combination &
transformation →

student

id	dob	start_date
S103	1990-06-20	2001-07-02
S102	1992-03-26	2010-06-30
S104	1995-03-26	2011-08-03

OUTPUT

id	info
S103	Born on a Wednesday, started studying on a Monday
S102	Born on a Thursday, started studying on a Wednesday
S104	Born on a Sunday, started studying on a Wednesday

Advanced WHERE

OK, so that was SELECT.

What about WHERE?

Recall:

- Filters (keeps or omits) rows based on a given condition.

Same as SELECT, rows are processed independently.

SELECT:

- we are deciding what to output given the facts for an entity occurrence

WHERE:

- we are deciding whether to *keep* the occurrence entity given the facts for that entity occurrence

A WHERE condition can therefore include:

- existing value,
- combination or
- transformations

of that entity occurrences (row's values/facts).

As with SELECT, it can also be a mix of these.

Advanced WHERE

```
SELECT *  
FROM student  
WHERE date_format(start_date, 'EEEE') = 'Wednesday' AND NOT id = 'S102'
```

Example:
Existing value
&
Transformation

student

id	dob	start_date
S103	1990-06-20	2001-07-02
S102	1992-03-26	2010-06-30
S104	1995-03-26	2011-08-03

id	dob	start_date
S103	1990-06-20	2001-07-02

2001-07-02 is NOT a Wednesday

Advanced WHERE

REMEMBER: Each entity occurrence is processed separately.

I.e. we go line-by-line

```
SELECT *  
FROM student  
WHERE date_format(start_date, 'EEEE') = 'Wednesday' AND NOT id = 'S102'
```

Example:
Existing value
&
Transformation

student

id	dob	start_date
S103	1990-06-20	2001-07-02
S102	1992-03-26	2010-06-30
S104	1995-03-26	2011-08-03

id	dob	start_date
S103	1990-06-20	2001-07-02
S102	1992-03-26	2010-06-30

Record is S102

Advanced WHERE

REMEMBER: Each entity occurrence is processed separately.

I.e. we go line-by-line

```
SELECT *  
FROM student  
WHERE date_format(start_date, 'EEEE') = 'Wednesday' AND NOT id = 'S102'
```

Example:
Existing value
&
Transformation →

student

id	dob	start_date
S103	1990-06-20	2001-07-02
S102	1992-03-26	2010-06-30
S104	1995-03-26	2011-08-03

id	dob	start_date
S103	1990-06-20	2001-07-02
S102	1992-03-26	2010-06-30
S104	1995-03-26	2011-08-03

Record is not S102 & 2011-08-03
is a Wednesday!

Advanced GROUP BY

OK, so that was SELECT.
And WHERE.

What about GROUP BY?

Recall:

→ GROUP BY specifies the column for which all distinct values will become bucket labels

More generally, however,
GROUP BY defines the measure for which entity occurrences (rows) will be considered the same for the purpose of grouping.

As such, per occurrence (row), this measure can be:

- existing value,
- combination or
- transformations

EXACTLY THE SAME THINGS AS:
→ SELECT
→ WHERE

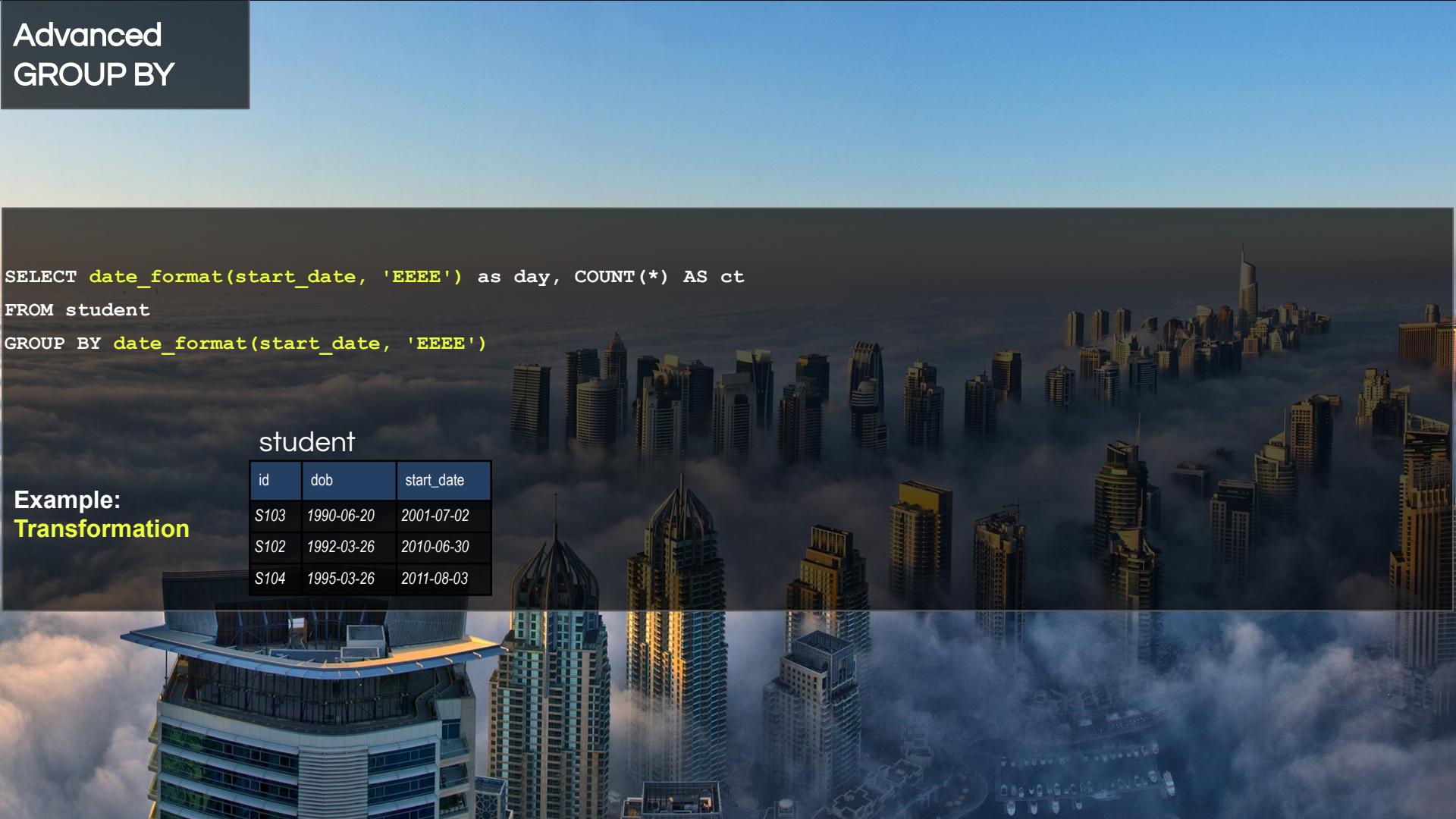
Advanced GROUP BY

```
SELECT date_format(start_date, 'EEEE') as day, COUNT(*) AS ct
FROM student
GROUP BY date_format(start_date, 'EEEE')
```

Example:
Transformation

student

id	dob	start_date
S103	1990-06-20	2001-07-02
S102	1992-03-26	2010-06-30
S104	1995-03-26	2011-08-03



Advanced GROUP BY

Recall: When using GROUP BY
SELECT can only have bucket labels
or
results of aggregate functions.

```
SELECT date_format(start_date, 'EEEE') as day, COUNT(*) as ct
FROM student
GROUP BY date_format(start_date, 'EEEE')
```

student

id	dob	start_date
S103	1990-06-20	2001-07-02
S102	1992-03-26	2010-06-30
S104	1995-03-26	2011-08-03

**Example:
Transformation**



Advanced GROUP BY

Recall: When using GROUP BY
SELECT can only have bucket labels
or
results of aggregate functions.

```
SELECT date_format(start_date, 'EEEE') as day, COUNT(*) as ct
FROM student
GROUP BY date_format(start_date, 'EEEE')
```

Example:
Transformation

id	dob	start_date
S103	1990-06-20	2001-07-02
S102	1992-03-26	2010-06-30
S104	1995-03-26	2011-08-03

NOTE the difference:

Aggregate functions:

Take a group of entity occurrences (rows), return a value summarizing the group.

Transformation functions:

Transform facts/values for a single entity occurrence (row)

Advanced GROUP BY

```
SELECT date_format(start_date, 'EEEE') as day, COUNT(*) as ct
FROM student
GROUP BY date_format(start_date, 'EEEE')
```

Example:
Transformation

id	dob	start_date	day
S103	1990-06-20	2001-07-02	Monday
S102	1992-03-26	2010-06-30	Wednesday
S104	1995-03-26	2011-08-03	Wednesday



Monday	S103	1990-06-20	2001-07-02
--------	------	------------	------------



Wednesday	S102	1992-03-26	2010-06-30
	S104	1995-03-26	2011-08-03

day	ct
Monday	1
Wednesday	2

Advanced HAVING

OK, so that was SELECT.

And WHERE.
And GROUP BY.

What about HAVING?

Recall:

→ HAVING simply filters the rows
after the GROUP BY has run

WHERE & HAVING therefore both:

→ decide whether to *keep* the
occurrence entity given the facts for
that entity occurrence (row)

WHERE:

entity occurrence = rows in original tables

HAVING

entity occurrence = rows created as a
result of a GROUP BY part of a query

Advanced HAVING

OK, so that was SELECT.
And WHERE.
And GROUP BY.

What about HAVING?

Recall:

→ HAVING simply filters the rows
after the GROUP BY has run

WHERE & HAVING therefore both:

→ decide whether to *keep* the
occurrence entity given the facts for
that entity occurrence (row)

WHERE:

entity occurrence = rows in original tables

HAVING

entity occurrence = rows created as a
result of a GROUP BY part of a query

As with everything else, it
can also be a mix of these.

A HAVING condition can therefore
include:

→ existing value
(in the table after the GROUP BY is applied),

→ combination or

→ transformations

of that entity occurrences (row's)
values/facts.

REMEMBER: the row are the rows
in the table formed AFTER the
group by!!!!!!

Advanced HAVING

```
SELECT date_format(start_date, 'EEEE') as day, COUNT(*)  
FROM student  
GROUP BY date_format(start_date, 'EEEE')  
HAVING COUNT(*) + 10 = 11
```

student

Example:
Combination
(Transformation)

	id	dob	start_date	day
	S103	1990-06-20	2001-07-02	Monday
	S102	1992-03-26	2010-06-30	Wednesday
	S104	1995-03-26	2011-08-03	Wednesday



	id	dob	start_date
	S103	1990-06-20	2001-07-02

	id	dob	start_date
	S102	1992-03-26	2010-06-30
	S104	1995-03-26	2011-08-03

day
Monday

day
Monday 1
Wednesday 2



NLAB:

Data at Scale

Dr Georgiana Nica-Avram

Summary

Advanced: SELECT,
WHERE, GROUP BY, HAVING

→ All require the specification of per entity occurrence measures to use

This *may* be existing attributes.

Or they may be new ones that are:

- combinations, or
- transformations



Date & Time:

Transformations

Date and time data is one of the most transformed types of data.

Why?

- standard arithmetic does not work
- we often like to format dates and time differently
- all analytics in businesses involves dates and times



Date & Time:

Transformations

How do I find out what functions I can use for problem X?

Answer 1:

- Google to find the documentation
- Google to find a tutorial / hints in solutions on stackoverflow

→ DO NOT try and copy paste a solution to a different problem and change things "until it works"

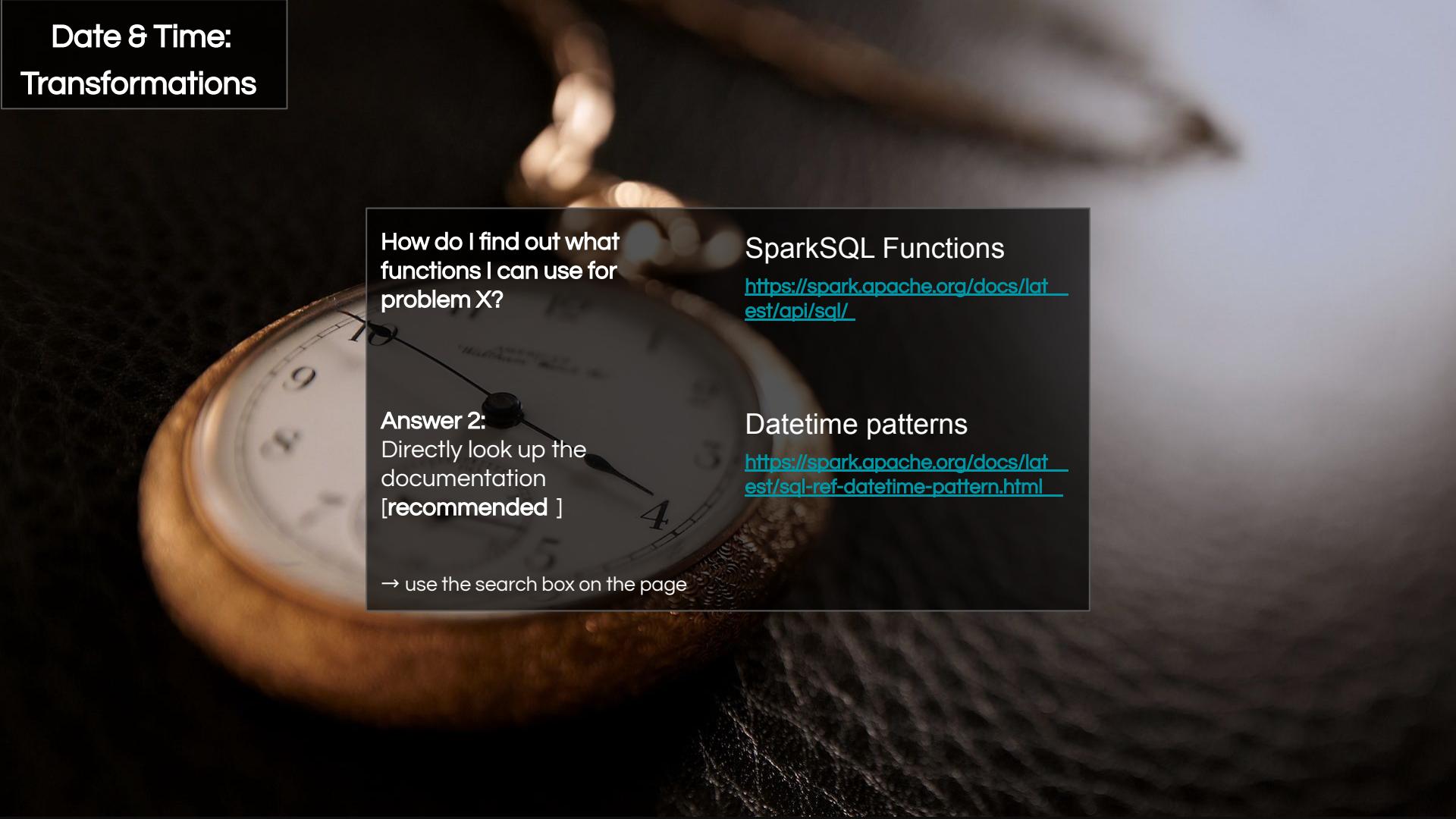
→ You will not learn, be uncertain if you have done it right. Likely a poor solution.

→ Use it as a hint to look up the documentation for functions (transformations) you don't understand.

→ Once you understand the functions you can CORRECTLY write your own

Date & Time:

Transformations



How do I find out what functions I can use for problem X?

Answer 2:

Directly look up the documentation
[recommended]

→ use the search box on the page

SparkSQL Functions

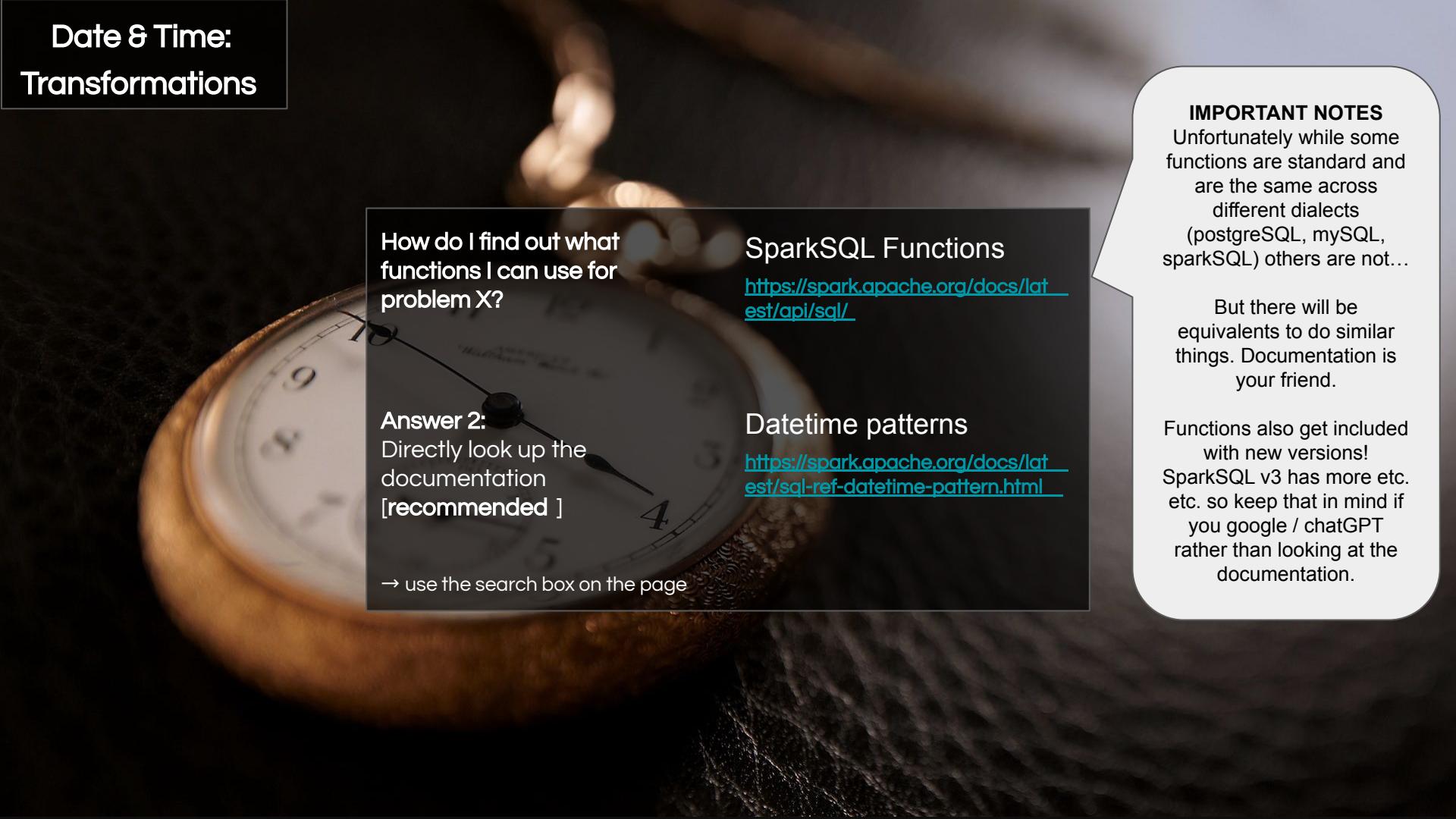
<https://spark.apache.org/docs/latest/api/sql/>

Datetime patterns

<https://spark.apache.org/docs/latest/sql-ref-datetime-pattern.html>

Date & Time:

Transformations



How do I find out what functions I can use for problem X?

Answer 2:

Directly look up the documentation
[recommended]

→ use the search box on the page

SparkSQL Functions

<https://spark.apache.org/docs/latest/api/sql/>

Datetime patterns

<https://spark.apache.org/docs/latest/sql-ref-datetime-pattern.html>

IMPORTANT NOTES

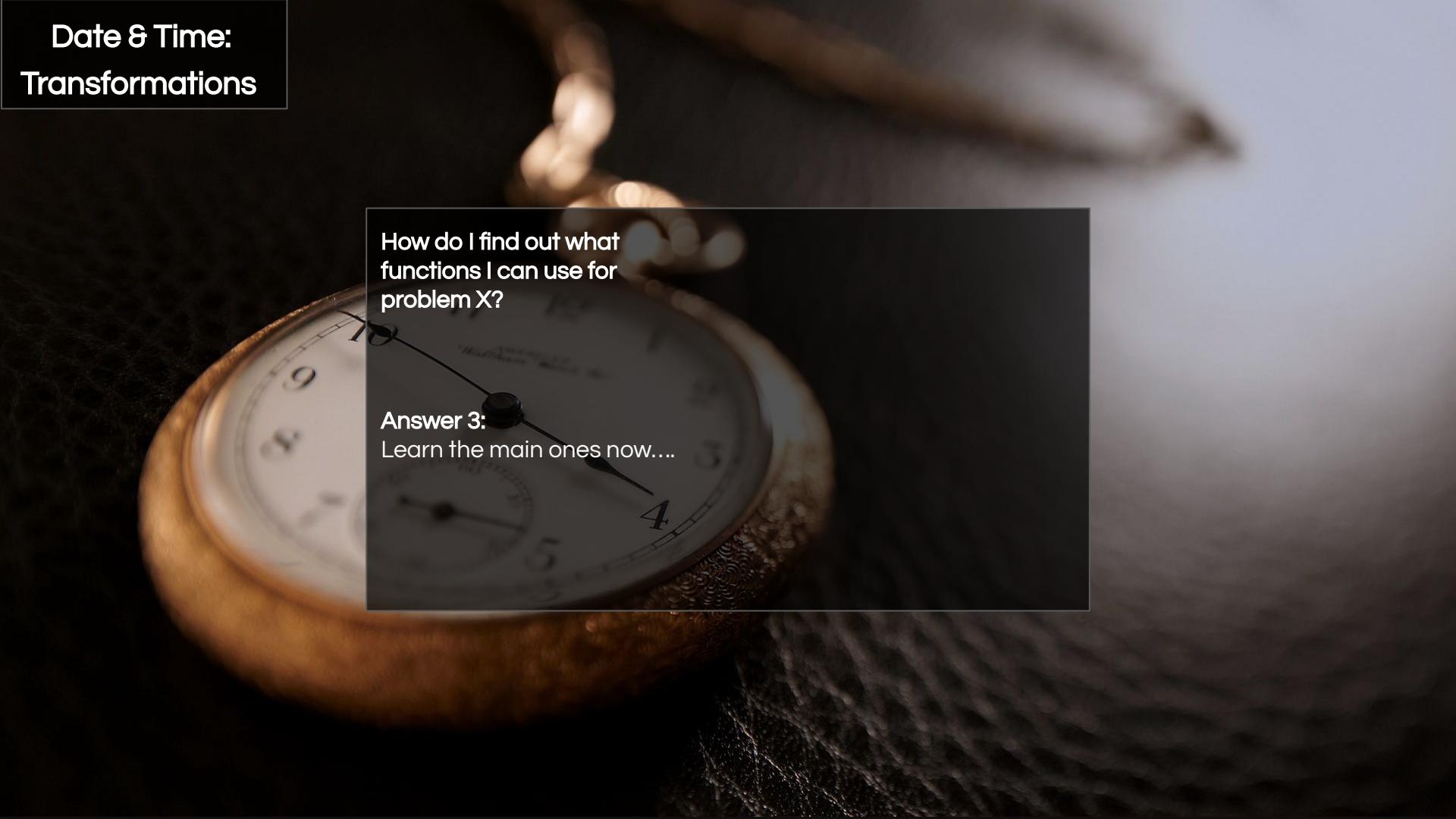
Unfortunately while some functions are standard and are the same across different dialects (postgreSQL, mySQL, sparkSQL) others are not...

But there will be equivalents to do similar things. Documentation is your friend.

Functions also get included with new versions! SparkSQL v3 has more etc. etc. so keep that in mind if you google / chatGPT rather than looking at the documentation.

Date & Time:

Transformations



How do I find out what functions I can use for problem X?

Answer 3:
Learn the main ones now....

Date & Time:

Transformations

The core data types we have are:
TIMESTAMP
DATE

The main date/time functions

DATE_FORMAT(...)
DATE_DIFF(...)
DATE_ADD(...)
DATE_SUB(...)
ADD_MONTHS(...)
MONTHS_BETWEEN(...)
CURRENT_TIMESTAMP()
EXTRACT(...)
DATE_TRUNC(...)

We already know
this one!

Difference (via functions)

What it achieves:

Computes the difference between two dates/times.

Function	Description	Example	Result
<code>DATE_DIFF(endDate, startDate)</code>	Compute the difference between TIMESTAMP or DATE objects. Returns the number of days .	<pre>SELECT DATE_DIFF('2007-06-06 10:00:00'::TIMESTAMP, '1706-03-30 09:00:00'::TIMESTAMP);</pre>	11006 (type = INTEGER)
<code>MONTHS_BETWEEN(endDate,startDate)</code>	Compute the difference between TIMESTAMP or DATE objects. Returns the number of months .	<pre>months_between('2007-03-02'::DATE, '2007-01-01'::DATE));</pre>	2.03225806 (type = DOUBLE)
<code>DATE_SUB(startDate, numDays)</code>	Subtract a given number of days from a DATE. Returns a DATE WARNING: Using a TIMESTAMP here will currently cause a silent error!	<pre>SELECT date_sub('2016-07-30'::DATE, 1); SELECT date_sub('2016-07-30 23:00:00'::TIMESTAMP, 1);</pre>	2016-07-29 (type = DATE) NULL (silent error - do not use!!)

Difference (via functions)

What it achieves:

Computes the difference between two dates/times.

Function	Description	Example	Result
<code>DATE_DIFF(endDate, startDate)</code>	Compute the difference between TIMESTAMP or DATE objects. Returns the number of days .	<pre>SELECT DATE_DIFF('2007-06-06 10:00:00'::TIMESTAMP, '1706-03-30 09:00:00'::TIMESTAMP);</pre>	11006 (type = INTEGER)
<code>MONTHS_BETWEEN(endDate, startDate)</code>	Compute the difference between TIMESTAMP or DATE objects. Returns the number of months .	<pre>months_between('2007-03-02'::DATE, '2007-01-01'::DATE));</pre>	2.03225806 (type = DOUBLE)
<code>DATE_SUB(startDate, numDays)</code>	Subtract a given number of days from a DATE. Returns a DATE WARNING: Using a TIMESTAMP here will currently cause a silent error!	<pre>SELECT date_sub('2016-07-30'::DATE, 1); SELECT date_sub('2016-07-30 23:00:00'::TIMESTAMP, 1);</pre>	2016-07-29 (type = DATE) NULL (silent error - do not use!!)

List the age (in years) of all customers

Real world use:

```
SELECT cust_id, (MONTHS_BETWEEN(CURRENT_TIMESTAMP(), dob) /12)::INT as age
FROM customer
```

Casting to INT forces a round down (simply removes the decimal component) - which is what we want in this case.

Difference (via subtract operator)

What it achieves:

Computes the difference between two dates/times.

Function	Description	Example	Result
<code>TIMESTAMP - TIMESTAMP</code>	<p>Compute the difference between TIMESTAMP. Returns an "INTERVAL DAY TO SECONDS". Cast to BIGINT to get seconds between the timestamps.</p> <p>or DATE objects. Returns an INTERVAL DAY. This is a special type</p>	<pre>SELECT '2007-06-06 10:00:00'::TIMESTAMP - '1706-03-30 09:00:00'::TIMESTAMP; SELECT ('2007-06-06 10:00:00'::TIMESTAMP - '1706-03-30 09:00:00'::TIMESTAMP)::BIGINT</pre>	Unreadable Object (type = INTERVAL DAY TO SECONDS) 9504522000 (type = BIGINT)
<code>DATE - DATE</code>	<p>Compute the difference between DATE. Returns an "INTERVAL DAY". Cast to INTEGER to get days between the timestamps.</p>	<pre>SELECT '2007-06-06 10:00:00'::DATE - '1706-03-30 09:00:00'::DATE; SELECT ('2007-06-06 10:00:00'::DATE - '1706-03-30 09:00:00'::DATE)::INTEGER</pre>	Unreadable Object (type = INTERVAL DAY TO SECONDS) 11006 (type = INTEGER)
<code>TIMESTAMP - INTERVAL</code> <code>DATE - INTERVAL</code> NOTE: BE CAREFUL OF THE RETURN TYPE!	<p>Subtract an interval of time from either a TIMESTAMP or DATE. Returns a DATE (if originally DATE and subtracting days, months or years) or TIMESTAMP otherwise.</p>	<pre>SELECT '2016-07-30 10:00:00'::TIMESTAMP - INTERVAL 30 MINUTES SELECT '2016-07-30'::DATE - INTERVAL 30 MINUTES SELECT '2016-07-30'::DATE - INTERVAL 2 DAYS</pre>	2016-07-30 09:30:00 (type = TIMESTAMP) 2016-07-29 23:30:00 (type = TIMESTAMP) 2016-07-28 (type = DATE)

Addition

What it achieves:

Adds an INTERVAL to a
DATE/TIME/INTERVAL/TIMESTAM
P

Function	Description	Example	Result
<code>TIMESTAMP + INTERVAL</code>	Adds an INTERVAL of time to a TIMESTAMP or DATE	<pre>SELECT '2007-06-06 10:00:00'::TIMESTAMP + INTERVAL 10 days;</pre>	2007-06-16 10:00:00 (type = TIMESTAMP)
<code>DATE + INTERVAL</code>	RETURNS a DATE if starting with DATE and adding days, months, years TIMESTAMP otherwise	<pre>SELECT '2007-06-06'::TIMESTAMP + INTERVAL 10 days;</pre>	2007-06-16 (type = DATE)
<code>DATE_ADD(startDate, numDays)</code>	Add a given number of days from a DATE or TIMESTAMP. Returns a DATE (TIMESTAMP will be truncated!) NOTE: Try not to use a TIMESTAMP, cast to DATE to show you know what is happening to those who read your code in the future!	<pre>SELECT date_add('2016-07-29'::DATE, 1); SELECT date_add('2016-07-29 23:00:00'::TIMESTAMP, 1);</pre>	2016-07-30 (type = DATE) 2016-07-30 (type = DATE)

CURRENT_TIMESTAMP()

What it achieves:

Gets the current time and date.

Want just the date? Cast!

CURRENT_TIMESTAMP()::DATE =
2018-10-19

Function	Description	Example	Result
<code>CURRENT_TIMESTAMP()</code>	Returns the current date and time. RETURNS a TIMESTAMP	<pre>SELECT CURRENT_TIMESTAMP();</pre>	2023-10-21T15:11:46.071+0000 (type = TIMESTAMP)

EXTRACT

What it achieves:

Extracts and returns PART of a DATE or TIMESTAMP

Function	Description	Example	Result
<code>EXTRACT(field FROM timestamp)</code>	Get subfield. Options: year, yearofweek, quarter, month, week, day, dayofweek, dayofweek_iso, dow, hour, minute, second See: https://spark.apache.org/docs/latest/api/sql/#extract	<code>extract(month from '2007-05-01'::TIMESTAMP)</code>	5
		<code>extract(dow from '2007-05-01'::DATE)</code>	3

DATE_TRUNC

Values of type DATE are cast automatically to TIMESTAMP

What it achieves:

Truncates a DATE or TIMESTAMP to a given precision by setting lower precision elements to zero (or one for day and month).

Function	Description	Example	Result
<code>DATE_TRUNC(field, timestamp)</code>	Sets elements from source that are less than the precision specified in field to zero (or one for day & month). Options for field: microseconds, milliseconds, second, minute, hour, day, week, month, quarter, year See: https://spark.apache.org/docs/latest/api/sql/#date_trunc	<pre>SELECT DATE_TRUNC('month', '2007-05-15'::DATE)</pre>	2007-05-01 00:00:00
		<pre>SELECT DATE_TRUNC('hour', '2007-05-01 09:30:05'::TIMESTAMP)</pre>	2007-05-01 09:00:00

DATE_TRUNC

Values of type DATE are cast automatically to TIMESTAMP

What it achieves:

Truncates a DATE or TIMESTAMP to a given precision by setting lower precision elements to zero (or one for day and month).

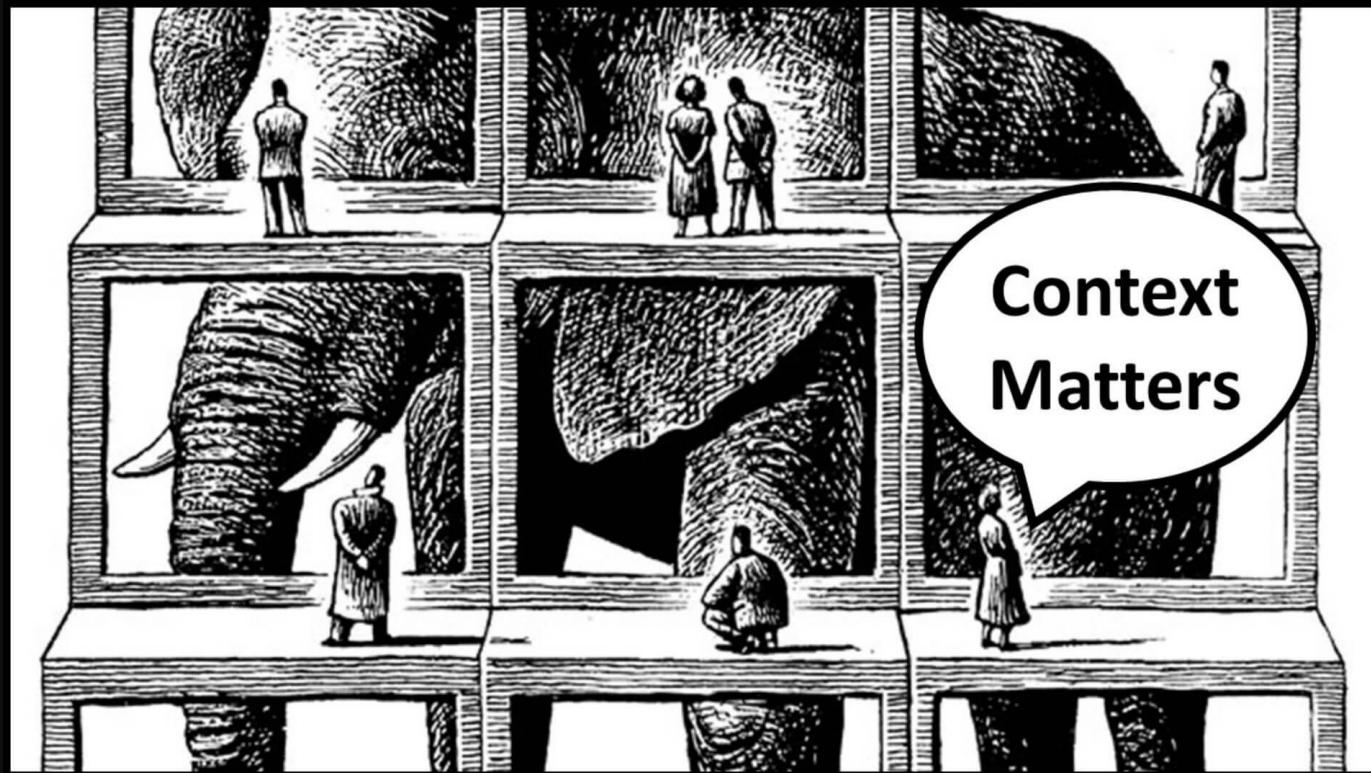
Real world use:

```
SELECT DATE_TRUNC('month', receipt_timestamp) as year_month,  
       SUM(price) as total_sales  
  FROM line_item  
 GROUP BY DATE_TRUNC('month', receipt_timestamp)
```

Function	Description	Example	Result
<code>DATE_TRUNC(field, timestamp)</code>	Sets elements from source that are less than the precision specified in field to zero (or one for day & month). Options for field: microseconds, milliseconds, second, minute, hour, day, week, month, quarter, year See: https://spark.apache.org/docs/latest/api/sql/#date_trunc	<pre>SELECT DATE_TRUNC('month', '2007-05-15'::DATE)</pre>	2007-05-01 00:00:00
		<pre>SELECT DATE_TRUNC('hour', '2007-05-01 09:30:05'::TIMESTAMP)</pre>	2007-05-01 09:00:00

Return the total sales per month.

*Let's put this all in
practice...*



Data at Scale

Dr Georgiana Nica-Avram

*Let's put this all in
practice...*



Let's put this all in practice...

Image you've just been employed as a data analyst by UNREALISTIC CORP. The company is a startup which sells coffee products by delivery over the Internet.

UNREALISTIC CORP. has employed you to help develop their customer loyalty program. The team developing the program has asked for answers to the following questions:

- 1) *List the customers by name who shop with us more than once a month?*
- 2) *How many customers shop with us more than once a month?*

Let's put this all in practice...

- 1) *List the customers by name who shop with us more than once a month?*

- 2) *How many customers shop with us more than once a month?*

To get you started UNREALISTIC CORP. provides you with a full set of documentation about their database.

They also introduce you to their data quality team which has infinite resources who can 100% guarantee there are no errors in the database.

The documentation

customers

Field Name	Data Type	Constraint	Default	Description	Example
customer_id	INTEGER	Primary Key	auto-incremet	Unique identifier generated for each customer	8734
name	STRING	NOT NULL		The name of the customer	Gavin Smith

orders

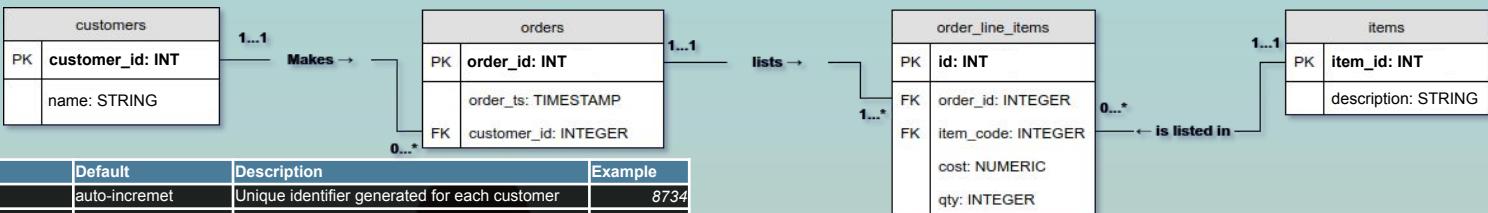
Field Name	Data Type	Constraint	Default	Description	Example
order_id	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	123456
order_ts	TIMESTAMP	NOT NULL	current_timestamp()	Timestamp of when the order was placed	2017-01-01 13:01:02
customer_id	INTEGER	NOT NULL		The id of the customer that made the order.	8734

items

Field Name	Data Type	Constraint	Default	Description	Example
item_id	INTEGER	Primary Key	auto-increment	Unique identifier generated for each item	545
description	STRING	NOT NULL		The description of the item	Mellowship Slinky Decaf

order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
id	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	9654000
order_id	INTEGER	Foreign Key (orders.order_id), NOT NULL		The order_id for the order which this line item belongs.	123456
item_code	INTEGER	Foreign Key (items.item_id), NOT NULL		The id for the item bought for this line item.	545
cost	NUMERIC	NOT NULL, CHECK(cost > 0)		The total cost (unit cost x qty) of the line item.	16.50
qty	INTEGER	CHECK(qty > 0)		The number of items brought for this line item.	1



items	
PK	item_id: INT
	description: STRING

**Let's put this all in
practice...**

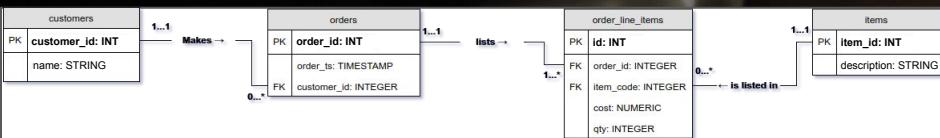
- 1) *List the customers by name who shop with us more than once a month while they were active customers?*

- 2) *How many customers shop with us more than once a month?*

Since we can guarantee there are no errors in the data, we can get started on the queries!

order line item

Field Name	Data Type	Constraint	Default	Description	Example
id	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	9654000
order_id	INTEGER	Foreign Key (orders.order_id), NOT NULL		The order_id for the order which this line item belongs.	123456
item_code	INTEGER	Foreign Key (items.item_id), NOT NULL		The id for the item bought for this line item.	545
cost	NUMERIC	NOT NULL, CHECK(cost > 0)		The total cost (unit cost x qty) of the line item.	16.50
qty	INTEGER	CHECK(qty > 0)		The number of items brought for this line item.	1



items

Field Name	Data Type	Constraint	Default	Description	Example
item_id	SERIAL	Primary Key	auto-increment	Unique identifier generated for each item	542
description	STRING	NOT NULL		The description of the item	Mellowip Slinky Deca

customers

Field Name	Data Type	Constraint	Default	Description	Example
customer_id	INTEGER	Primary Key	auto-increment	Unique identifier generated for each customer	8734
name	STRING	NOT NULL		The name of the customer	Gavin Smith

orders

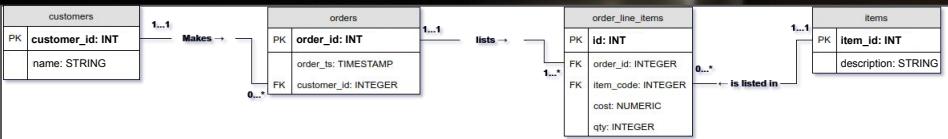
Field Name	Data Type	Constraint	Default	Description	Example
order_id	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	123456
order_ts	TIMESTAMP	NOT NULL	now()	Timestamp of when the order was placed	2017-01-01 13:01:01
customer_id	INTEGER	NOT NULL		The id of the customer that made the order.	8734

QUESTION

List the customers by name who shop with us more than once a month while they were active customers?

order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
<code>id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	9654000
<code>order_id</code>	INTEGER	Foreign Key (orders.order_id), NOT NULL		The order_id for the order which this line item belongs.	123456
<code>item_code</code>	INTEGER	Foreign Key (items.item_id), NOT NULL		The id for the item bought for this line item.	545
<code>cost</code>	NUMERIC	NOT NULL, CHECK(cost > 0)		The total cost (unit cost x qty) of the line item.	16.50
<code>qty</code>	INTEGER	CHECK(qty > 0)		The number of items brought for this line item.	1



items

Field Name	Data Type	Constraint	Default	Description	Example
<code>item_id</code>	SERIAL	Primary Key	auto-increment	Unique identifier generated for each item	545
<code>description</code>	STRING	NOT NULL		The description of the item	Mellowship Slinky Decaf

customers

Field Name	Data Type	Constraint	Default	Description	Example
<code>customer_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each customer	8734
<code>name</code>	STRING	NOT NULL		The name of the customer	Gavin Smith

orders

Field Name	Data Type	Constraint	Default	Description	Example
<code>order_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	123456
<code>order_ts</code>	TIMESTAMP	NOT NULL	now()	Timestamp of when the order was placed	2017-01-01 13:01:02
<code>customer_id</code>	INTEGER	NOT NULL		The id of the customer that made the order.	8734

QUESTION

List the customers by name who shop with us more than once a month while they were active customers?

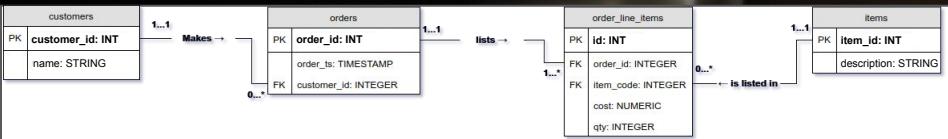
How to do this?

I'd work how to do it in parts.



order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
<code>id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	9654000
<code>order_id</code>	INTEGER	Foreign Key (orders.order_id), NOT NULL		The order_id for the order which this line item belongs.	123456
<code>item_code</code>	INTEGER	Foreign Key (items.item_id), NOT NULL		The id for the item bought for this line item.	545
<code>cost</code>	NUMERIC	NOT NULL, CHECK(cost > 0)		The total cost (unit cost x qty) of the line item.	16.50
<code>qty</code>	INTEGER	CHECK(qty > 0)		The number of items brought for this line item.	1



items

Field Name	Data Type	Constraint	Default	Description	Example
<code>item_id</code>	SERIAL	Primary Key	auto-increment	Unique identifier generated for each item	545
<code>description</code>	STRING	NOT NULL		The description of the item	Mellowship Slinky Decaf

customers

Field Name	Data Type	Constraint	Default	Description	Example
<code>customer_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each customer	8734
<code>name</code>	STRING	NOT NULL		The name of the customer	Gavin Smith

orders

Field Name	Data Type	Constraint	Default	Description	Example
<code>order_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	123456
<code>order_ts</code>	TIMESTAMP	NOT NULL	now()	Timestamp of when the order was placed	2017-01-01 13:01:02
<code>customer_id</code>	INTEGER	NOT NULL		The id of the customer that made the order.	8734

QUESTION

List the customers by name who **shop with us more than once a month** while they were active customers?

How to do this?

I'd work how to do it in parts.

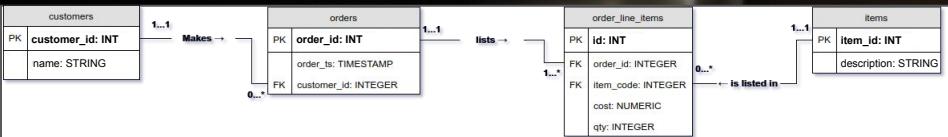
SELECT

FROM **customers**

WHERE

order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
<code>id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	9654000
<code>order_id</code>	INTEGER	Foreign Key (orders.order_id), NOT NULL		The order_id for the order which this line item belongs.	123456
<code>item_code</code>	INTEGER	Foreign Key (items.item_id), NOT NULL		The id for the item bought for this line item.	545
<code>cost</code>	NUMERIC	NOT NULL, CHECK(cost > 0)		The total cost (unit cost x qty) of the line item.	16.50
<code>qty</code>	INTEGER	CHECK(qty > 0)		The number of items brought for this line item.	1



items

Field Name	Data Type	Constraint	Default	Description	Example
<code>item_id</code>	SERIAL	Primary Key	auto-increment	Unique identifier generated for each item	545
<code>description</code>	STRING	NOT NULL		The description of the item	Mellowship Slinky Decaf

customers

Field Name	Data Type	Constraint	Default	Description	Example
<code>customer_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each customer	8734
<code>name</code>	STRING	NOT NULL		The name of the customer	Gavin Smith

orders

Field Name	Data Type	Constraint	Default	Description	Example
<code>order_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	123456
<code>order_ts</code>	TIMESTAMP	NOT NULL	now()	Timestamp of when the order was placed	2017-01-01 13:01:02
<code>customer_id</code>	INTEGER	NOT NULL		The id of the customer that made the order.	8734

QUESTION

List the customers by name who **shop with us more than once a month** while they were active customers?

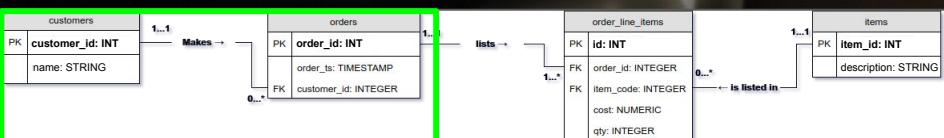
Ah! So we have a join to deal with!

order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
<code>id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	9654000
<code>order_id</code>	INTEGER	Foreign Key (orders.order_id), NOT NULL		The order_id for the order which this line item belongs.	123456
<code>item_code</code>	INTEGER	Foreign Key (items.item_id), NOT NULL		The id for the item bought for this line item.	545
<code>cost</code>	NUMERIC	NOT NULL, CHECK(cost > 0)		The total cost (unit cost x qty) of the line item.	16.50
<code>qty</code>	INTEGER	CHECK(qty > 0)		The number of items brought for this line item.	1

QUESTION

List the customers by name who shop with us more than once a month while they were active customers?



items

Field Name	Data Type	Constraint	Default	Description	Example
<code>item_id</code>	SERIAL	Primary Key	auto-increment	Unique identifier generated for each item	545
<code>description</code>	STRING	NOT NULL		The description of the item	Mellowship Slinky Decaf

customers

Field Name	Data Type	Constraint	Default	Description	Example
<code>customer_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each customer	8734
<code>name</code>	STRING	NOT NULL		The name of the customer	Gavin Smith

orders

Field Name	Data Type	Constraint	Default	Description	Example
<code>order_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	123456
<code>order_ts</code>	TIMESTAMP	NOT NULL	now()	Timestamp of when the order was placed	2017-01-01 13:01:02
<code>customer_id</code>	INTEGER	NOT NULL		The id of the customer that made the order.	8734

We have a one-to-many join.

This is the **most common** by far.

SELECT

```

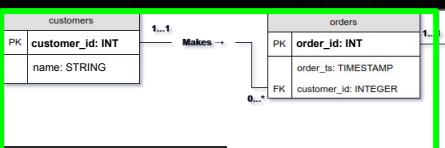
  FROM orders
  JOIN customers
  ON orders.customer_id =
  customers.customer_id
  
```

order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
<code>id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	9654000
<code>order_id</code>	INTEGER	Foreign Key (orders.order_id), NOT NULL		The order_id for the order which this line item belongs.	123456
<code>item_code</code>	INTEGER	Foreign Key (items.item_id), NOT NULL		The id for the item bought for this line item.	545
<code>cost</code>	NUMERIC	NOT NULL, CHECK(cost > 0)		The total cost (unit cost x qty) of the line item.	16.50
<code>qty</code>	INTEGER	CHECK(qty > 0)		The number of items brought for this line item.	1

QUESTION

List the customers by name who shop with us more than once a month while they were active customers?



items

Field Name	Data Type	Constraint	Default	Description	Example
<code>item_id</code>	SERIAL	Primary Key	auto-increment	Unique identifier generated for each item	545
<code>description</code>	STRING	NOT NULL		The description of the item	Mellowship Slinky Decaf

customers

Field Name	Data Type	Constraint	Default	Description	Example
<code>customer_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each customer	8734
<code>name</code>	STRING	NOT NULL		The name of the customer	Gavin Smith

orders

Field Name	Data Type	Constraint	Default	Description	Example
<code>order_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	123456
<code>order_ts</code>	TIMESTAMP	NOT NULL	now()	Timestamp of when the order was placed	2017-01-01 13:01:02
<code>customer_id</code>	INTEGER	NOT NULL		The id of the customer that made the order.	8734

We have a one-to-many join.

This is the **most common** by far.

One interpretation is:

Each **row in orders** gets **extended** with (the corresponding information from the **customer** table) **name** (each row in orders matches exactly one row in customer).

SELECT

FROM **orders**

JOIN **customers**

```

ON orders.customer_id =
customers.customer_id
  
```

So my logic:

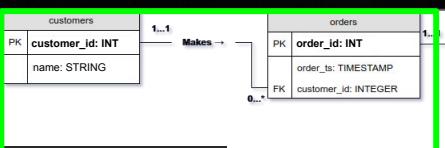
- 1) Each row: order + customer name
- 2) For each customer compute: number of months visited
- 3) For each customer compute: number of months active
- 4) Then filter: (2) / (3) > 1

order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
<code>id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	9654000
<code>order_id</code>	INTEGER	Foreign Key (orders.order_id), NOT NULL		The order_id for the order which this line item belongs.	123456
<code>item_code</code>	INTEGER	Foreign Key (items.item_id), NOT NULL		The id for the item bought for this line item.	545
<code>cost</code>	NUMERIC	NOT NULL, CHECK(cost > 0)		The total cost (unit cost x qty) of the line item.	16.50
<code>qty</code>	INTEGER	CHECK(qty > 0)		The number of items brought for this line item.	1

QUESTION

List the customers by name who shop with us more than once a month while they were active customers?



items

Field Name	Data Type	Constraint	Default	Description	Example
<code>item_id</code>	SERIAL	Primary Key	auto-increment	Unique identifier generated for each item	545
<code>description</code>	STRING	NOT NULL		The description of the item	Mellowship Slinky Decaf

customers

Field Name	Data Type	Constraint	Default	Description	Example
<code>customer_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each customer	8734
<code>name</code>	STRING	NOT NULL		The name of the customer	Gavin Smith

orders

Field Name	Data Type	Constraint	Default	Description	Example
<code>order_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	123456
<code>order_ts</code>	TIMESTAMP	NOT NULL	now()	Timestamp of when the order was placed	2017-01-01 13:01:02
<code>customer_id</code>	INTEGER	NOT NULL		The id of the customer that made the order.	8734

We have a one-to-many join.

This is the **most common** by far.

One interpretation is:

Each **row in orders** gets **extended** with (the corresponding information from the **customer** table) **name** (each row in orders matches exactly one row in customer).

SELECT

FROM orders

JOIN customers

ON orders.customer_id =
customers.customer_id

So my logic:

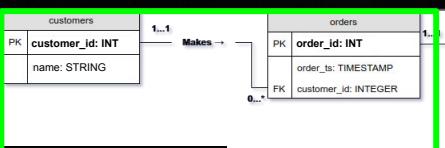
- 1) Each row: order + customer name
- 2) For each customer compute: number of months visited
- 3) For each customer compute: number of months active
- 4) Then filter: (2) / (3) > 1

order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
<code>id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	9654000
<code>order_id</code>	INTEGER	Foreign Key (orders.order_id), NOT NULL		The order_id for the order which this line item belongs.	123456
<code>item_code</code>	INTEGER	Foreign Key (items.item_id), NOT NULL		The id for the item bought for this line item.	545
<code>cost</code>	NUMERIC	NOT NULL, CHECK(cost > 0)		The total cost (unit cost x qty) of the line item.	16.50
<code>qty</code>	INTEGER	CHECK(qty > 0)		The number of items brought for this line item.	1

QUESTION

List the customers by name who shop with us more than once a month while they were active customers?



items

Field Name	Data Type	Constraint	Default	Description	Example
<code>item_id</code>	SERIAL	Primary Key	auto-increment	Unique identifier generated for each item	545
<code>description</code>	STRING	NOT NULL		The description of the item	Mellowship Slinky Decaf

customers

Field Name	Data Type	Constraint	Default	Description	Example
<code>customer_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each customer	8734
<code>name</code>	STRING	NOT NULL		The name of the customer	Gavin Smith

orders

Field Name	Data Type	Constraint	Default	Description	Example
<code>order_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	123456
<code>order_ts</code>	TIMESTAMP	NOT NULL	now()	Timestamp of when the order was placed	2017-01-01 13:01:02
<code>customer_id</code>	INTEGER	NOT NULL		The id of the customer that made the order.	8734

We have a one-to-many join.

This is the **most common** by far.

One interpretation is:

Each **row in orders** gets **extended** with (the corresponding information from the **customer** table) **name** (each row in orders matches exactly one row in customer).

SELECT

```

    FROM orders
    JOIN customers
    ON orders.customer_id =
        customers.customer_id
  
```

So my logic:

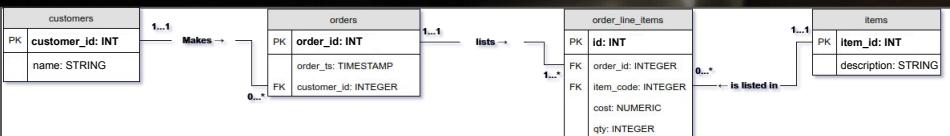
- 1) Each row: order + customer name
- 2) For each customer compute: number of months visited
- 3) For each customer compute: number of months active
- 4) Then filter: (2) / (3) > 1

order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
<code>id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	9654000
<code>order_id</code>	INTEGER	Foreign Key (orders.order_id), NOT NULL		The order_id for the order which this line item belongs.	123456
<code>item_code</code>	INTEGER	Foreign Key (items.item_id), NOT NULL		The id for the item bought for this line item.	545
<code>cost</code>	NUMERIC	NOT NULL, CHECK(cost > 0)		The total cost (unit cost x qty) of the line item.	16.50
<code>qty</code>	INTEGER	CHECK(qty > 0)		The number of items brought for this line item.	1

QUESTION

List the customers by name who shop with us more than once a month while they were active customers?



items

Field Name	Data Type	Constraint	Default	Description	Example
<code>item_id</code>	SERIAL	Primary Key	auto-increment	Unique identifier generated for each item	545
<code>description</code>	STRING	NOT NULL		The description of the item	Mellowship Slinky Decaf

customers

Field Name	Data Type	Constraint	Default	Description	Example
<code>customer_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each customer	8734
<code>name</code>	STRING	NOT NULL		The name of the customer	Gavin Smith

orders

Field Name	Data Type	Constraint	Default	Description	Example
<code>order_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	123456
<code>order_ts</code>	TIMESTAMP	NOT NULL	now()	Timestamp of when the order was placed	2017-01-01 13:01:02
<code>customer_id</code>	INTEGER	NOT NULL		The id of the customer that made the order.	8734

So my logic:

- 1) Each row: order + customer name
- 2) For each customer compute: number of months visited
- 3) For each customer compute: number of months active
- 4) Then filter: (2) / (3) > 1

Let's break this logic down!

```

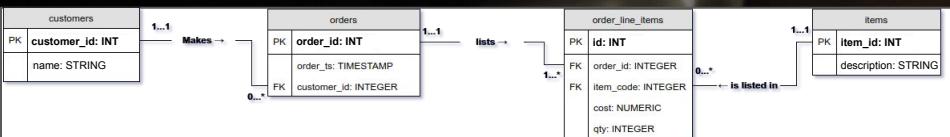
SELECT customer_id, name
FROM orders
JOIN customers
ON orders.customer_id =
customers.customer_id
GROUP BY customer_id, name
    
```

order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
<code>id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	9654000
<code>order_id</code>	INTEGER	Foreign Key (orders.order_id), NOT NULL		The order_id for the order which this line item belongs.	123456
<code>item_code</code>	INTEGER	Foreign Key (items.item_id), NOT NULL		The id for the item bought for this line item.	545
<code>cost</code>	NUMERIC	NOT NULL, CHECK(cost > 0)		The total cost (unit cost x qty) of the line item.	16.50
<code>qty</code>	INTEGER	CHECK(qty > 0)		The number of items brought for this line item.	1

QUESTION

List the customers by name who shop with us more than once a month while they were active customers?



items

Field Name	Data Type	Constraint	Default	Description	Example
<code>item_id</code>	SERIAL	Primary Key	auto-increment	Unique identifier generated for each item	545
<code>description</code>	STRING	NOT NULL		The description of the item	Mellowship Slinky Decaf

customers

Field Name	Data Type	Constraint	Default	Description	Example
<code>customer_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each customer	8734
<code>name</code>	STRING	NOT NULL		The name of the customer	Gavin Smith

orders

Field Name	Data Type	Constraint	Default	Description	Example
<code>order_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	123456
<code>order_ts</code>	TIMESTAMP	NOT NULL	now()	Timestamp of when the order was placed	2017-01-01 13:01:02
<code>customer_id</code>	INTEGER	NOT NULL		The id of the customer that made the order.	8734

So my logic:

- 1) Each row: order + customer name
- 2) For each customer compute: number of months visited
- 3) For each customer compute: number of months active
- 4) Then filter: (2) / (3) > 1

Let's break this logic down!

```

SELECT customer_id, name,
       COUNT(DATE_TRUNC('month', order_ts))
FROM orders
JOIN customers
ON orders.customer_id = customers.customer_id
GROUP BY customer_id, name
    
```

order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
<code>id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	9654000
<code>order_id</code>	INTEGER	Foreign Key (<code>orders.order_id</code>), <code>NOT NULL</code>		The <code>order_id</code> for the order which this line item belongs	123456
<code>item_qty</code>	INTEGER				545
<code>cost_qty</code>	DECIMAL(10,2)				16.50
<code>item_c</code>	CHAR(1)				1

Ok. So this line got a little more complex.

Let's break it down.

- GROUP BY = buckets, one per customer
- Running the SELECT processes row-by-row (so bucket-by-bucket)

QUESTION

List the customers by name who shop with us more than once a month while they were active customers?

So my logic:

- 1) Each row: order + customer name
- 2) For each customer compute:
number of months visited
- 3) For each customer compute:
number of months active
- 4) Then filter: $(2) / (3) > 1$

Let's break this logic down!

```
SELECT customer_id, name,  
       COUNT(DATE_TRUNC('month', order_ts)),  
       CEIL( MONTHS_BETWEEN( MAX(order_ts), MIN(order_ts) )  
FROM orders  
JOIN customers  
ON orders.customer_id = customers.customer_id  
GROUP BY customer_id, name
```

order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
<code>id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	9654000
<code>order_id</code>	INTEGER	Foreign Key (<code>orders.order_id</code>), <code>NOT NULL</code>		The <code>order_id</code> for the order which this line item belongs	123456
<code>item_qty</code>	INTEGER				545
<code>cost_qty</code>	DECIMAL(10,2)				16.50
<code>item_c</code>	CHAR(1)				1

Ok. So this line got a little more complex.

Let's break it down.

- GROUP BY = buckets, one per customer
- Running the SELECT processes row-by-row (so bucket-by-bucket)
- For customer 1:
 - Look in bucket, compute: `MAX(order_ts)`
 - Look in bucket, compute: `MIN(order_ts)`

QUESTION

List the customers by name who shop with us more than once a month while they were active customers?

So my logic:

- 1) Each row: order + customer name
- 2) For each customer compute:
number of months visited
- 3) For each customer compute:
number of months active
- 4) Then filter: $(2) / (3) > 1$

Let's break this logic down!

```
SELECT customer_id, name,  
       COUNT(DATE_TRUNC('month', order_ts)),  
       CEIL( MONTHS_BETWEEN( MAX(order_ts), MIN(order_ts) )  
FROM orders  
JOIN customers  
ON orders.customer_id = customers.customer_id  
GROUP BY customer_id, name
```

order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
<code>id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	9654000
<code>order_id</code>	INTEGER	Foreign Key (<code>orders.order_id</code>), <code>NOT NULL</code>		The <code>order_id</code> for the order which this line item belongs	123456 545 16.50 1

Ok. So this line got a little more complex.

Let's break it down.

- GROUP BY = buckets, one per customer
- Running the SELECT processes row-by-row (so bucket-by-bucket)
- For customer 1:
 - Look in bucket, compute: `MAX(order_ts)`
 - Look in bucket, compute: `MIN(order_ts)`
 - Take these two results (now TIMESTAMPS) and pass it to the `MONTHS_BETWEEN(..)` function to get the number of months between these - the number of months the customer has been active

QUESTION

List the customers by name who shop with us more than once a month while they were active customers?

So my logic:

- 1) Each row: order + customer name
- 2) For each customer compute: number of months visited
- 3) For each customer compute: number of months active
- 4) Then filter: $(2) / (3) > 1$

```
SELECT customer_id, name,  
       COUNT(DATE_TRUNC('month', order_ts)),  
       CEIL( MONTHS_BETWEEN( MAX(order_ts), MIN(order_ts) )  
FROM orders  
JOIN customers  
ON orders.customer_id = customers.customer_id  
GROUP BY customer_id, name
```

Let's break this logic down!

order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
<code>id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	9654000
<code>order_id</code>	INTEGER	Foreign Key (<code>orders.order_id</code>), <code>NOT NULL</code>		The <code>order_id</code> for the order which this line item belongs	123456
<code>item_qty</code>	INTEGER				545
<code>cost_qty</code>	DECIMAL				16.50
<code>customer_id</code>	INTEGER				1

Ok. So this line got a little more complex.

Let's break it down.

- GROUP BY = buckets, one per customer
- Running the SELECT processes row-by-row (so bucket-by-bucket)
- For customer 1:
 - Look in bucket, compute: `MAX(order_ts)`
 - Look in bucket, compute: `MIN(order_ts)`
 - Take these two results (now TIMESTAMPS) and pass it to the `MONTHS_BETWEEN(..)` function to get the number of months between these - the number of months the customer has been active
 - The customer doesn't have to have come in exactly two months apart to be considered shopped in two months, so we round this difference up.
 - This is done by a new function `CEIL(..)`

QUESTION

List the customers by name who shop with us more than once a month while they were active customers?

So my logic:

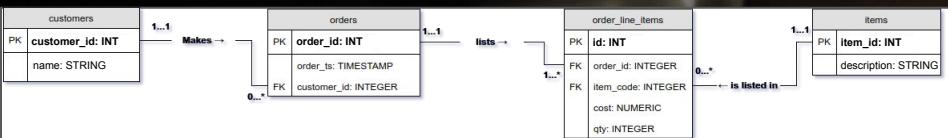
- 1) Each row: order + customer name
- 2) For each customer compute: number of months visited
- 3) For each customer compute: number of months active
- 4) Then filter: $(2) / (3) > 1$

```
SELECT customer_id, name,  
       COUNT(DATE_TRUNC('month', order_ts)),  
       CEIL( MONTHS_BETWEEN( MAX(order_ts), MIN(order_ts) )  
FROM orders  
JOIN customers  
ON orders.customer_id = customers.customer_id  
GROUP BY customer_id, name
```

Let's break this logic down!

order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
<code>id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	9654000
<code>order_id</code>	INTEGER	Foreign Key (orders.order_id), NOT NULL		The order_id for the order which this line item belongs.	123456
<code>item_code</code>	INTEGER	Foreign Key (items.item_id), NOT NULL		The id for the item bought for this line item.	545
<code>cost</code>	NUMERIC	NOT NULL, CHECK(cost > 0)		The total cost (unit cost x qty) of the line item.	16.50
<code>qty</code>	INTEGER	CHECK(qty > 0)		The number of items brought for this line item.	1



items

Field Name	Data Type	Constraint	Default	Description	Example
<code>item_id</code>	SERIAL	Primary Key	auto-increment	Unique identifier generated for each item	545
<code>description</code>	STRING	NOT NULL		The description of the item	Mellowship Slinky Decaf

customers

Field Name	Data Type	Constraint	Default	Description	Example
<code>customer_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each customer	8734
<code>name</code>	STRING	NOT NULL		The name of the customer	Gavin Smith

orders

Field Name	Data Type	Constraint	Default	Description	Example
<code>order_id</code>	INTEGER	Primary Key	auto-increment	Unique identifier generated for each order	123456
<code>order_ts</code>	TIMESTAMP	NOT NULL	now()	Timestamp of when the order was placed	2017-01-01 13:01:02
<code>customer_id</code>	INTEGER	NOT NULL		The id of the customer that made the order.	8734

QUESTION

List the customers by name who shop with us more than once a month while they were active customers?

So my logic:

- 1) Each row: order + customer name
- 2) For each customer compute: number of months visited
- 3) For each customer compute: number of months active
- 4) Then filter: (2) / (3) > 1

```

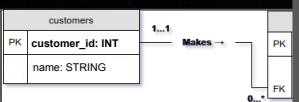
SELECT customer_id, name,
       COUNT(DATE_TRUNC('month', order_ts)) /
       CEIL( MONTHS_BETWEEN( MAX(order_ts), MIN(order_ts) ) )
           AS shops_per_month
FROM orders
JOIN customers
ON orders.customer_id = customers.customer_id
GROUP BY customer_id, name
HAVING COUNT(DATE_TRUNC('month', order_ts)) /
       CEIL( MONTHS_BETWEEN( MAX(order_ts), MIN(order_ts) ) ) > 1
  
```

Let's break this logic down!

order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
id	INTEGER	Primary Key		each order	9654000
order_id	INTEGER	Foreign Key	NOT NULL	this line item	123456
item_code	INTEGER	Foreign Key	NOT NULL	item.	545
cost	NUMERIC	NOT NULL		line item.	16.50
qty	INTEGER	CHECK(line item.	1

And we're done!



Field Name	Data Type	Constraint
item_id	SERIAL	Primary Key
description	STRING	NOT NULL

customers

```
SELECT 1/x, avg(y)
FROM tab
GROUP BY x
HAVING NOT x = 0;
```

orders

Field Name	Data Type	Constraint
order_id	INTEGER	Primary Key
order_ts	TIMESTAMP	NOT NULL
customer_id	INTEGER	NOT NULL

Side note:

Why can't I just refer to **shops_per_month** in the HAVING?

As the HAVING part is computed first.

Why? Consider:

```
SELECT 1/x, avg(y)
FROM tab
GROUP BY x
HAVING NOT x = 0;
```

Computing the SELECT fields before the HAVING would result in a divide by zero error.

each order	9654000
this line item	123456
item.	545
line item.	16.50
line item.	1

is	INT
name: STRING	

sample	545
Mellowship	
inky Decaf	

sample	8734
avin Smith	

sample	8734
avin Smith	

sample	123456
017-01-01	
13:01:02	

sample	8734
avin Smith	

QUESTION

List the customers by name who shop with us more than once a month while they were active customers?

Let's break this logic down!

So my logic:

- 1) Each row: order + customer name
- 2) For each customer compute: number of months visited
- 3) For each customer compute: number of months active
- 4) Then filter: (2) / (3) > 1

```
SELECT customer_id, name,
       COUNT(DATE_TRUNC('month', order_ts)) /
       CEIL( MONTHS_BETWEEN( MAX(order_ts), MIN(order_ts) ) )
            AS shops_per_month
  FROM orders
  JOIN customers
    ON orders.customer_id = customers.customer_id
 GROUP BY customer_id, name
 HAVING COUNT(DATE_TRUNC('month', order_ts)) /
       CEIL( MONTHS_BETWEEN( MAX(order_ts), MIN(order_ts) ) ) > 1
```

One down, one to go!

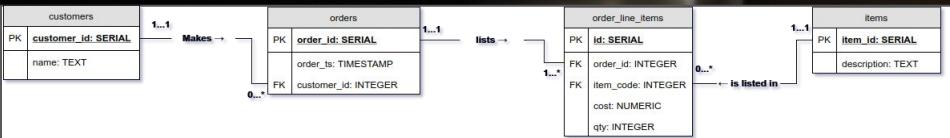
- 1) *List the customers by name who shop with us more than once a month while they were active customers?*

- 2) *How many customers shop with us more than once a month?*

```
SELECT customer_id, name
FROM orders
JOIN customers
ON orders.customer_id = customers.customer_id
GROUP BY customer_id, name
HAVING COUNT( DATE_TRUNC('month', order_ts)) /
CEIL( MONTHS_BETWEEN( MAX(order_ts), MIN(order_ts) ) ) > 1
```

order_line_item

Field Name	Data Type	Constraint	Default	Description	Example
<code>id</code>	SERIAL	Primary Key	auto-increment	Unique identifier generated for each order	9654000
<code>order_id</code>	INTEGER	Foreign Key (orders.order_id), NOT NULL		The order_id for the order which this line item belongs.	123456
<code>item_code</code>	INTEGER	Foreign Key (items.item_id), NOT NULL		The id for the item bought for this line item.	545
<code>cost</code>	NUMERIC	NOT NULL, CHECK(cost > 0)		The total cost (unit cost x qty) of the line item.	16.50
<code>qty</code>	INTEGER	CHECK(qty > 0)		The number of items brought for this line item.	1



items

Field Name	Data Type	Constraint	Default	Description	Example
<code>item_id</code>	SERIAL	Primary Key	auto-increment	Unique identifier generated for each item	545
<code>description</code>	TEXT	NOT NULL		The description of the item	Mellowship Slinky Decaf

customers

Field Name	Data Type	Constraint	Default	Description	Example
<code>customer_id</code>	SERIAL	Primary Key	auto-increment	Unique identifier generated for each customer	8734
<code>name</code>	TEXT	NOT NULL		The name of the customer	Gavin Smith

orders

Field Name	Data Type	Constraint	Default	Description	Example
<code>order_id</code>	SERIAL	Primary Key	auto-increment	Unique identifier generated for each order	123456
<code>order_ts</code>	TIMESTAMP	NOT NULL	now()	Timestamp of when the order was placed	2017-01-01 13:01:02
<code>customer_id</code>	INTEGER	NOT NULL		The id of the customer that made the order.	8734

QUESTION

How many customers shop with us more than once a month?

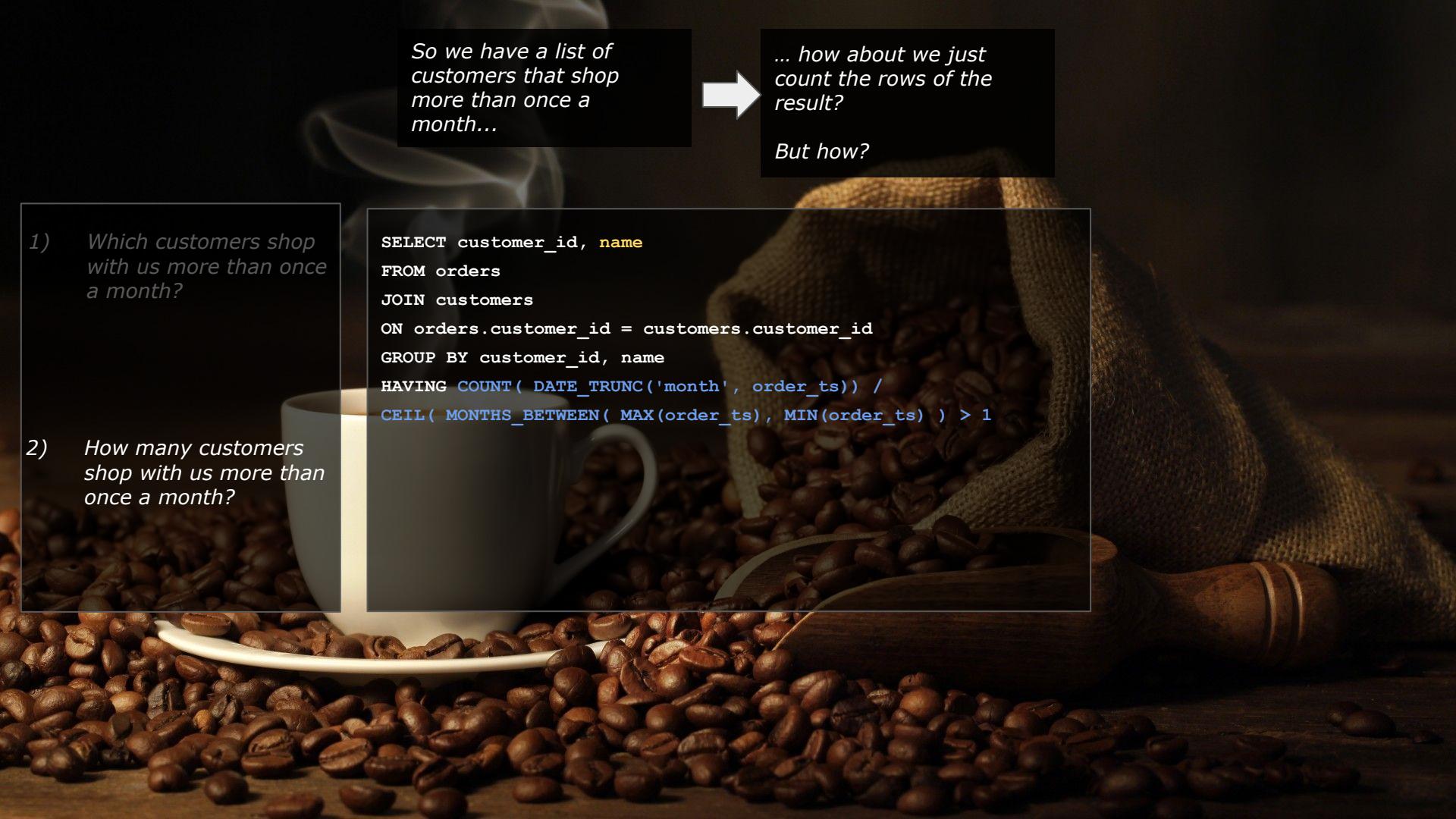
Wait... this looks very similar to before...

Before:

List the customers by name who shop with us more than once a month?

Let's start from there..





So we have a list of customers that shop more than once a month...

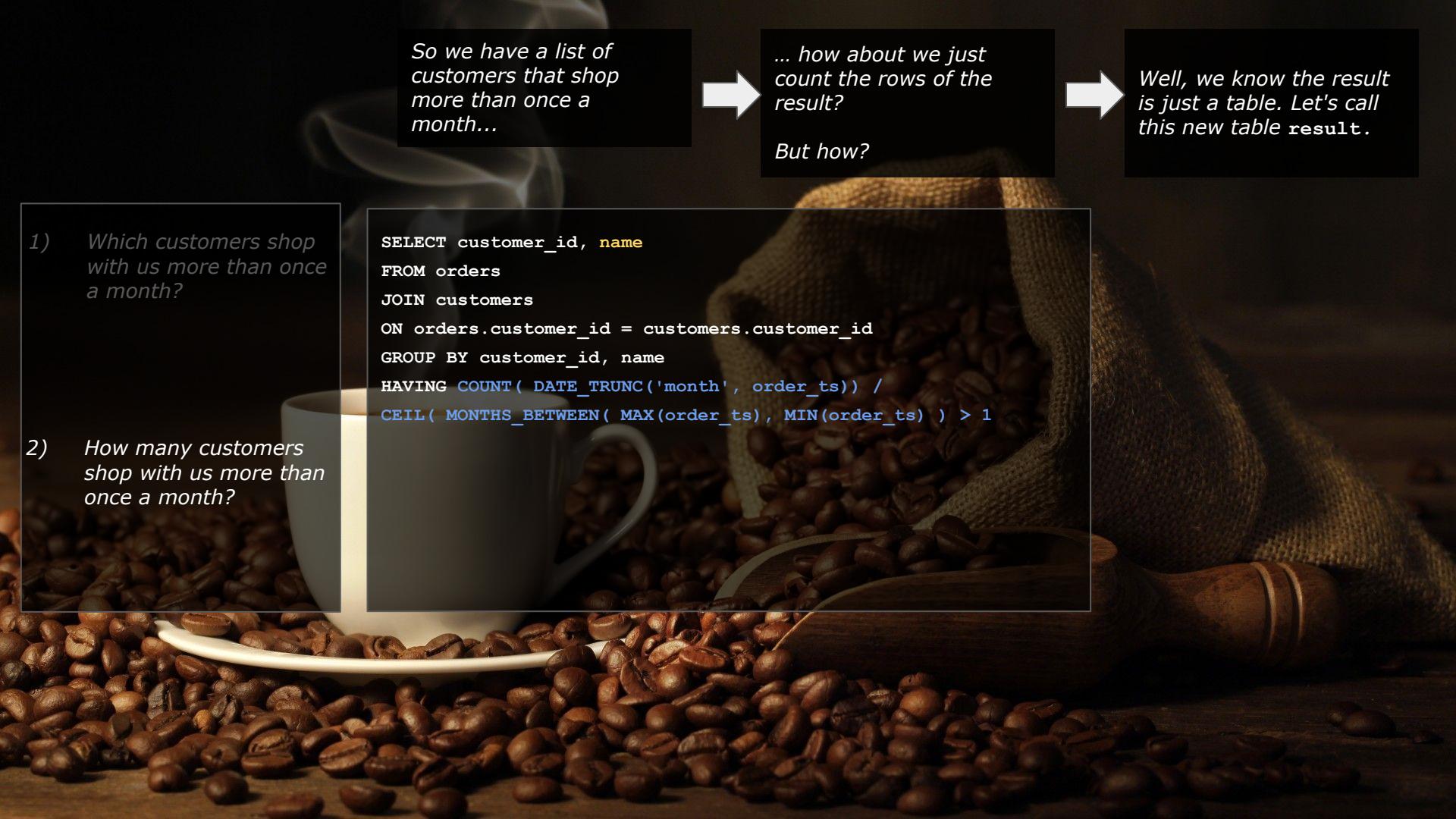
... how about we just count the rows of the result?

But how?

1) *Which customers shop with us more than once a month?*

2) *How many customers shop with us more than once a month?*

```
SELECT customer_id, name  
FROM orders  
JOIN customers  
ON orders.customer_id = customers.customer_id  
GROUP BY customer_id, name  
HAVING COUNT( DATE_TRUNC('month', order_ts)) /  
CEIL( MONTHS_BETWEEN( MAX(order_ts), MIN(order_ts) ) ) > 1
```



So we have a list of customers that shop more than once a month...

... how about we just count the rows of the result?

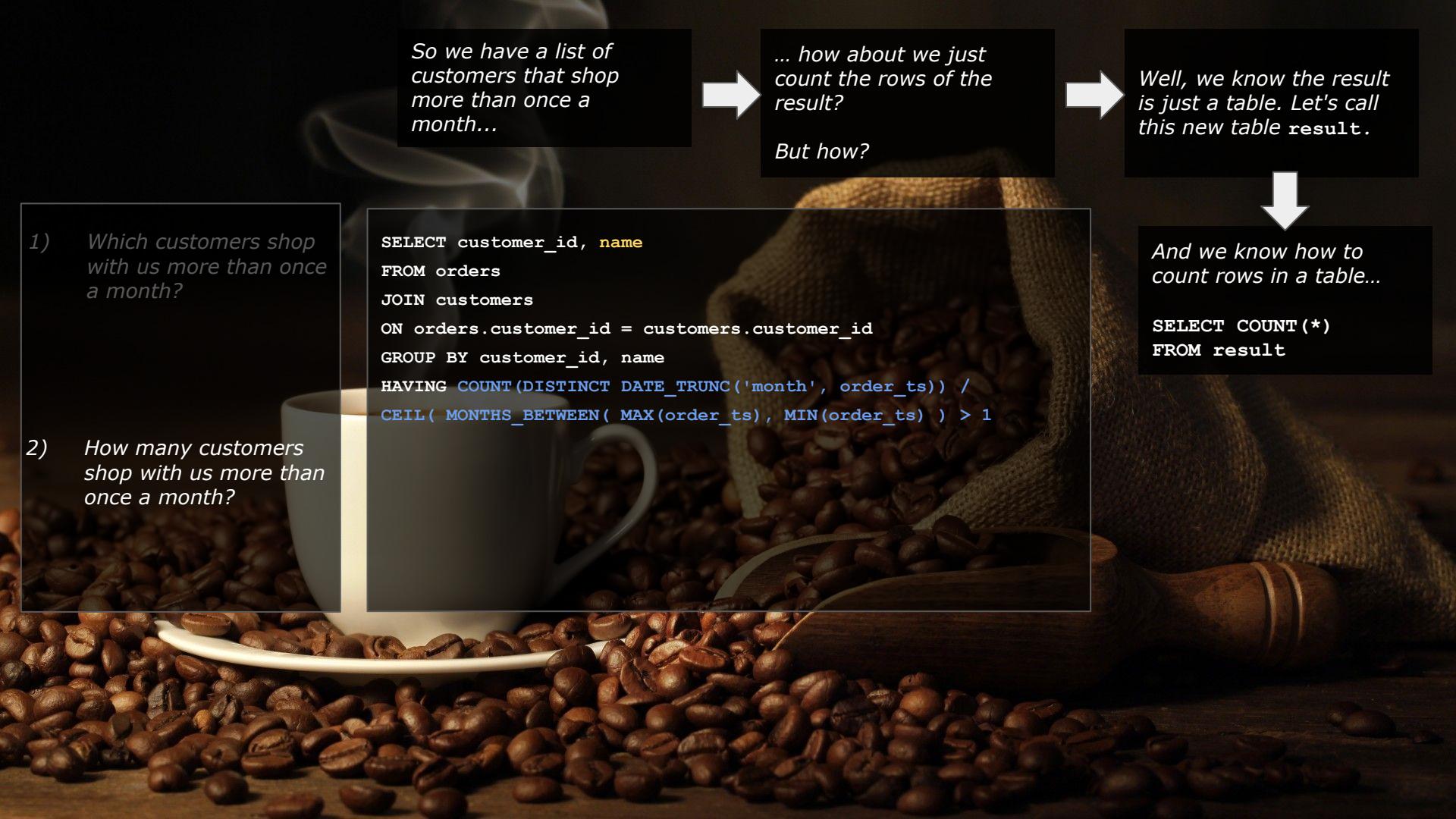
But how?

Well, we know the result is just a table. Let's call this new table **result**.

1) Which customers shop with us more than once a month?

2) How many customers shop with us more than once a month?

```
SELECT customer_id, name
FROM orders
JOIN customers
ON orders.customer_id = customers.customer_id
GROUP BY customer_id, name
HAVING COUNT( DATE_TRUNC('month', order_ts)) /
CEIL( MONTHS_BETWEEN( MAX(order_ts), MIN(order_ts) ) ) > 1
```

- 
- 1) Which customers shop with us more than once a month?
 - 2) How many customers shop with us more than once a month?

So we have a list of customers that shop more than once a month...

... how about we just count the rows of the result?

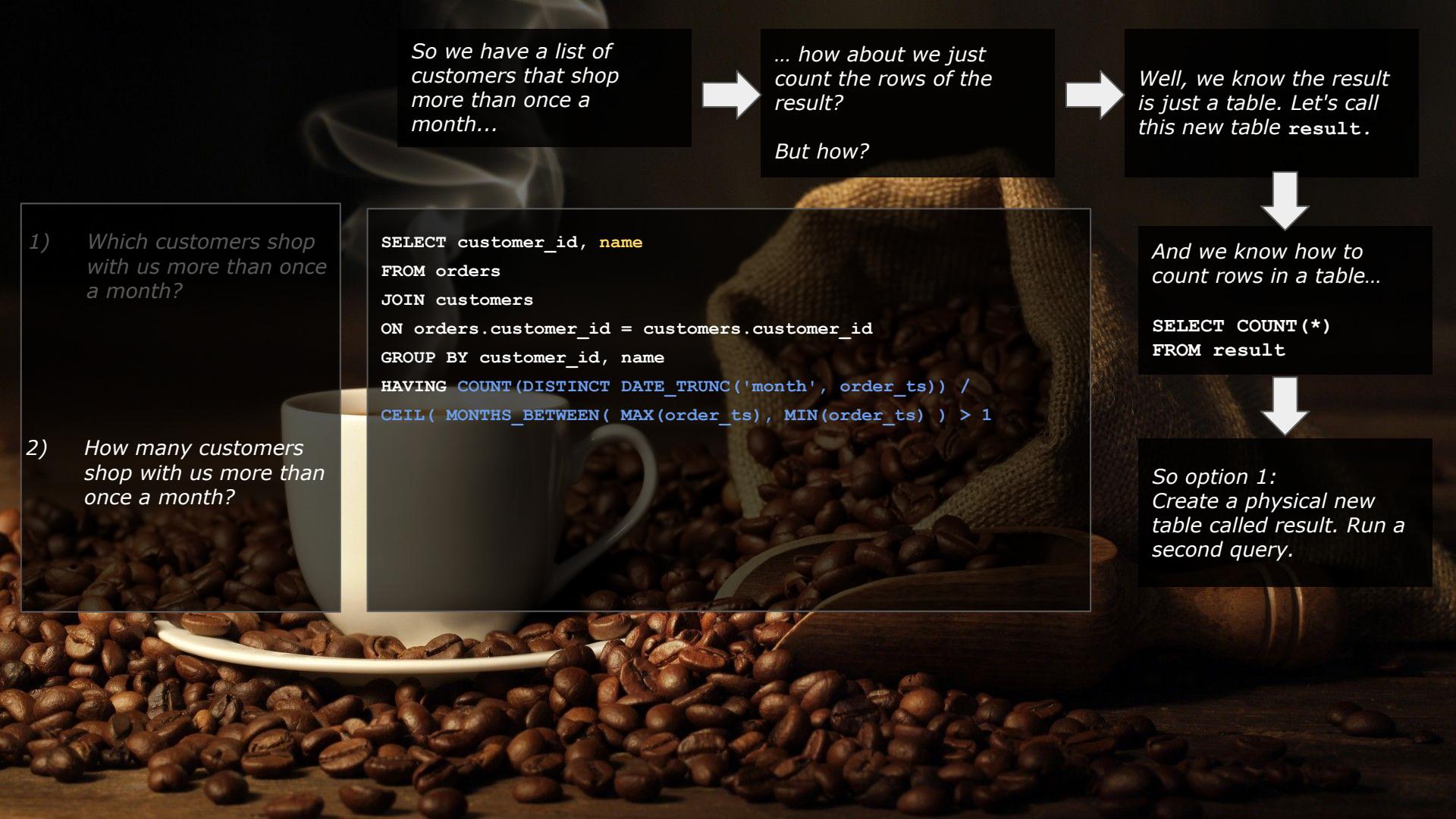
But how?

```
SELECT customer_id, name  
FROM orders  
JOIN customers  
ON orders.customer_id = customers.customer_id  
GROUP BY customer_id, name  
HAVING COUNT(DISTINCT DATE_TRUNC('month', order_ts)) /  
CEIL(MONTHS_BETWEEN(MAX(order_ts), MIN(order_ts))) > 1
```

Well, we know the result is just a table. Let's call this new table **result**.

And we know how to count rows in a table...

```
SELECT COUNT(*)  
FROM result
```

- 
- 1) Which customers shop with us more than once a month?
 - 2) How many customers shop with us more than once a month?

So we have a list of customers that shop more than once a month...

... how about we just count the rows of the result?

But how?

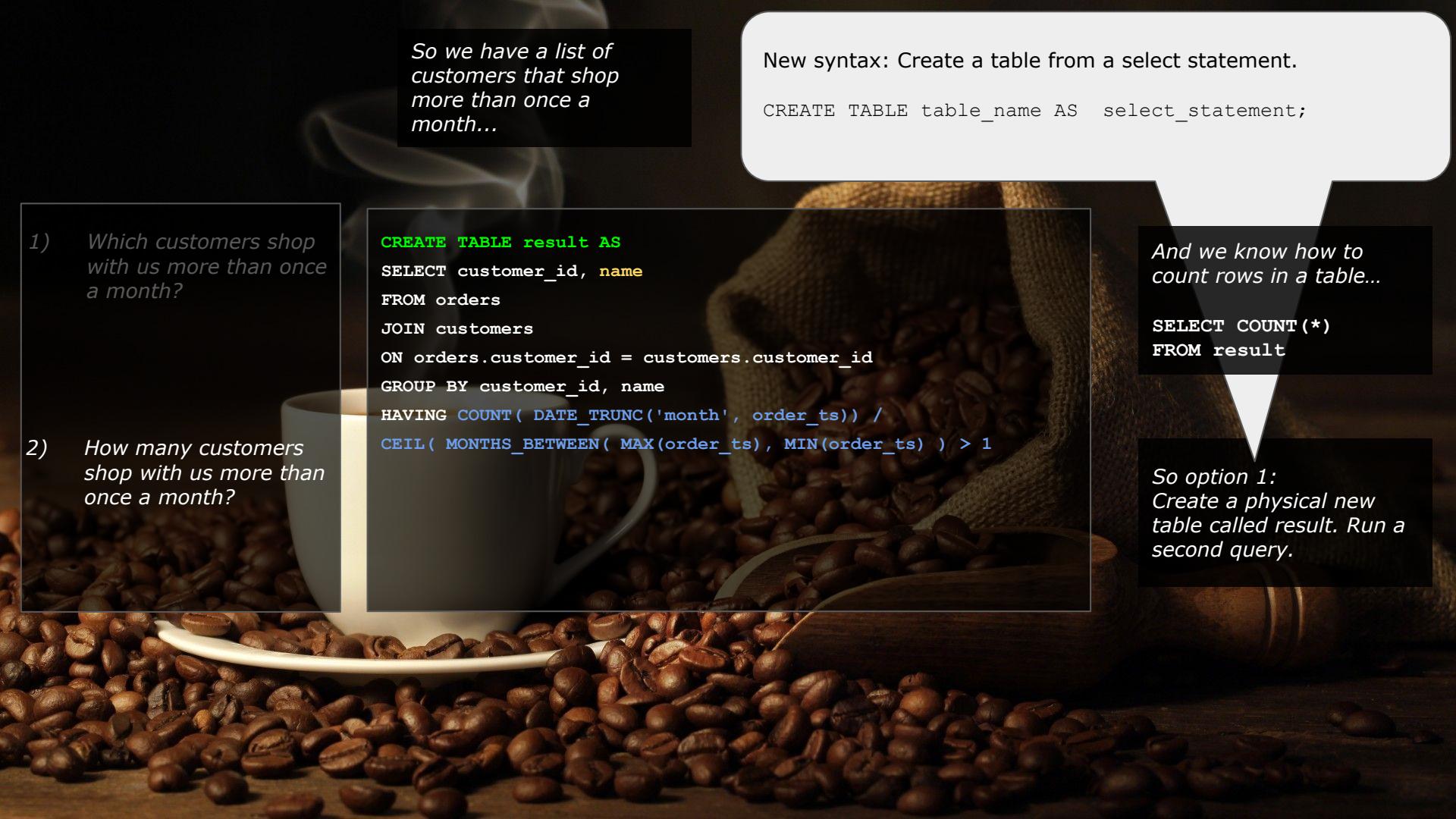
```
SELECT customer_id, name  
FROM orders  
JOIN customers  
ON orders.customer_id = customers.customer_id  
GROUP BY customer_id, name  
HAVING COUNT(DISTINCT DATE_TRUNC('month', order_ts)) /  
CEIL(MONTHS_BETWEEN(MAX(order_ts), MIN(order_ts))) > 1
```

Well, we know the result is just a table. Let's call this new table **result**.

And we know how to count rows in a table...

```
SELECT COUNT(*)  
FROM result
```

So option 1:
Create a physical new table called **result**. Run a second query.

- 
- 1) Which customers shop with us more than once a month?
 - 2) How many customers shop with us more than once a month?

So we have a list of customers that shop more than once a month...

New syntax: Create a table from a select statement.

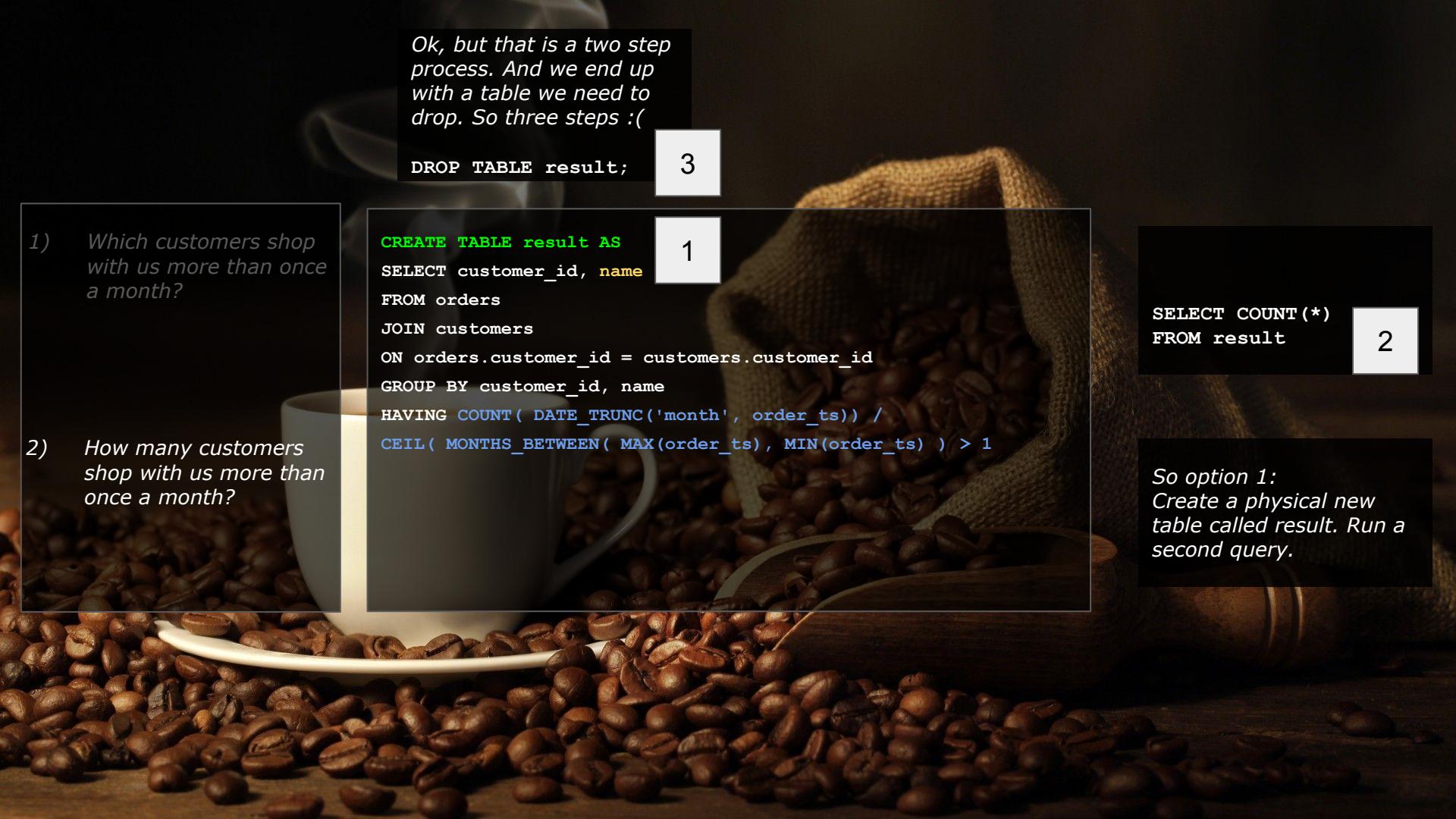
```
CREATE TABLE table_name AS select_statement;
```

```
CREATE TABLE result AS
SELECT customer_id, name
FROM orders
JOIN customers
ON orders.customer_id = customers.customer_id
GROUP BY customer_id, name
HAVING COUNT( DATE_TRUNC('month', order_ts)) /
CEIL( MONTHS_BETWEEN( MAX(order_ts), MIN(order_ts) ) ) > 1
```

And we know how to count rows in a table...

```
SELECT COUNT(*)
FROM result
```

So option 1:
Create a physical new table called result. Run a second query.



Ok, but that is a two step process. And we end up with a table we need to drop. So three steps :(

DROP TABLE result;

3

- 1) *Which customers shop with us more than once a month?*
- 2) *How many customers shop with us more than once a month?*

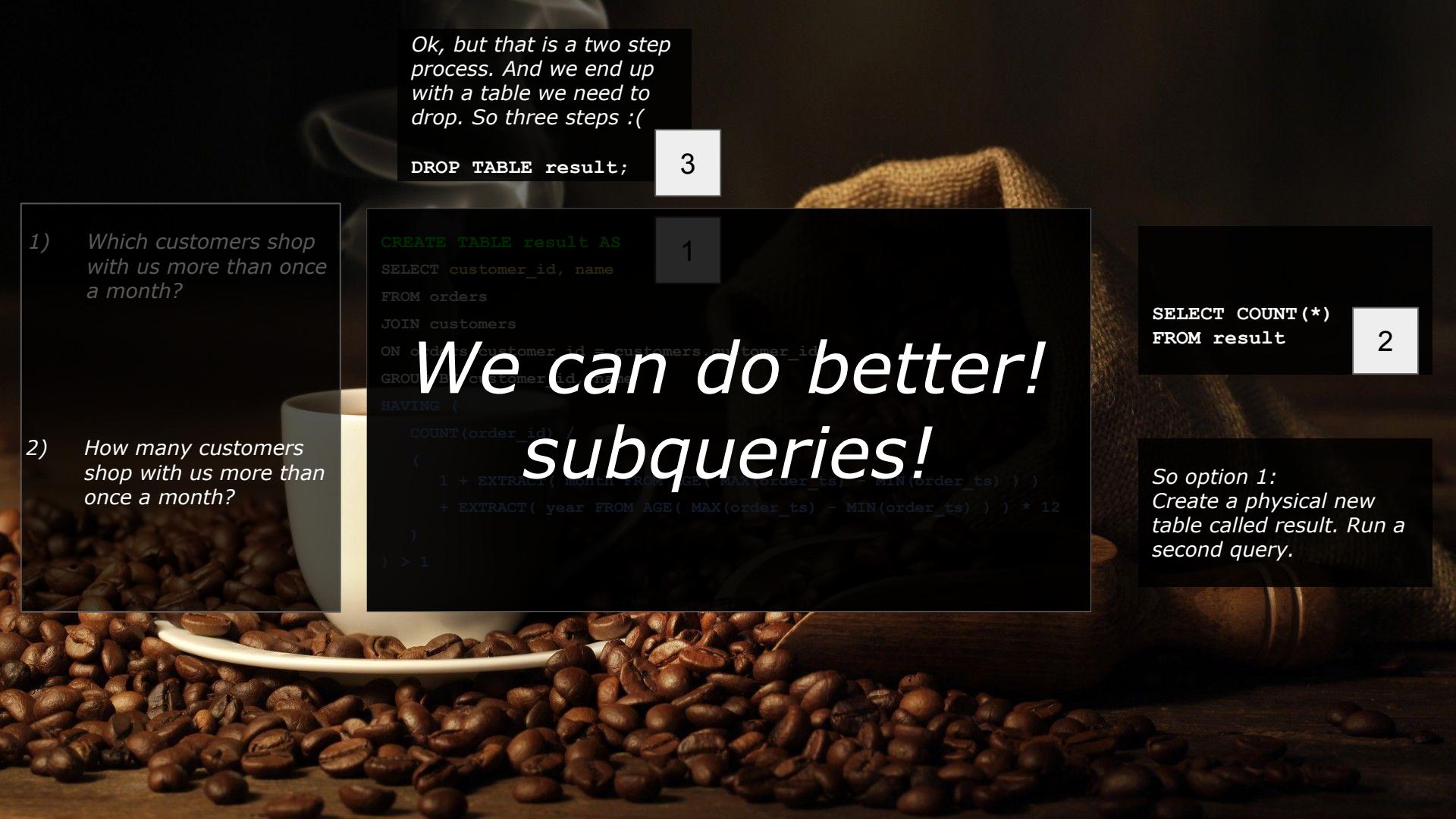
```
CREATE TABLE result AS
SELECT customer_id, name
FROM orders
JOIN customers
ON orders.customer_id = customers.customer_id
GROUP BY customer_id, name
HAVING COUNT( DATE_TRUNC('month', order_ts)) /
CEIL( MONTHS_BETWEEN( MAX(order_ts), MIN(order_ts) ) ) > 1
```

1

SELECT COUNT(*)
FROM result

2

*So option 1:
Create a physical new table called result. Run a second query.*



Ok, but that is a two step process. And we end up with a table we need to drop. So three steps :(

DROP TABLE result;

3

- 1) *Which customers shop with us more than once a month?*

- 2) *How many customers shop with us more than once a month?*

```
CREATE TABLE result AS
SELECT customer_id, name
FROM orders
JOIN customers
ON orders.customer_id = customers.customer_id
GROUP BY customer_id
HAVING (
    COUNT(order_id) /
    (
        1 + EXTRACT( month FROM AGE( MAX(order_ts), MIN(order_ts) ) )
        + EXTRACT( year FROM AGE( MAX(order_ts) - MIN(order_ts) ) ) * 12
    )
) > 1
```

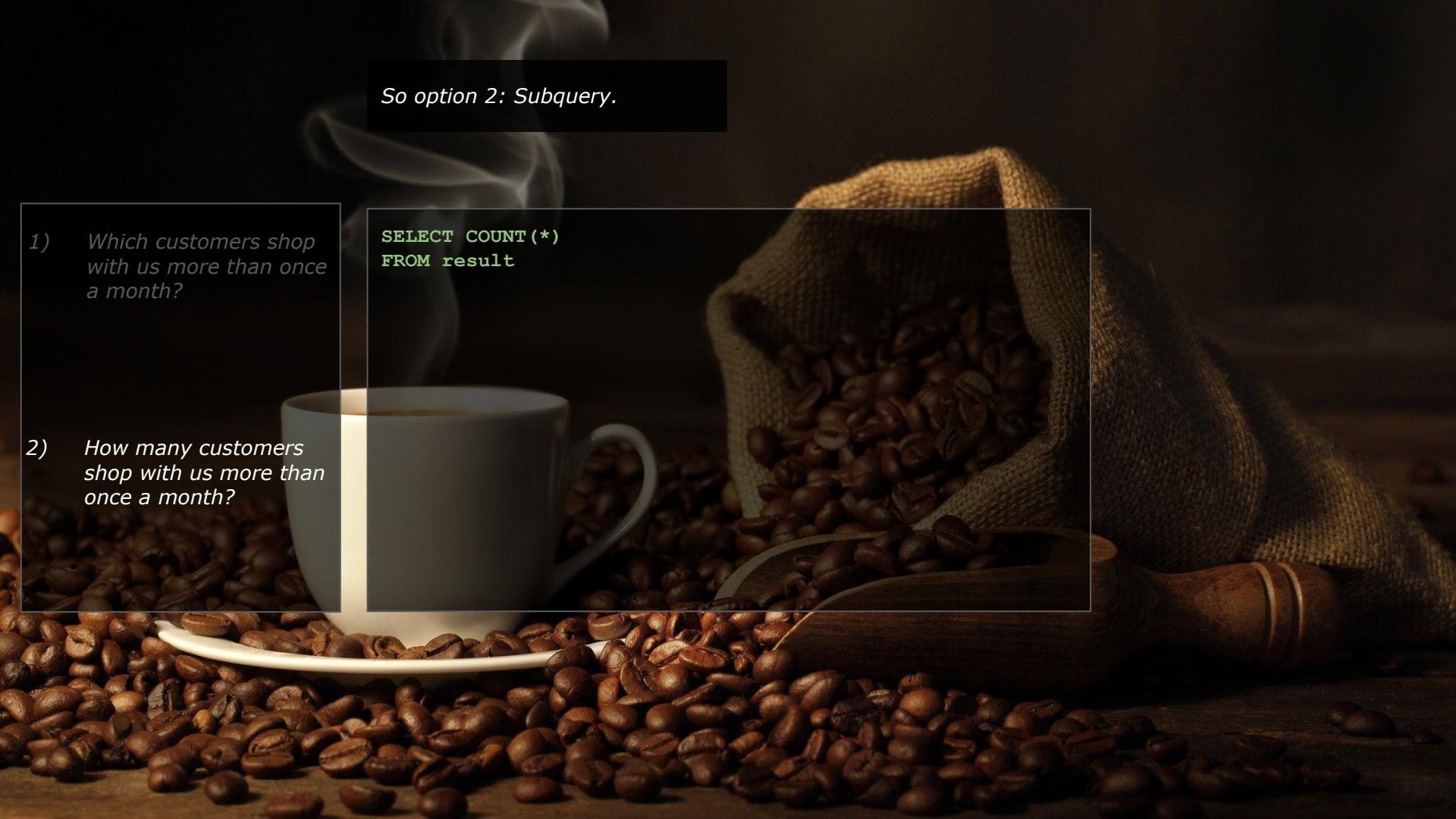
1

SELECT COUNT(*)
FROM result

2

We can do better! subqueries!

*So option 1:
Create a physical new
table called result. Run a
second query.*

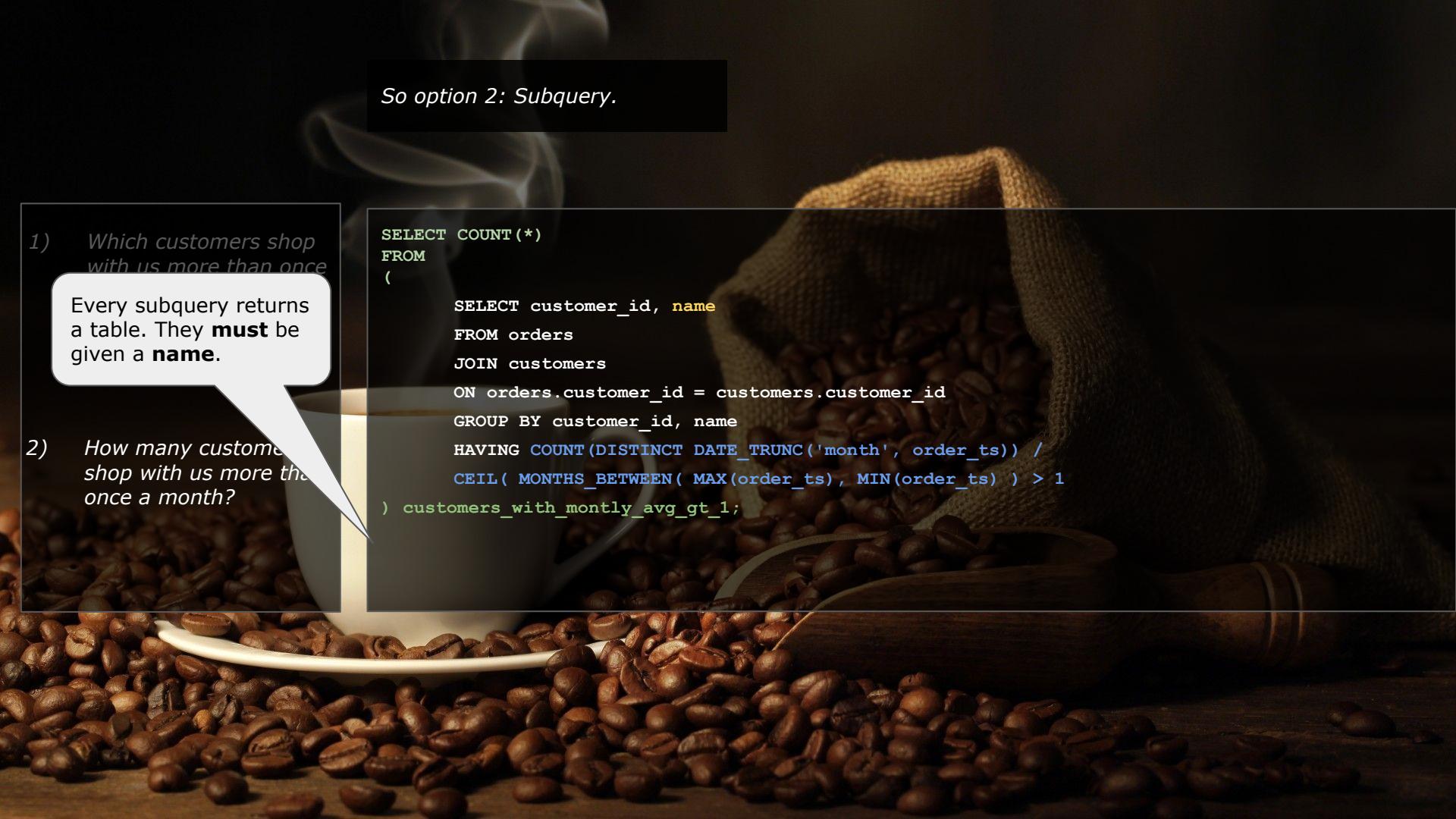


So option 2: Subquery.

- 1) *Which customers shop with us more than once a month?*

- 2) *How many customers shop with us more than once a month?*

```
SELECT COUNT(*)  
FROM result
```



So option 2: Subquery.

- 1) *Which customers shop with us more than once*

Every subquery returns a table. They **must** be given a **name**.

- 2) *How many customers shop with us more than once a month?*

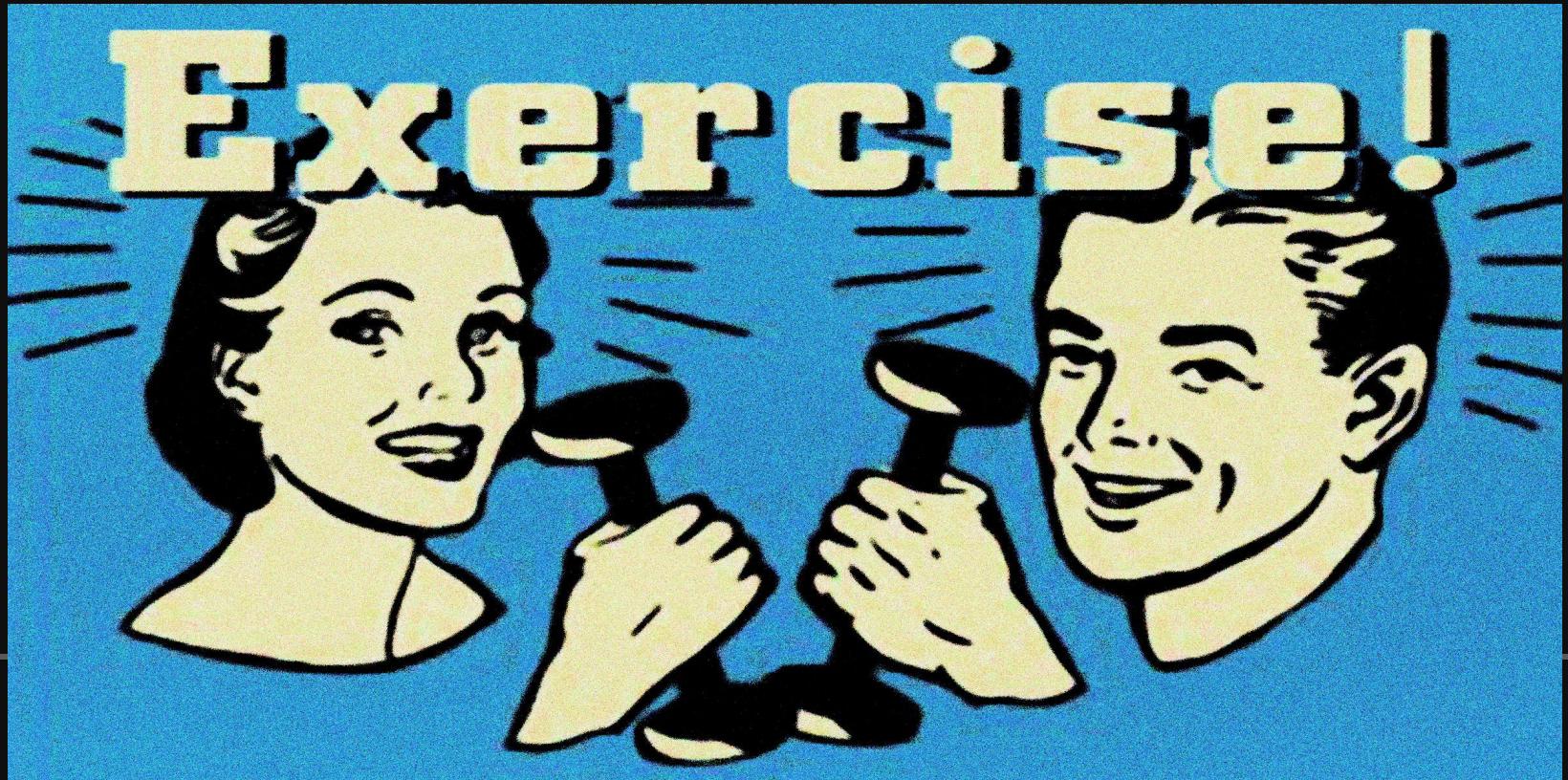
```
SELECT COUNT(*)
FROM
(
    SELECT customer_id, name
    FROM orders
    JOIN customers
    ON orders.customer_id = customers.customer_id
    GROUP BY customer_id, name
    HAVING COUNT(DISTINCT DATE_TRUNC('month', order_ts)) /
        CEIL( MONTHS_BETWEEN( MAX(order_ts), MIN(order_ts) ) ) > 1
) customers_with_monthly_avg_gt_1;
```

**Well, that was
easier.
And we're done!**

- 1) *Which customers shop with us more than once a month?*

- 2) *How many customers shop with us more than once a month?*

```
SELECT COUNT(*)
FROM
(
    SELECT customer_id, customers.name
    FROM orders, customers
    WHERE orders.customer_id = customers.customer_id
    GROUP BY customer_id, name
    HAVING COUNT( DATE_TRUNC('month', order_ts)) /
        CEIL( MONTHS_BETWEEN( MAX(order_ts), MIN(order_ts) ) ) > 1
) customers_with_montly_avg_gt_1;
```



NLAB: *Data at Scale*

Dr Georgiana Nica-Avram

Session 15 & 16

Implementing KPIs (SQL IV)



SQL. Again.

SQL

We've covered **most** SQL.

You can solve **almost all problems with what you know!**

However, there are some shortcuts and a few little things.

SQL. Again.

SQL

Thinking in SQL takes practice.

The more examples you see the better!

Today . . .

- Example 1
- Example 2
- Example 3
- Example 4
- Example 5
- Example 6
- Example 7

Through these examples we'll also introduce a few new SQL concepts and syntax.

SQL. Again.



SQL. Again.

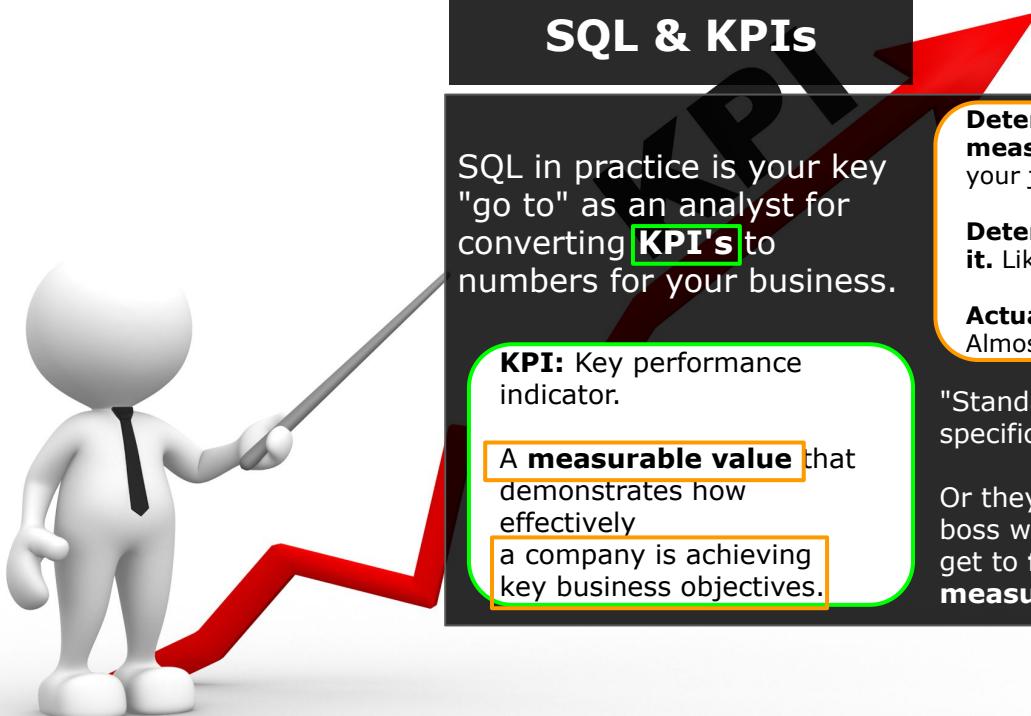
SQL & KPIs

SQL in practice is your key "go to" as an analyst for converting **KPI's** to numbers for your business.

KPI: Key performance indicator.

A **measurable value** that demonstrates how effectively a company is achieving key business objectives.

SQL. Again.



SQL & KPIs

SQL in practice is your key "go to" as an analyst for converting **KPI's** to numbers for your business.

KPI: Key performance indicator.

A **measurable value** that demonstrates how effectively a company is achieving key business objectives.

Determining what to measure. May or may not be your job.

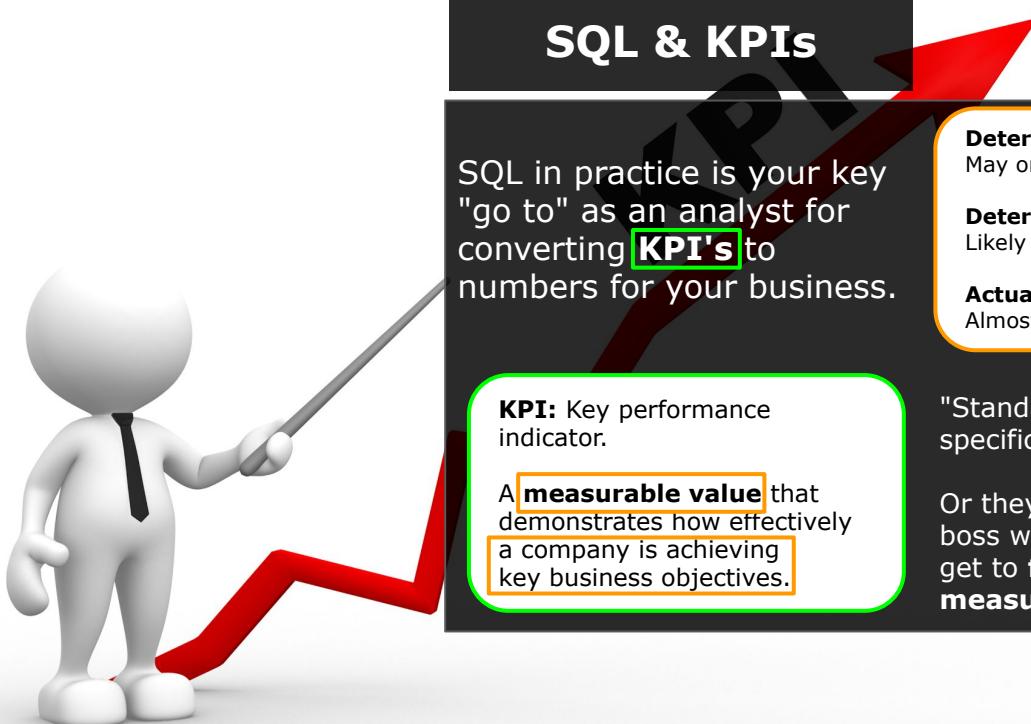
Determining how to measure it. Likely your job.

Actually getting the data. Almost certainly your job.

"Standard" KPI's may have a specific formula.

Or they may be something your boss wants **you** to measure. You get to **formalize how it is measured & measure it.**

SQL. Again.



SQL & KPIs

SQL in practice is your key "go to" as an analyst for converting **KPI's** to numbers for your business.

KPI: Key performance indicator.

A **measurable value** that demonstrates how effectively a company is achieving key business objectives.

Determining what to measure.
May or may not be your job.

Determining how to measure it.
Likely your job.

Actually getting the data.
Almost certainly your job.

"Standard" KPI's may have a specific formula.

Or they may be something your boss wants **you** to measure. You get to **formalize how it is measured & measure it.**

Examples of KPIs:

Retail

Sales & Gross Margin

Sales per Square Foot

Average customer spend

Stock turnover rate

Return on investment on marketing spend

Growth

Engagement

Retention rate

Churn rate

Supply chain

Back order rate

Rate of return

% of out-of-stock items

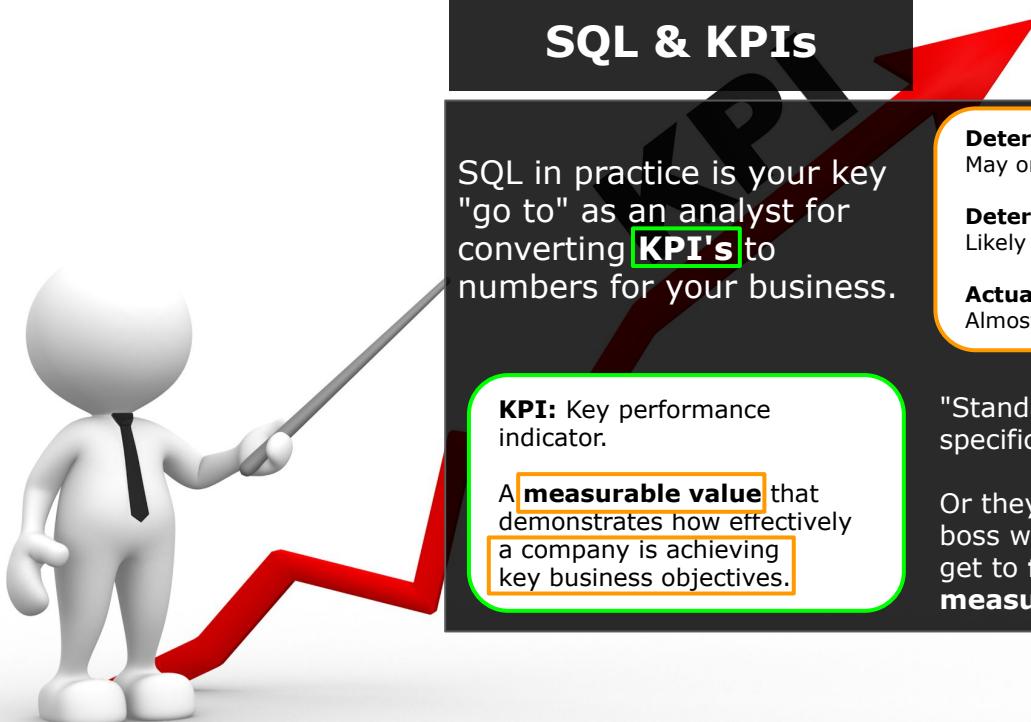
Perfect order rate

Inventory turnover

Carrying cost of inventory

Inventory Accuracy

SQL. Again.



SQL & KPIs

SQL in practice is your key "go to" as an analyst for converting **KPI's** to numbers for your business.

KPI: Key performance indicator.

A **measurable value** that demonstrates how effectively a company is achieving key business objectives.

Determining what to measure.
May or may not be your job.

Determining how to measure it.
Likely your job.

Actually getting the data.
Almost certainly your job.

"Standard" KPI's may have a specific formula.

Or they may be something your boss wants **you** to measure. You get to **formalize how it is measured & measure it.**

Examples of KPIs:

Retail

Sales & Gross Margin

Sales per Square Foot

Average customer spend

Stock turnover rate

Return on investment on marketing spend

Growth

Engagement

Retention rate

Churn rate

Supply chain

Back order rate

Rate of return

% of out-of-stock items

Perfect order rate

Inventory turnover

Carrying cost of inventory

Inventory Accuracy

Example 1

Growth
Engagement
Retention rate

Step 1:

Write down a concrete measurable description of the KPI

Step 2. Write in SQL.



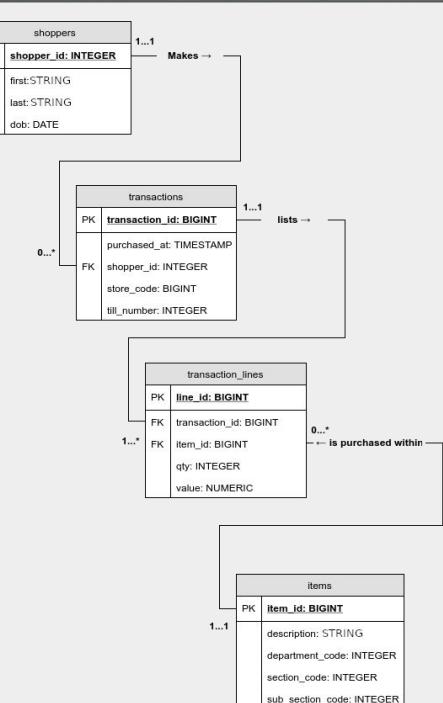
Example 1

Growth
Engagement
Retention rate

Step 1:

Write down a concrete measurable description of the KPI

*Growth, measured as
→ Count of active customers per month (KPI 1)*



Step 2. Write in SQL.



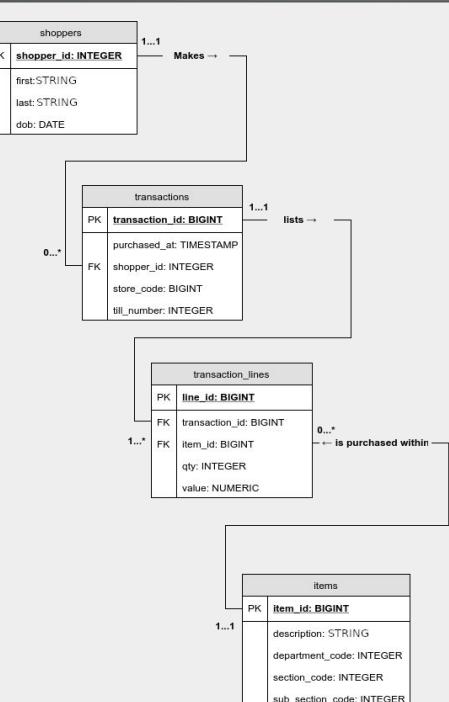
Example 1

Growth
Engagement
Retention rate

Step 1:

Write down a concrete measurable description of the KPI

*Growth, measured as
→ Count of active customers per
month (KPI 1)*



Step 2. Write in SQL.

```
SELECT  
FROM
```



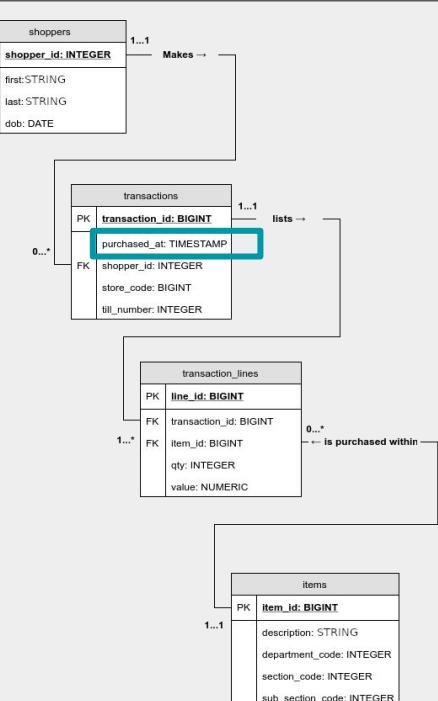
Example 1

Growth
Engagement
Retention rate

Step 1:

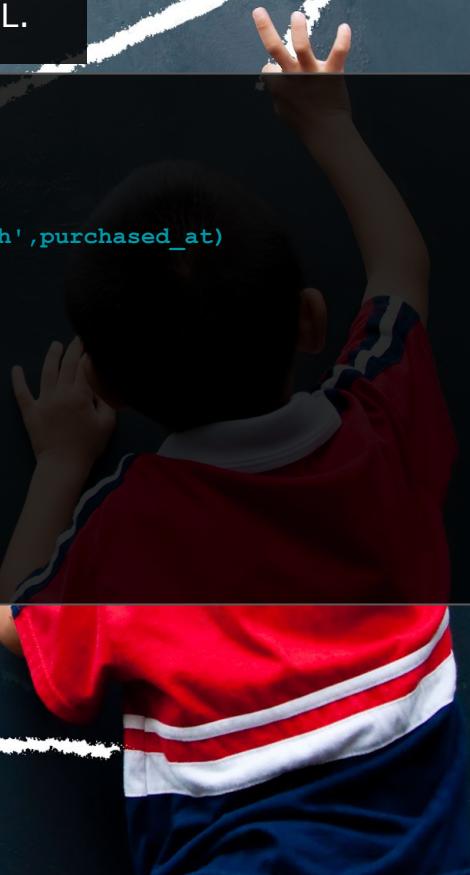
Write down a concrete measurable description of the KPI

*Growth, measured as
→ Count of active customers per month (KPI 1)*



Step 2. Write in SQL.

```
SELECT
  FROM transactions
  GROUP BY DATE_TRUNC('month', purchased_at)
```



Example 1

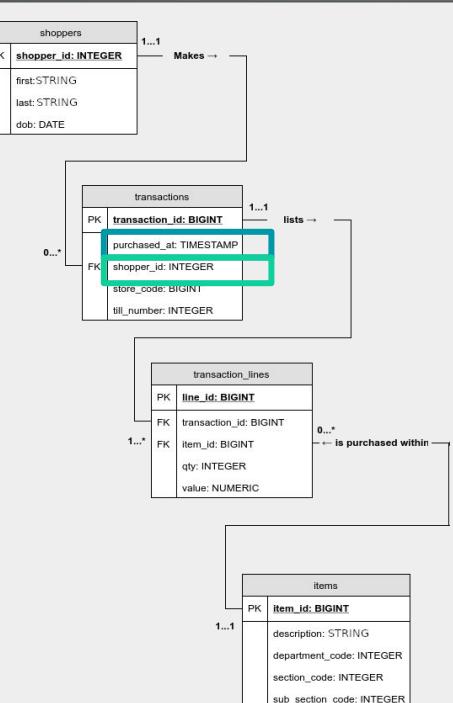
Growth
Engagement
Retention rate

Step 1:

Write down a concrete measurable description of the KPI

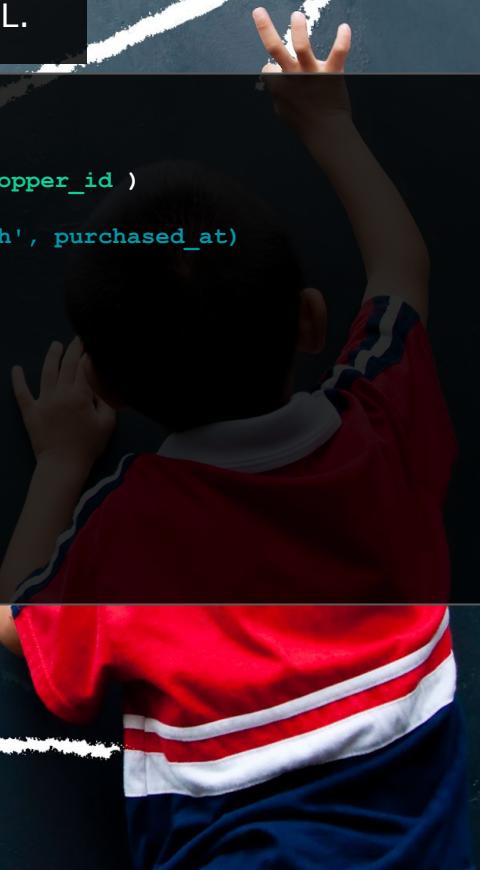
*Growth, measured as
→ Count of active customers per month (KPI 1)*

Implicitly we mean distinct here, otherwise we'd be counting customer visits



Step 2. Write in SQL.

```
SELECT COUNT( DISTINCT shopper_id )
FROM transactions
GROUP BY DATE_TRUNC('month', purchased_at)
```



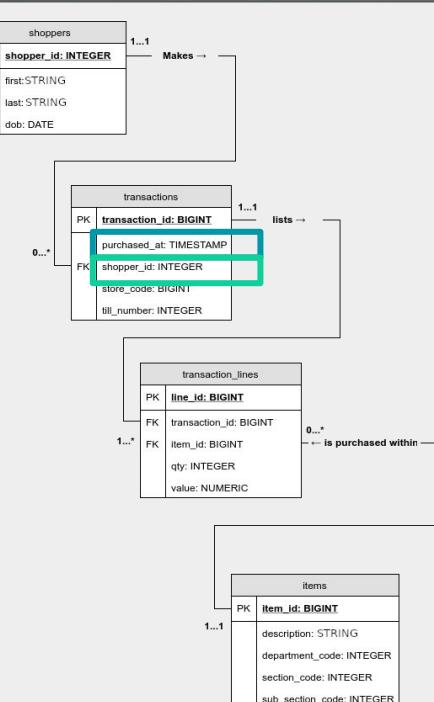
Example 1

Growth
Engagement
Retention rate

Step 1:

Write down a concrete measurable description of the KPI

*Growth, measured as
→ Count of active customers per month (KPI 1)*



Step 2. Write in SQL.

```
SELECT COUNT( DISTINCT shopper_id ) AS active_customers,
       DATE_FORMAT(DATE_TRUNC('month',purchased_at),'yyyy-MM') AS mth
  FROM transactions
 GROUP BY DATE_TRUNC('month',purchased_at)
 ORDER BY 2 DESC;
```

We now "pretty-up" the output so:

(1) it gives the headings we want

and

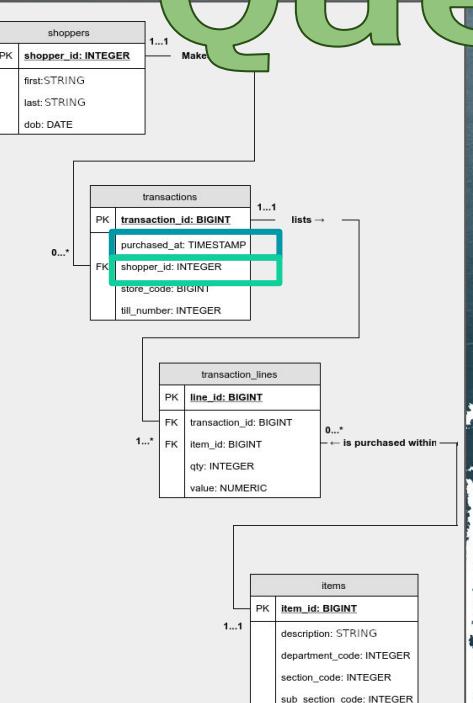
(2) it presents to the user in the order we want.

Growth
Engagement
Retention rate

Step 1:

Write down a concrete measurable description of the KPI

*Growth, measured as
→ Count of active customers per month (KPI 1)*



Step 2. Write in SQL.

```
SELECT COUNT( DISTINCT shopper_id ) AS active_customers,
       DATE_FORMAT(DATE_TRUNC('month',purchased_at),'yyyy-MM') AS mth
  FROM transactions
 GROUP BY DATE_TRUNC('month',purchased_at)
 ORDER BY 2 DESC;
```

We now "pretty-up" the output so:

(1) it gives the headings we want

and

(2) it presents to the user in the order we want.

Example 2

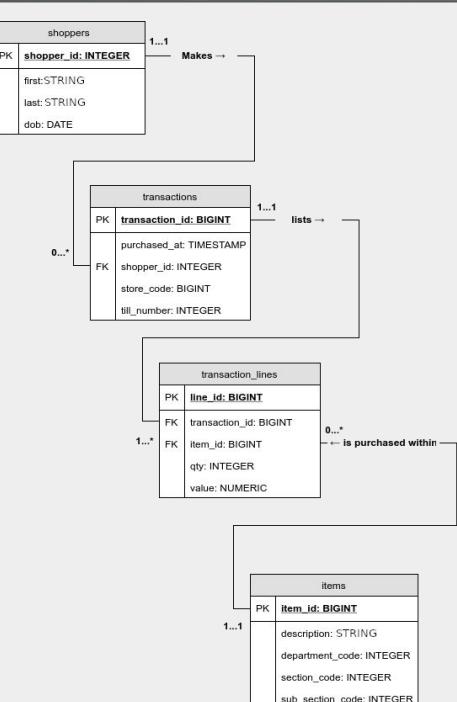
Growth
Engagement
Retention rate

Step 1:

Write down a concrete measurable description of the KPI

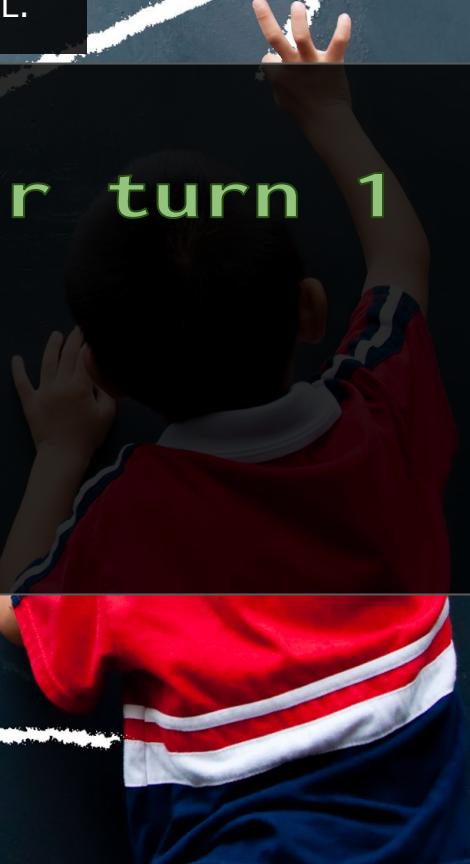
Growth, measured as
→ Count of active customers per month (KPI 1)

→ Total sales per month for store 5000128068369 (KPI 2)



Step 2. Write in SQL.

Your turn 1



Example 2

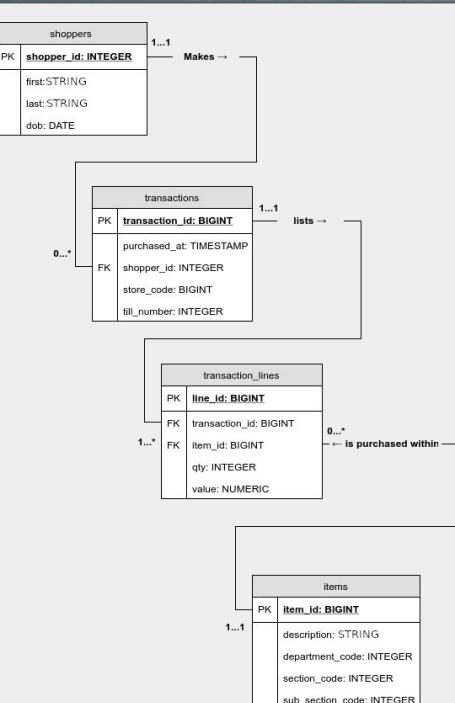
Growth
Engagement
Retention rate

Step 1:

Write down a concrete measurable description of the KPI

Growth, measured as
→ Count of active customers per month (KPI 1)

→ Total sales per month for store 5000128068369 (KPI 2)



Step 2. Write in SQL.

```
SELECT SUM(value) AS total_sales,  
       DATE_FORMAT(DATE_TRUNC('month', purchased_at), 'yyyy-MM') AS mth  
  FROM transactions  
  JOIN transaction_lines  
    USING (transaction_id)  
   WHERE store_code = 5000128068369  
 GROUP BY DATE_TRUNC('month', purchased_at);
```

Example 3

Growth
Engagement
Retention rate

Step 1:

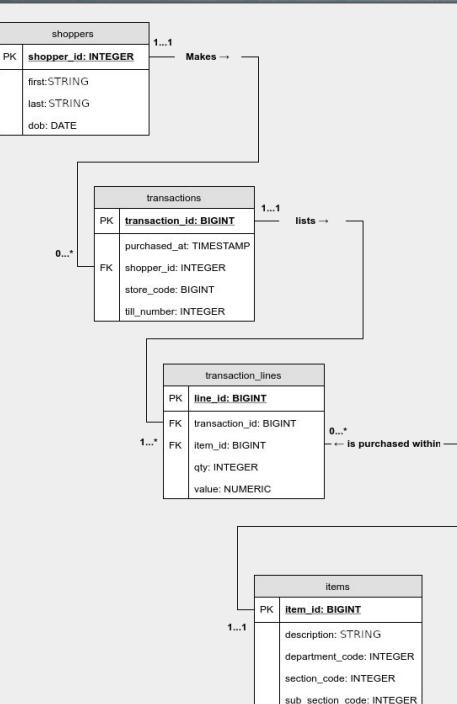
Write down a concrete measurable description of the KPI

Growth, measured as

→ Count of active customers per month (KPI 1)

→ Total sales per month for store 5000128068369 (KPI 2)

→ Total sales per month, per store (KPI 3)



Step 2. Write in SQL.

```
SELECT store_code, SUM(value) AS total_sales,  
       DATE_FORMAT(DATE_TRUNC('month', purchased_at), 'yyyy-MM') AS mth  
  FROM transactions  
  JOIN transaction_lines  
    USING (transaction_id)  
 WHERE store_code = 5000128068369  
 GROUP BY DATE_TRUNC('month', purchased_at), store_code
```

Example 3

Growth
Engagement
Retention rate

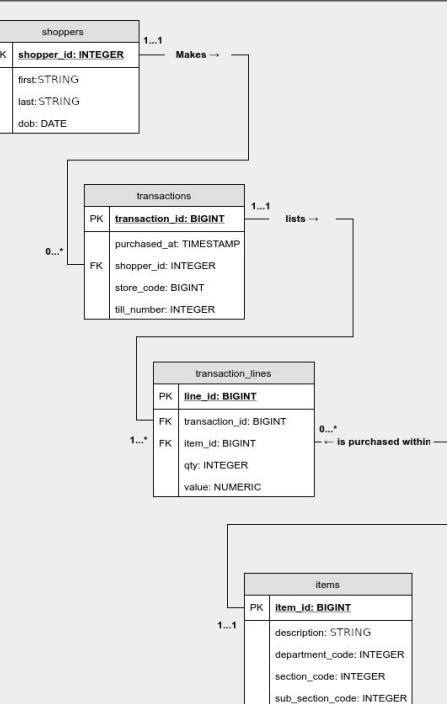
Step 1:

Write down a concrete measurable description of the KPI

Growth, measured as
→ Count of active customers per month (KPI 1)

→ Total sales per month for store 5000128068369 (KPI 2)

→ Total sales per month, per store (KPI 3)



Step 2. Write in SQL.

```
SELECT store_code, SUM(value) AS total_sales,  
       DATE_FORMAT(DATE_TRUNC('month', purchased_at), 'yyyy-MM') AS mth  
  FROM transactions  
  JOIN transaction_lines  
    USING (transaction_id)  
 GROUP BY DATE_TRUNC('month', purchased_at), store_code;
```

But what about "prettying up"?
Two options.

- Use order by.
- Graph / visualize

Demo!

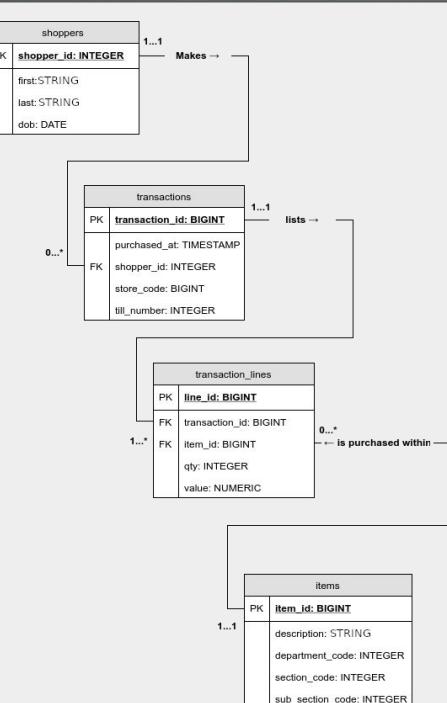
Example 4

Growth Engagement Retention rate

Step 1:

Write down a concrete measurable description of the KPI

*Engagement, measured as
→ Count of repeat customers per
month for store
5000128068369 (KPI 4)*



Step 2. Write in SQL.

```
SELECT
  FROM transactions
  WHERE store_code = 5000128068369
  GROUP BY DATE_TRUNC('month', purchased_at)
```

Question: How to encode the notion of "repeat"?

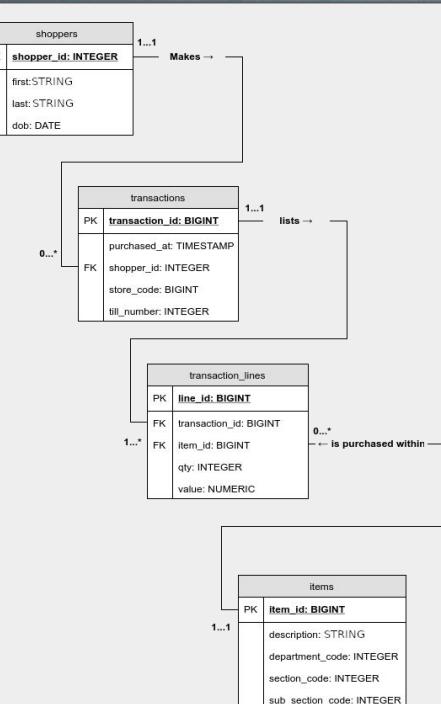
Example 4

Growth
Engagement
Retention rate

Step 1:

Write down a concrete measurable description of the KPI

*Engagement, measured as
→ Count of repeat customers per
month for store
5000128068369 (KPI 4)*



Step 2. Write in SQL.

```
SELECT
  FROM transactions
  WHERE store_code = 5000128068369
  GROUP BY DATE_TRUNC('month', purchased_at)
```

Question: How to encode the notion of "repeat"?

Answer: In each bucket, only count shoppers if they have two different transaction_ids in the bucket.

→ What assumptions have I made?

Example 4

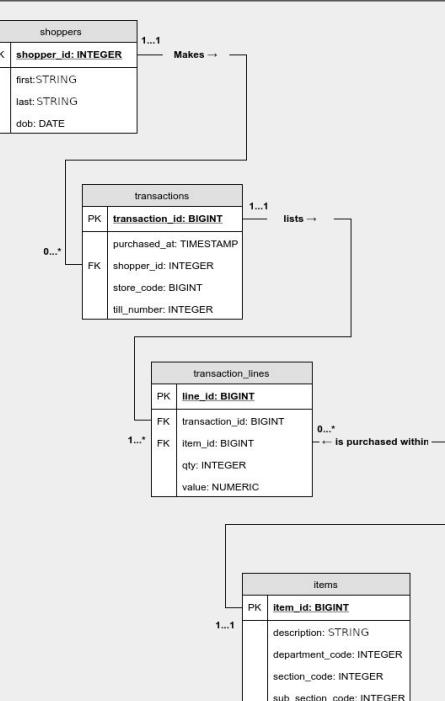
Growth Engagement Retention rate

Step 1:

Write down a concrete measurable description of the KPI

*Engagement, measured as
→ Count of repeat customers per
month for store
5000128068369 (KPI 4)*

*KPI 4 Assumptions:
→ Returning on the same day still
counts as a repeat customer.*



Step 2. Write in SQL.

```
SELECT
  FROM transactions
  WHERE store_code = 5000128068369
  GROUP BY DATE_TRUNC('month', purchased_at)
```

Question: How to encode the notion of "repeat"?

Answer: In each bucket, only count shoppers if they have two different transaction_ids in the bucket.

→ What assumptions have I made?

Example 4

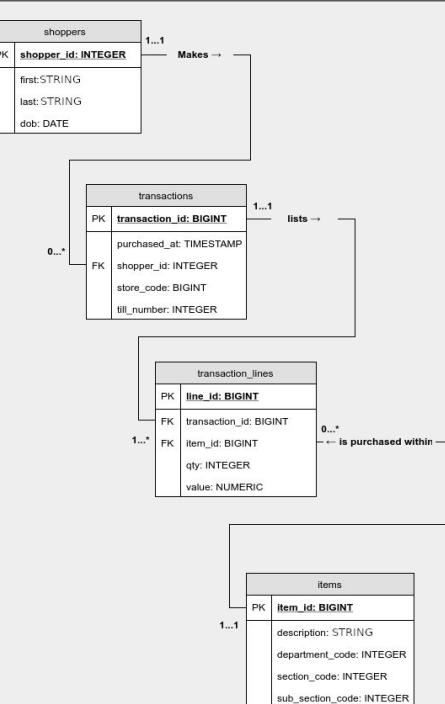
Growth Engagement Retention rate

Step 1:

Write down a concrete measurable description of the KPI

*Engagement, measured as
→ Count of repeat customers per
month for store
5000128068369 (KPI 4)*

*KPI 4 Assumptions:
→ Returning on the same day still
counts as a repeat customer.*



Step 2. Write in SQL.

```
SELECT DATE_TRUNC('month', purchased_at) as mth, shopper_id  
FROM transactions  
WHERE store_code = 5000128068369  
GROUP BY DATE_TRUNC('month', purchased_at), shopper_id
```

Counting buckets (groups) takes two steps.

- One query to make a table with one row per bucket [the subquery].
- One query to count the rows

The subquery:

First group by month + shopper_id.

Example 4

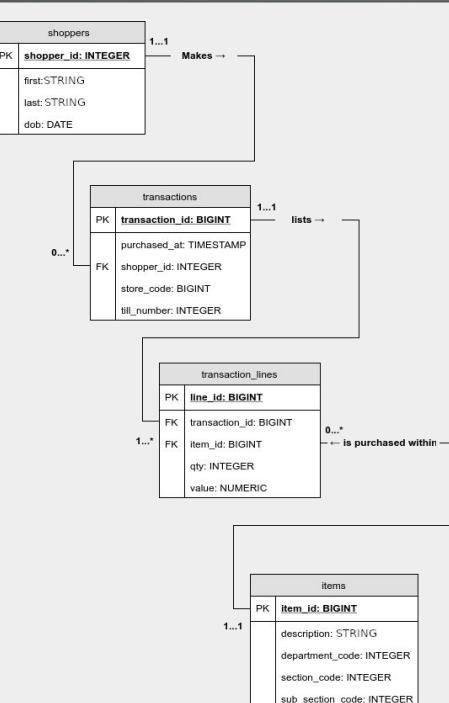
Growth Engagement Retention rate

Step 1:

Write down a concrete measurable description of the KPI

*Engagement, measured as
→ Count of repeat customers per
month for store
5000128068369 (KPI 4)*

*KPI 4 Assumptions:
→ Returning on the same day still
counts as a repeat customer.*



Step 2. Write in SQL.

```
SELECT DATE_TRUNC('month', purchased_at) as mth, shopper_id
FROM transactions
WHERE store_code = 5000128068369
GROUP BY DATE_TRUNC('month', purchased_at), shopper_id
HAVING COUNT(DISTINCT transaction_id) > 1
```

First group by month +
shopper_id.

Keep only those rows where
the count of distinct
transaction_ids > 1.

Question: What does
this table represent?

Example 4

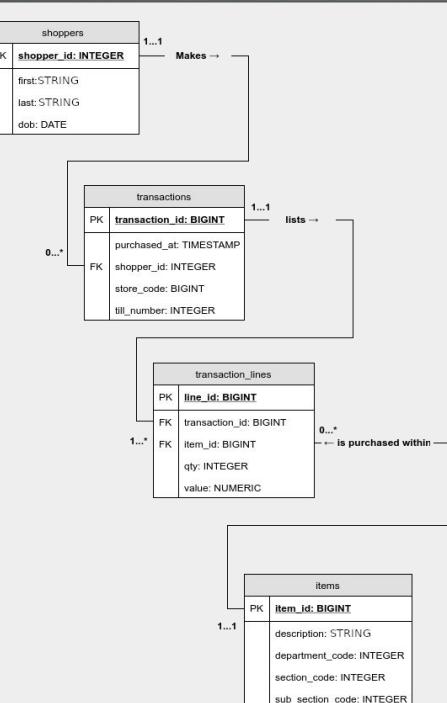
Growth Engagement Retention rate

Step 1:

Write down a concrete measurable description of the KPI

*Engagement, measured as
→ Count of repeat customers per
month for store
5000128068369 (KPI 4)*

*KPI 4 Assumptions:
→ Returning on the same day still
counts as a repeat customer.*



Step 2. Write in SQL.

```
SELECT DATE_TRUNC('month', purchased_at) as mth, shopper_id
FROM transactions
WHERE store_code = 5000128068369
GROUP BY DATE_TRUNC('month', purchased_at), shopper_id
HAVING COUNT(DISTINCT transaction_id) > 1
```

First group by month +
shopper_id.

Keep only those rows where
the count of distinct
transaction_ids > 1.

Question: What does this table represent?

Answer: Pairs of shopper_ids and months, where the shopper shopped at least twice in the month.

Fact known per row:
shopper_id shopped at least twice in mth

Example 4

Growth Engagement Retention rate

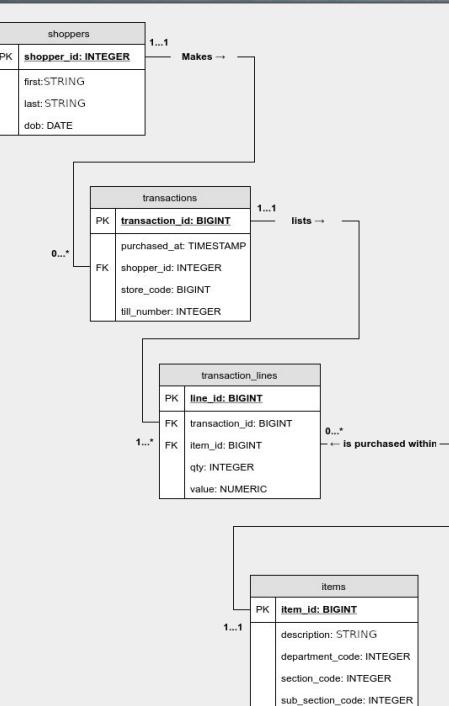
Step 1:

Write down a concrete measurable description of the KPI

*Engagement, measured as
→ Count of repeat customers per
month for store
5000128068369 (KPI 4)*

KPI 4 Assumptions:

→ Returning on the same day still
counts as a repeat customer.



Step 2. Write in SQL.

```
SELECT date_format(mth, 'yyyy-MM') AS mth, COUNT(*)
FROM (
    SELECT DATE_TRUNC('month', purchased_at) as mth,
           shopper_id
    FROM transactions
    WHERE store_code = 5000128068369
    GROUP BY DATE_TRUNC('month', purchased_at), shopper_id
    HAVING COUNT(DISTINCT transaction_id) > 1
) x
GROUP BY mth;
```

Question: What does this table represent?

Answer: Pairs of shopper_ids and months, where the shopper shopped twice in the month.

Now use a subquery to group into and count.

Fact known per row:
shopper_id shopped twice in mth

Example 5

Growth Engagement Retention rate

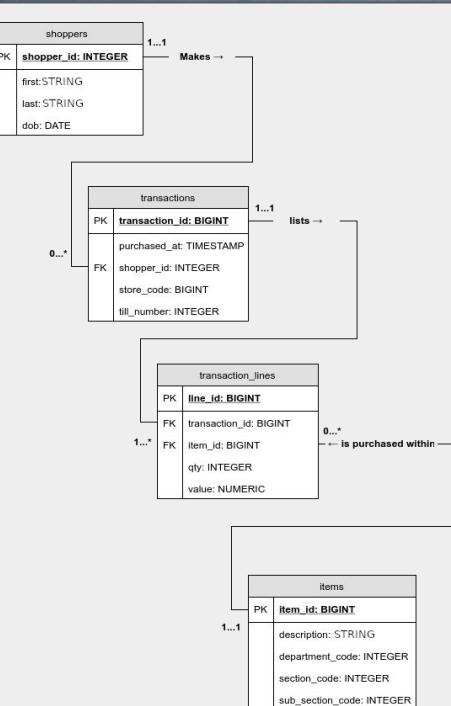
Step 1:

Write down a concrete measurable description of the KPI

*Engagement, measured as
→ Count of repeat customers per
month for store
5000128068369 (KPI 4)*

KPI 4 Assumptions:

→ *Returning is defined as returning
on a separate day.*



Step 2. Write in SQL.

```
SELECT date_format(mth, 'yyyy-MM') AS mth, COUNT(*)  
FROM (  
    SELECT DATE_TRUNC('month', purchased_at) as mth,  
          shopper_id  
    FROM transactions  
    WHERE store_code = 5000128068369  
    GROUP BY DATE_TRUNC('month', purchased_at), shopper_id  
    HAVING COUNT(DISTINCT transaction_id) > 1  
) x  
GROUP BY mth;
```

Code from Example 4

Your turn 2

Example 5

Growth Engagement Retention rate

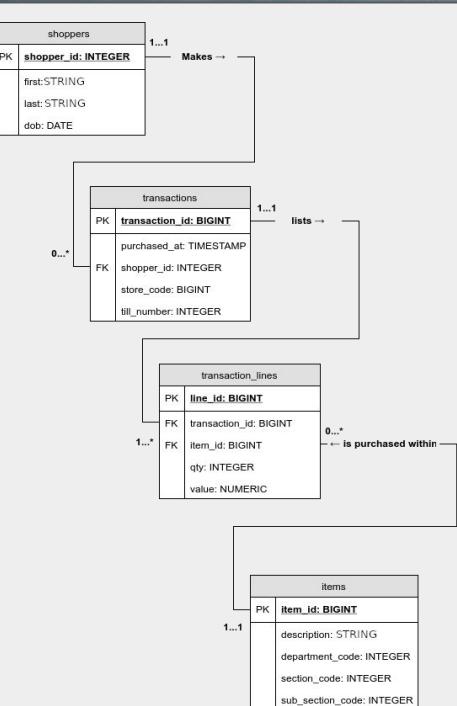
Step 1:

Write down a concrete measurable description of the KPI

Engagement, measured as
→ Count of repeat customers per month for store 5000128068369 (KPI 5)

KPI 5 Assumptions:

→ Returning is defined as returning on a separate day.



Step 2. Write in SQL.

```
SELECT date_format(mth, 'yyyy-MM') AS mth, COUNT(*)  
FROM (  
    SELECT DATE_TRUNC('month', purchased_at) as mth,  
          shopper_id  
     FROM transactions  
    WHERE store_code = 5000128068369  
    GROUP BY DATE_TRUNC('month', purchased_at), shopper_id  
    HAVING COUNT(DISTINCT purchased_at::DATE) > 1  
) x  
GROUP BY mth;
```

Q. How to encode the notion of "repeat on different days"

A1. Count of distinct purchased_at date (as a DATE) > 1

A2.

`MAX(purchased_at::DATE) - MIN(purchased_at::DATE) > 0`

Example 6

Growth
engagement
attention rate

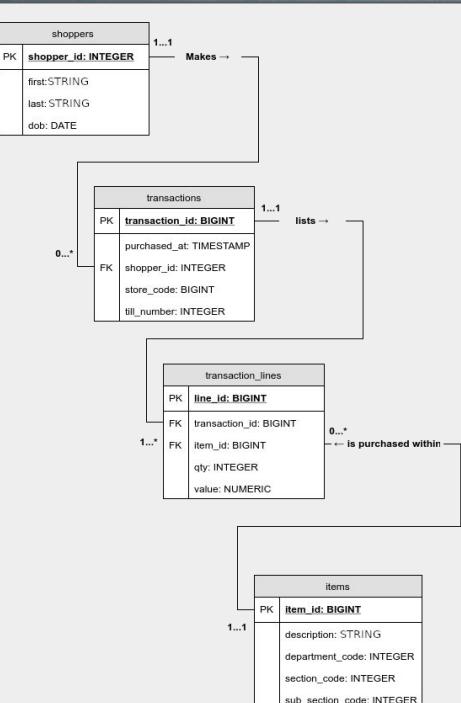
Step 1:

Write down a concrete measurable description of the KPI

*Engagement, measured as
→ Count of new customers per
month
(KPI 6)*

KPI 6 Assumptions:

→ Customers are considered new if they have never been seen before.



Step 2. Write in SQL.

Your turn 3

Example 6

Growth Engagement Retention rate

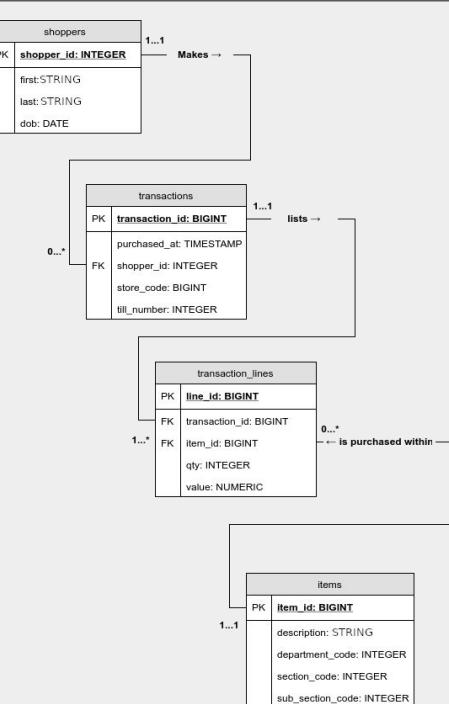
Step 1:

Write down a concrete measurable description of the KPI

*Engagement, measured as
→ Count of new customers per
month
(KPI 6)*

KPI 6 Assumptions:

→ Customers are considered new if they have never been seen before.



Step 2. Write in SQL.

```
SELECT shopper_id,
       DATE_TRUNC( 'month', MIN(purchased_at) ) as first_month
  FROM transactions
 GROUP BY shopper_id
```

Q. How to identify "new customers" in SQL?

A. If their MIN(purchased_at) is within the month.

Plan:

First compute a table of (shopper_id, months) representing shoppers and their first month

Then count the number of rows per month.

Example 6

Growth Engagement Retention rate

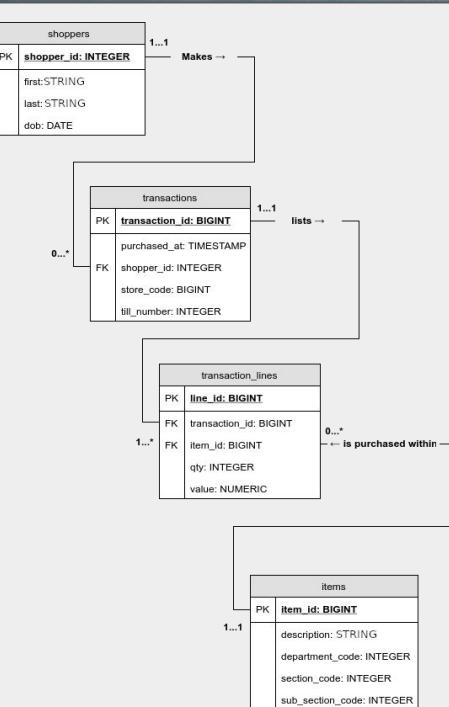
Step 1:

Write down a concrete measurable description of the KPI

*Engagement, measured as
→ Count of new customers per
month
(KPI 6)*

KPI 6 Assumptions:

→ Customers are considered new if
they have never been seen
before.



Step 2. Write in SQL.

```
SELECT date_format(first_month, 'yyyy-MM') as mth, COUNT(*) as ct
FROM (
    SELECT shopper_id,
           DATE_TRUNC( 'month', MIN(purchased_at) ) as first_month
    FROM transactions
    GROUP BY shopper_id
) x
GROUP BY first_month
```

Q. How to identify "new customers" in SQL?

A. If their MIN(purchased_at) is within the month.

Plan:

First compute a table of (shopper_id, months) representing shoppers and their first month

Then count the number of rows per month.

Growth Engagement Retention rate

Ok, so returning customers and new customers are two measures of engagement.

A more sophisticated approach is via **cohort retention analysis**.



Example

Active Users:

Jan	Feb	Mar	Apr	May
-----	-----	-----	-----	-----

180	195	200	190	180
-----	-----	-----	-----	-----

Returning customers

50	30	20	0	0
----	----	----	---	---

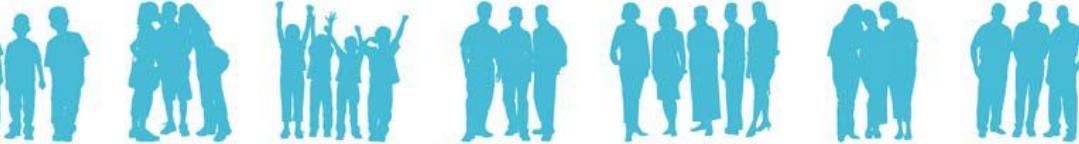
New customers

130	165	180	190	180
-----	-----	-----	-----	-----

Significant problem for most businesses!

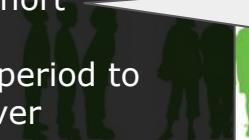
Retention / churn is a very common engagement KPI.

Growth Engagement Retention rate



Cohort Retention Analysis:

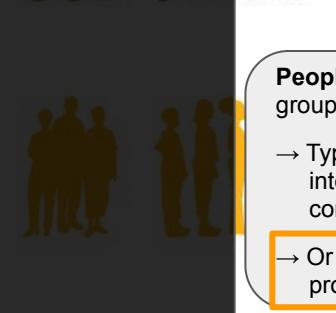
→ Define a cohort



Cohort:

- Group of people with a shared **temporal characteristic**
- Group of products with a shared **temporal characteristic**

→ Define the period to measure over



People will be assigned to each group based on a characteristic.

- Typically this is the first interaction date with the company.
- Or their first interaction with a product following a product launch

App Launched ↓ % Active users after App Launches →

Cohort	Users	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
Jan 25	1,098	100%	33.9%	23.5%	18.7%	15.9%	16.3%	14.2%	14.5%

Products will be assigned to each group based on a characteristic.

- Product launch date
- Advert campaign launch date
- Email promotion inclusion date



Growth Engagement Retention rate



Cohort Retention Analysis:

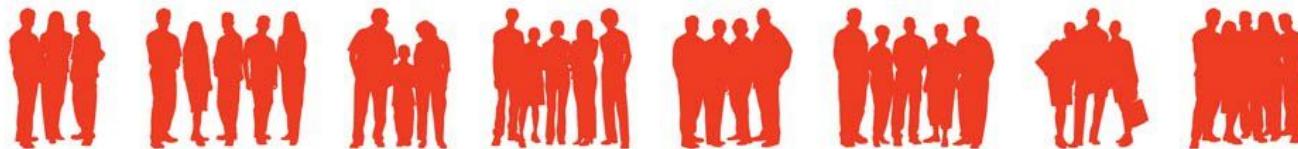
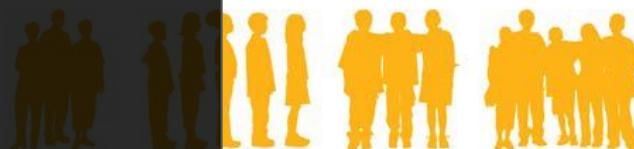
→ Define a cohort

→ Define the period to measure over

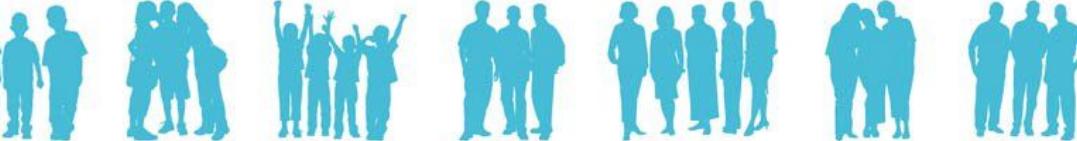
i.e. **daily, hourly, monthly....**

App Launched ↓ % Active users after App Launches →

Cohort	Users	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
Jan 25	1,098	100%	33.9%	23.5%	18.7%	15.9%	16.3%	14.2%	14.5%



Growth Engagement Retention rate

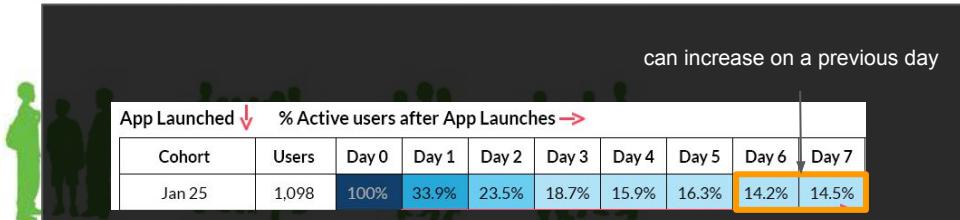


Cohort Retention Analysis:

→ Define a cohort



→ Define the period to measure over



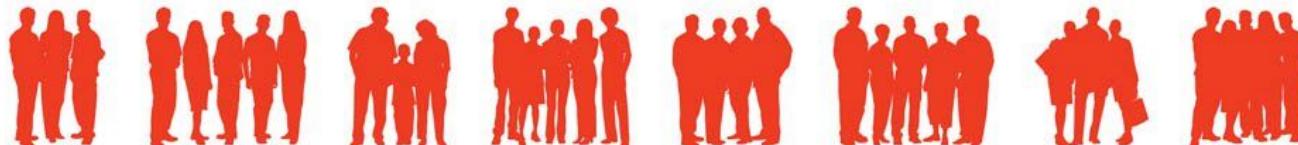
→ Select what to measure per cohort, per day.

For retention:

- **% active users**
- **% non-lapsed**
- ...

For products:

- % department sales
- % returns
- ...



Growth Engagement Retention rate



Cohorts performance per time period is then plotted against each other **relative to the initial value of the temporal characteristic.**

Cohort Retention Analysis:

→ Define a cohort

→ Define the period to measure over

→ Select what to measure per cohort, per day.

Cohort:

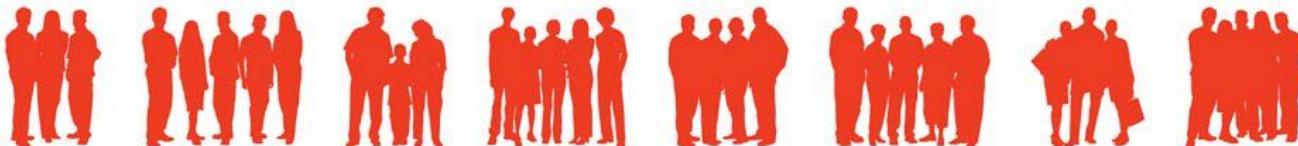
→ Group of people with a shared **temporal characteristic:**
their first interaction with a product following its launch

i.e. daily

For retention:
→ % active users

App Launched ↓ % Active users after App Launches →

Cohort	Users	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9	Day 10
Jan 25	1,098	100%	33.9%	23.5%	18.7%	15.9%	16.3%	14.2%	14.5%			12.1%
Jan 26	1,358	100%	31.1%	18.6%	14.3%	16.0%	14.9%	13.2%	12.9%			
Jan 27	1,257	100%	27.2%	19.6%	14.5%	12.9%	13.4%	13.0%	10.8%			11.4%
Jan 28	1,587	100%	26.6%	17.9%	14.6%	14.8%	14.9%	13.7%	11.9%			
Jan 29	1,758	100%	26.2%	20.4%	16.9%	14.3%	12.7%	12.5%				
Jan 30	1,624	100%	26.4%	18.1%	13.7%	15.4%	11.8%					
Jan 31	1,541	100%	23.9%	19.6%	15.0%	14.8%						
Feb 01	868	100%	24.7%	16.9%	15.8%							
Feb 02	1,143	Retention over product lifetime	18.5%									
Feb 03	1,253											
All Users	13,487	100%	27.0%	19.2%	15.4%	14.9%	14.0%	13.3%	12.5%	13.1%	12.2%	12.1%



Growth
Engagement
Retention rate



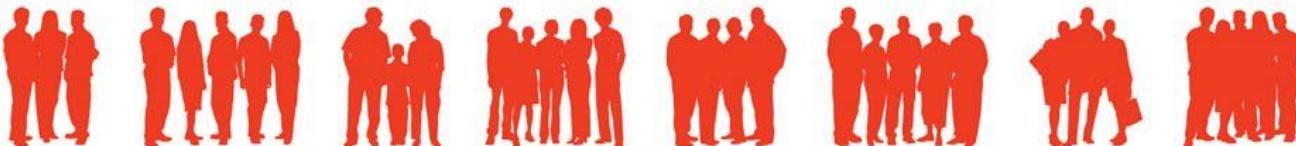
Cohorts performance per time period is then plotted against each other **relative to the initial value of the temporal characteristic**.

Great. So what next?

How to do it in SQL of course...

App Launched ↓ % Active users after App Launches →

Cohort	Users	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9	Day 10
Jan 25	1,098	100%	33.9%	23.5%	18.7%	15.9%	16.3%	14.2%	14.5%	Retention over user lifetime	12.1%	
Jan 26	1,358	100%	31.1%	18.6%	14.3%	16.0%	14.9%	13.2%	12.9%			
Jan 27	1,257	100%	27.2%	19.6%	14.5%	12.9%	13.4%	13.0%	10.8%	11.4%		
Jan 28	1,587	100%	26.6%	17.9%	14.6%	14.8%	14.9%	13.7%	11.9%			
Jan 29	1,758	100%	26.2%	20.4%	16.9%	14.3%	12.7%	12.5%				
Jan 30	1,624	100%	26.4%	18.1%	13.7%	15.4%	11.8%					
Jan 31	1,541	100%	23.9%	19.6%	15.0%	14.8%						
Feb 01	868	100%	24.7%	16.9%	15.8%							
Feb 02	1,143	Retention over product lifetime	18.5%									
Feb 03	1,253											
All Users	13,487	100%	27.0%	19.2%	15.4%	14.9%	14.0%	13.3%	12.5%	13.1%	12.2%	12.1%



Growth
Engagement
Retention rate



Cohort	Users	% Active users after App Launches →																													
		Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9																				
Jan 25	1,098	100%	33.9%	23.5%	18.7%	15.9%	16.3%	14.2%	14.5%	Retention over user lifetime																					
Jan 26	1,358	100%	31.1%	18.6%	14.3%	16.0%	14.9%	13.2%	12.9%																						
Jan 27	1,257	100%	27.2%	19.6%	14.5%	12.9%	13.4%	13.0%	10.8%	11.4%																					
Jan 28	1,587	100%	26.6%	17.9%	14.6%	14.8%	14.9%	13.7%	11.9%																						
Jan 29	1,758	100%	26.2%	20.4%	16.9%	14.3%	12.7%	12.5%																							
Jan 30	1,624	100%	26.4%	18.1%	13.7%	15.4%	11.8%																								
Jan 31	1,541	100%	23.9%	19.6%	15.0%	14.8%																									
Feb 01	868	100%	24.7%	16.9%	15.8%																										
Feb 02	1,143	Retention over product lifetime		38.5%																											
Feb 03	1,253																														
All Users	13,487	100%	27.0%	19.2%	15.4%	14.9%	14.0%	13.3%	12.5%	13.1%	12.2%	12.1%																			

This is a more complex query.

COMMON STRUCTURE. A table of data.

Could create a table like in the picture.

BUT WHAT HAPPENS TOMORROW? Add a column?

New table per day?



Growth
Engagement
Retention rate

Expanding Tables of Data

This is a more complex query.

COMMON STRUCTURE. A table of data.

Data in a table format is represented in a RBMS as tuples of:

(row_val, col_val, cell_val)

App Launched ↓ % Active users after App Launches →

Cohort	Users	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9	Day 10
Jan 25	1,098	100%	33.9%	23.5%	18.7%	15.9%	16.3%	14.2%	14.5%	Retention over user lifetime		12.1%
Jan 26	1,358	100%	31.1%	18.6%	14.3%	16.0%	14.9%	13.2%	12.9%			
Jan 27	1,257	100%	27.2%	19.6%	14.5%	12.9%	13.4%	13.0%	10.8%	11.4%		
Jan 28	1,587	100%	26.6%	17.9%	14.6%	14.8%	14.9%	13.7%	11.9%			
Jan 29	1,758	100%	26.2%	20.4%	16.9%	14.3%	12.7%	12.5%				
Jan 30	1,624	100%	26.4%	18.1%	13.7%	15.4%	11.8%					
Jan 31	1,541	100%	23.9%	19.6%	15.0%	14.8%						
Feb 01	868	100%	24.7%	16.9%	15.8%							
Feb 02	1,143	Retention over product lifetime		18.5%								
Feb 03	1,253											
All Users	13,487	100%	27.0%	19.2%	15.4%	14.9%	14.0%	13.3%	12.5%	13.1%	12.2%	12.1%

(row_val, col_val, cell_val)

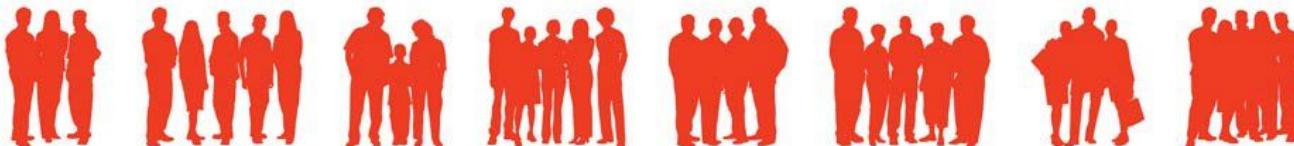
('Jan 25', 'Users', 1098)

('Jan 25', 'Day 0', 100.0)

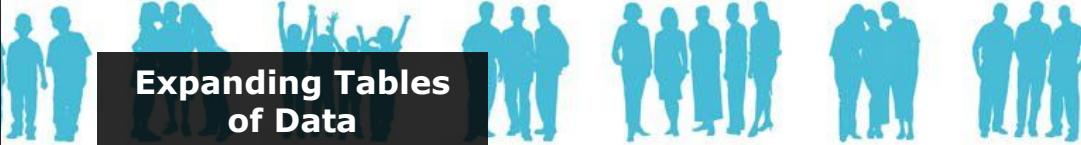
('Jan 25', 'Day 1', 33.9)

('Jan 29', 'Day 5', 12.7)

('All Users', 'Day 10', 12.1)



Growth
Engagement
Retention rate



Expanding Tables of Data

This is a more complex query.

COMMON STRUCTURE. A table of data.

Data in a table format is represented in a RBMS as tuples of:

(row_val, col_val, cell_val)

Exercise: Translate the following tuples into a table:

row_val, col_val, cell_val

'Jack', 'spend', 10.25

'John', 'spend', 6.25

'John', 'visits', 4

'Jack', 'visits', 6

Your turn 4

(row_val, col_val, cell_val)

('Jan 25', 'Users', 1098)

('Jan 25', 'Day 0', 100.0)

('Jan 25', 'Day 1', 33.9)

:

:

('Jan 29', 'Day 5', 12.7)

:

:

('All Users', 'Day 10', 12.1)



Growth
Engagement
Retention rate



This is a more complex query.

COMMON STRUCTURE. A table of data.

Data in a table format is represented in a RBMS as tuples of:

(row_val, col_val, cell_val)

Exercise: Translate the following tuples into a table:

row_val, col_val, cell_val

'Jack', 'spend', 10.25

'John', 'spend', 6.25

'John', 'visits', 4

'Jack', 'visits', 6

	spend	visits
Jack	10.25	6
John	6.25	4

(row_val, col_val, cell_val)

('Jan 25', 'Users', 1098)

('Jan 25', 'Day 0', 100.0)

('Jan 25', 'Day 1', 33.9)

:

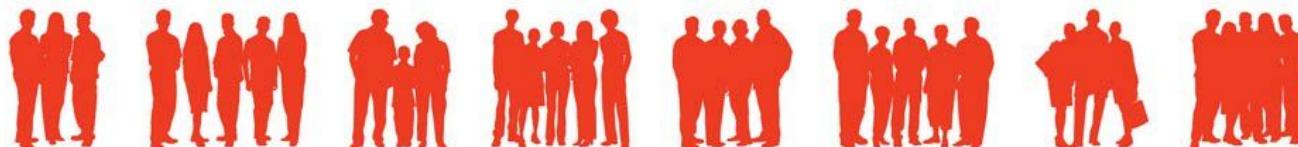
:

('Jan 29', 'Day 5', 12.7)

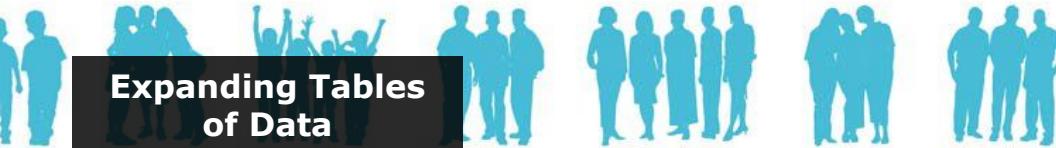
:

:

('All Users', 'Day 10', 12.1)



Growth
Engagement
Retention rate



This is a more complex query.

COMMON STRUCTURE. A table of data.

Data in a table format is represented in a RBMS as tuples of:

(row_val, col_val, cell_val)

Exercise: Translate the following table into tuples:

row_val, col_val, cell_val

Your turn 5

	cost (£)	units sold
iphone	800	157
nokia	255	300

(row_val, col_val, cell_val)

('Jan 25', 'Users', 1098)

('Jan 25', 'Day 0', 100.0)

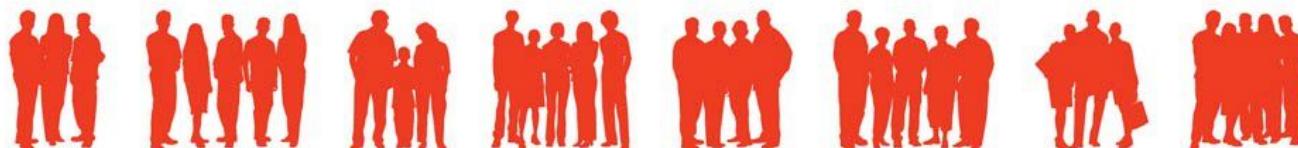
('Jan 25', 'Day 1', 33.9)

:

('Jan 29', 'Day 5', 12.7)

:

('All Users', 'Day 10', 12.1)



Growth
Engagement
Retention rate

Expanding Tables of Data

This is a more complex query.

COMMON STRUCTURE. A table of data.

Data in a table format is represented in a RBMS as tuples of:

(row_val, col_val, cell_val)

Exercise: Translate the following table into tuples:

row_val, col_val, cell_val

'iphone', 'cost (£)', 800
'iphone', 'units sold', 157
'nokia', 'cost (£)', 255
'nokia', 'units sold', 300

	cost (£)	units sold
iphone	800	157
nokia	255	300

(row_val, col_val, cell_val)

('Jan 25', 'Users', 1098)

('Jan 25', 'Day 0', 100.0)

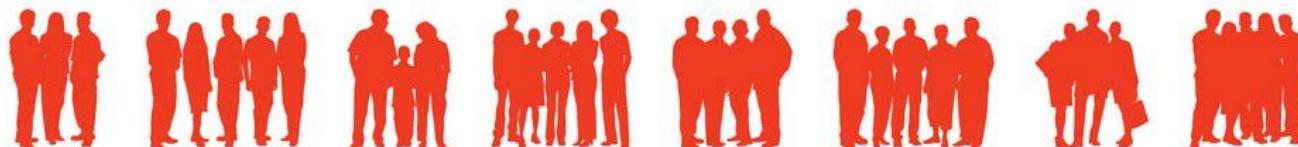
('Jan 25', 'Day 1', 33.9)

:

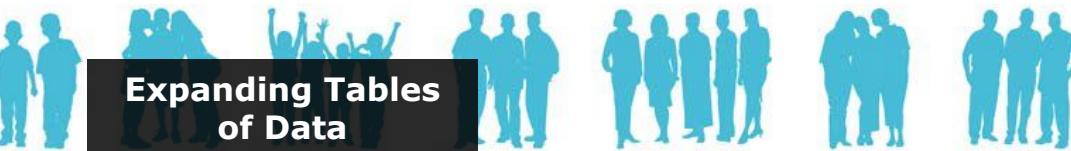
('Jan 29', 'Day 5', 12.7)

:

('All Users', 'Day 10', 12.1)



Growth
Engagement
Retention rate



This is a more complex query.

COMMON STRUCTURE. A table of data.

Data in a table format is represented in a RBMS as tuples of:

(row_val, col_val, cell_val)

Cohort	Users	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9	Day 10
Jan 25	1,098	100%	33.9%	23.5%	18.7%	15.9%	16.3%	14.2%	14.5%	Retention over user lifetime		12.1%
Jan 26	1,358	100%	31.1%	18.6%	14.3%	16.0%	14.9%	13.2%	12.9%			
All Users	13,487	100%	27.0%	19.2%	15.4%	14.9%	14.0%	13.3%	12.5%	13.1%	12.2%	12.1%

This can be easily visualized in Tableau from a tuple representation.

(I'll show you how later today)

(row_val, col_val, cell_val)

('Jan 25', 'Users', 1098)

('Jan 25', 'Day 0', 100.0)

('Jan 25', 'Day 1', 33.9)

('Jan 29', 'Day 5', 12.7)

('All Users', 'Day 10', 12.1)



Growth Engagement Retention rate

Expanding Tables of Data

OK, so we'll store the table in:

(**row_val**, **col_val**, **cell_val**)

format. **How?**

App Launched ↓ % Active users after App Launches →

Cohort	Users	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9	Day 10
Jan 25	1,098	100%	33.9%	23.5%	18.7%	15.9%	16.3%	14.2%	14.5%	Retention over user lifetime		12.1%
Jan 26	1,358	100%	31.1%	18.6%	14.3%	16.0%	14.9%	13.2%	12.9%			
Jan 27	1,257	100%	27.2%	19.6%	14.5%	12.9%	13.4%	13.0%	10.8%	11.4%		
Jan 28	1,587	100%	26.6%	17.9%	14.6%	14.8%	14.9%	13.7%	11.9%			
Jan 29	1,758	100%	26.2%	20.4%	16.9%	14.3%	12.7%	12.5%				
Jan 30	1,624	100%	26.4%	18.1%	13.7%	15.4%	11.8%					
Jan 31	1,541	100%	23.9%	19.6%	15.0%	14.8%						
Feb 01	868	100%	24.7%	16.9%	15.8%							
Feb 02	1,143	Retention over product lifetime		18.5%								
Feb 03	1,253											
All Users	13,487	100%	27.0%	19.2%	15.4%	14.9%	14.0%	13.3%	12.5%	13.1%	12.2%	12.1%

Multiple steps:

- 1) Assign customers to cohorts.
- 2) Count the number of customers active in each subsequent period.
- 3) Convert the counts to percents
- 4) Count the number of users per cohort. Add column to table.
- 5) Compute "All Users" row. Add row to table.

Growth Engagement Retention rate

Expanding Tables of Data

OK, so we'll store the table in:

(row_val, col_val, cell_val)

format. **How?**

cohort period value
(Jan 25, Day 0, 100%)

App Launched ↓ % Active users after App Launches →

Cohort	Users	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9	Day 10
Jan 25	1,098	100%	33.9%	23.5%	18.7%	15.9%	16.3%	14.2%	14.5%	Retention over user lifetime		12.1%
Jan 26	1,358	100%	31.1%	18.6%	14.3%	16.0%	14.9%	13.2%	12.9%			
Jan 27	1,257	100%	27.2%	19.6%	14.5%	12.9%	13.4%	13.0%	10.8%	11.4%		
Jan 28	1,587	100%	26.6%	17.9%	14.6%	14.8%	14.9%	13.7%	11.9%			
Jan 29	1,758	100%	26.2%	20.4%	16.9%	14.3%	12.7%	12.5%				
Jan 30	1,624	100%	26.4%	18.1%	13.7%	15.4%	11.8%					
Jan 31	1,541	100%	23.9%	19.6%	15.0%	14.8%						
Feb 01	868	100%	24.7%	16.9%	15.8%							
Feb 02	1,143	Retention over product lifetime		18.5%								
Feb 03	1,253											
All Users	13,487	100%	27.0%	19.2%	15.4%	14.9%	14.0%	13.3%	12.5%	13.1%	12.2%	12.1%

Multiple steps:

- 1) Assign customers to cohorts.
- 2) Count the number of customers active in each subsequent period.
- 3) Convert the counts to percents
- 4) Count the number of users per cohort. Add column to table.
- 5) Compute "All Users" row. Add row to table.

Growth Engagement Retention rate

Expanding Tables of Data

App Launched ↓ % Active users after App Launches →

Cohort	Users	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9	Day 10
Jan 25	1,098	100%	33.9%	23.5%	18.7%	15.9%	16.3%	14.2%	14.5%	Retention over user lifetime		12.1%
Jan 26	1,358	100%	31.1%	18.6%	14.3%	16.0%	14.9%	13.2%	12.9%			
Jan 27	1,257	100%	27.2%	19.6%	14.5%	12.9%	13.4%	13.0%	10.8%	11.4%		
Jan 28	1,587	100%	26.6%	17.9%	14.6%	14.8%	14.9%	13.7%	11.9%			
Jan 29	1,758	100%	26.2%	20.4%	16.9%	14.3%	12.7%	12.5%				
Jan 30	1,624	100%	26.4%	18.1%	13.7%	15.4%	11.8%					
Jan 31	1,541	100%	23.9%	19.6%	15.0%	14.8%						
Feb 01	868	100%	24.7%	16.9%	15.8%							
Feb 02	1,143	Retention over product lifetime		18.5%								
Feb 03	1,253											
All Users	13,487	100%	27.0%	19.2%	15.4%	14.9%	14.0%	13.3%	12.5%	13.1%	12.2%	12.1%

Multiple steps:

- 1) Assign customers to cohorts.
- 2) Count the number of customers active in each subsequent period.
- 3) Convert the counts to percents
- 4) Count the number of users per cohort (actually will have already been done as part of 3). Add column to table.
- 5) Compute "All Users" row. Add row to table.

Growth Engagement Retention rate

Expanding Tables of Data

OK, so we'll store the table in:

(**row_val**, **col_val**, **cell_val**)

format. **How?**

App Launched ↓ % Active users after App Launches →

Cohort	Users	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9	Day 10
Jan 25	1,098	100%	33.9%	23.5%	18.7%	15.9%	16.3%	14.2%	14.5%	Retention over user lifetime		12.1%
Jan 26	1,358	100%	31.1%	18.6%	14.3%	16.0%	14.9%	13.2%	12.9%			
Jan 27	1,257	100%	27.2%	19.6%	14.5%	12.9%	13.4%	13.0%	10.8%	11.4%		
Jan 28	1,587	100%	26.6%	17.9%	14.6%	14.8%	14.9%	13.7%	11.9%			
Jan 29	1,758	100%	26.2%	20.4%	16.9%	14.3%	12.7%	12.5%				
Jan 30	1,624	100%	26.4%	18.1%	13.7%	15.4%	11.8%					
Jan 31	1,541	100%	23.9%	19.6%	15.0%	14.8%						
Feb 01	868	100%	24.7%	16.9%	15.8%							
Feb 02	1,143	Retention over product lifetime		18.5%								
Feb 03	1,253											
All Users	13,487	100%	27.0%	19.2%	15.4%	14.9%	14.0%	13.3%	12.5%	13.1%	12.2%	12.1%

Multiple steps:

- 1) Assign customers to cohorts.
- 2) Count the number of customers active in each subsequent period.
- 3) Convert the counts to percents
- 4) Count the number of users per cohort. Add column to table.
- 5) Compute "All Users" row. Add row to table.

Growth Engagement Retention rate

We're going to stop here to keep things simple in this lecture (cells are then all percentages)

Feel free to try Step 5 yourself.

Expanding Tables of Data

App Launched ↓ % Active users after App Launches →

Cohort	Users	Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9	Day 10
Jan 25	1,098	100%	33.9%	23.5%	18.7%	15.9%	16.3%	14.2%	14.5%	Retention over user lifetime		12.1%
Jan 26	1,358	100%	31.1%	18.6%	14.3%	16.0%	14.9%	13.2%	12.9%			
Jan 27	1,257	100%	27.2%	19.6%	14.5%	12.9%	13.4%	13.0%	10.8%	11.4%		
Jan 28	1,587	100%	26.6%	17.9%	14.6%	14.8%	14.9%	13.7%	11.9%			
Jan 29	1,758	100%	26.2%	20.4%	16.9%	14.3%	12.7%	12.5%				
Jan 30	1,624	100%	26.4%	18.1%	13.7%	15.4%	11.8%					
Jan 31	1,541	100%	23.9%	19.6%	15.0%	14.8%						
Feb 01	868	100%	24.7%	16.9%	15.8%							
Feb 02	1,143	Retention over product lifetime		18.5%								
Feb 03	1,253											
All Users	13,487	100%	27.0%	19.2%	15.4%	14.9%	14.0%	13.3%	12.5%	13.1%	12.2%	12.1%

Multiple steps:

- 1) Assign customers to cohorts.
- 2) Count the number of customers active in each subsequent period.
- 3) Convert the counts to percents
- 4) Count the number of users per cohort. Add column to table.
- 5) Compute "All Users" row. Add row to table.

Example 7

Cohort Retention Analysis: Supermarket customer retention

First, fix our definitions

Cohort Retention Analysis:

- Define a cohort
- Define the period to measure over
- Select what to measure per cohort, per period.

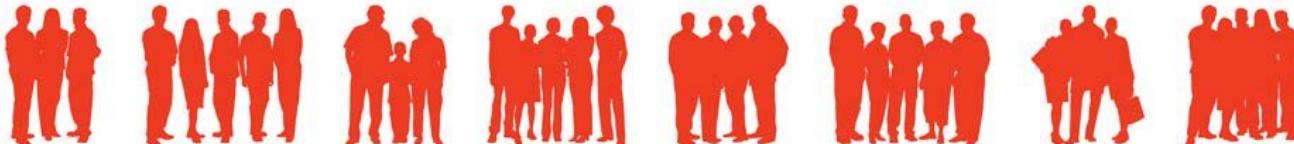
Cohort:

→ Group of people with a shared **temporal** characteristic:
their first purchase date

Monthly

For retention:

→ **% active users**



Example 7

Cohort Retention Analysis:

Supermarket customer retention

shoppers	
PK	shopper_id: INTEGER
first:	STRING
last:	STRING
dob:	DATE

transactions	
PK	transaction_id: BIGINT
purchased_at:	TIMESTAMP
shopper_id:	INTEGER
store_code:	BIGINT
till_number:	INTEGER

transaction_lines	
PK	line_id: BIGINT
transaction_id:	BIGINT
item_id:	BIGINT
qty:	INTEGER
value:	NUMERIC

items	
PK	item_id: BIGINT
description:	STRING
department_code:	INTEGER
section_code:	INTEGER
sub_section_code:	INTEGER

Cohort characteristic:
1st purchase date

Period to measure over:
Monthly

Measure, per cohort, per period:
% active users

Multiple steps:

- 1) Assign customers to cohorts.
- 2) Count the number of customers active in each subsequent period.
- 3) Convert the counts to percents
- 4) Count the number of users per cohort. Add column to table.

```
-- let's make a table of all shopper_ids and their cohort, denoted by their first month
```

Output table headings:
shopper_id (INT),
cohort_date (DATE)

We are first going to do this
in stages, creating
temporary tables.

Remember the syntax?

```
CREATE TABLE  
cohort_assignment AS  
.... <your select  
statement?
```

Your turn 6



Example 7

Cohort Retention Analysis:

Supermarket customer retention

shoppers	
PK	shopper_id: INTEGER
first:	STRING
last:	STRING
dob:	DATE

transactions	
PK	transaction_id: BIGINT
purchased_at:	TIMESTAMP
shopper_id:	INTEGER
store_code:	BIGINT
till_number:	INTEGER

transaction_lines	
PK	line_id: BIGINT
transaction_id:	BIGINT
item_id:	BIGINT
qty:	INTEGER
value:	Numeric

items	
PK	item_id: BIGINT
description:	STRING
department_code:	INTEGER
section_code:	INTEGER
sub_section_code:	INTEGER

Cohort characteristic:
1st purchase date

Period to measure over:
Monthly

Measure, per cohort, per period:
% active users

Multiple steps:

1) Assign customers to cohorts.

```
-- let's make a table of all shopper_ids and their cohort, denoted by their first month
```

```
CREATE TABLE cohort_assignment AS
SELECT shopper_id,
       DATE_TRUNC('month', MIN(purchased_at))::DATE as cohort_date
FROM transactions
GROUP BY shopper_id;
```

2) Count the number of customers active in each subsequent period.

3) Convert the counts to percents



4) Count the number of users per cohort. Add column to table.



Example 7

Cohort Retention Analysis:

Supermarket customer retention

shoppers	
PK	shopper_id: INTEGER
first:	STRING
last:	STRING
dob:	DATE

transactions	
PK	transaction_id: BIGINT
purchased_at:	TIMESTAMP
shopper_id:	INTEGER
store_code:	BIGINT
till_number:	INTEGER

transaction_lines	
PK	line_id: BIGINT
transaction_id:	BIGINT
item_id:	BIGINT
qty:	INTEGER
value:	Numeric

items	
PK	item_id: BIGINT
description:	STRING
department_code:	INTEGER
section_code:	INTEGER
sub_section_code:	INTEGER

Cohort characteristic:
1st purchase date

Period to measure over:
Monthly

Measure, per cohort, per period:
% active users

Multiple steps:

1) Assign customers to cohorts.

2) Count the number of customers active in each subsequent period.

3) Convert the counts to percents

4) Count the number of users per cohort. Add column to table.

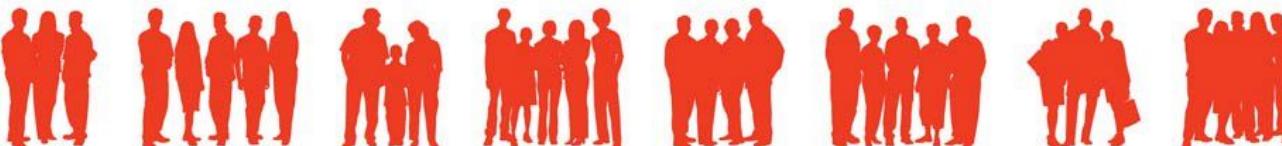
-- Logically this can be done by:
 1) Extending all transaction records with the associated shopper's cohort date.
 2) Compute a new column that transforms the records purchased_at date → relative period (month) since the shopper's cohort date.
 3) Putting records in buckets based on the cohort_date and relative_period and working how many shoppers were active

Output table headings:

cohort_date (DATE),
relative_period (INT),
active_ct (INT)

Your turn 7

App Launched	Users	% Active users after App Launches -->									
		Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9
Jan 25	1096	100%	33.9%	23.5%	18.7%	15.9%	16.3%	14.2%	14.5%	12.9%	12.1%
Jan 26	1358	100%	31.1%	18.6%	14.3%	16.0%	14.9%	13.2%	10.8%	11.4%	
Jan 27	1257	100%	27.2%	19.6%	14.5%	12.9%	13.4%	13.0%	10.8%		
Jan 28	1587	100%	26.6%	17.9%	14.6%	14.8%	14.9%	13.7%	11.9%		
Jan 29	1758	100%	26.2%	20.4%	16.9%	14.3%	12.7%	12.5%			
Jan 30	1624	100%	26.4%	18.1%	13.7%	15.4%	11.8%				
Jan 31	1541	100%	23.9%	19.6%	15.0%	14.8%					
Feb 01	868	100%	24.7%	16.9%	15.8%						
Feb 02	1143	Retention over product lifetime	18.5%								
Feb 03	1253	Retention over product lifetime									
All Users	13487	100%	27.0%	19.2%	15.4%	14.9%	14.0%	13.3%	12.5%	12.1%	12.1%



Example 7

Cohort Retention Analysis:

Supermarket customer retention

shoppers	
PK	shopper_id: INTEGER
first:	STRING
last:	STRING
dob:	DATE

transactions	
PK	transaction_id: BIGINT
purchased_at:	TIMESTAMP
shopper_id:	INTEGER
store_code:	BIGINT
till_number:	INTEGER

transaction_lines	
PK	line_id: BIGINT
transaction_id:	BIGINT
item_id:	BIGINT
qty:	INTEGER
value:	NUMERIC

items	
PK	item_id: BIGINT
description:	STRING
department_code:	INTEGER
section_code:	INTEGER
sub_section_code:	INTEGER

Cohort characteristic:
1st purchase date

Period to measure over:
Monthly

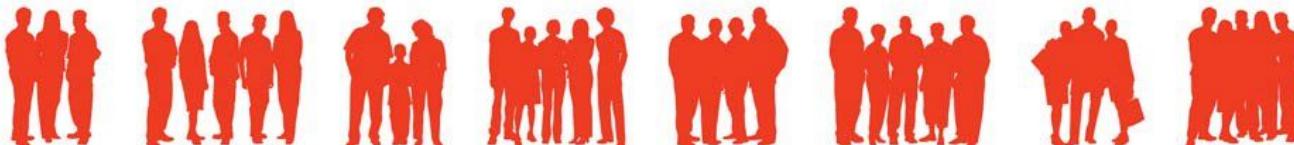
Measure, per cohort, per period:
% active users

Multiple steps:

- 1) Assign customers to cohorts.
- 2) Count the number of customers active in each subsequent period.
- 3) Convert the counts to percents
- 4) Count the number of users per cohort. Add column to table.

-- Logically this can be done by:
1) Extending all transaction records with the associated shopper's cohort date.

```
CREATE TABLE cohort_mth_cts AS
SELECT
  *
FROM transactions
JOIN cohort_assignment
USING (shopper_id);
```



Example 7

Cohort Retention Analysis:

Supermarket customer retention

shoppers	
PK	shopper_id: INTEGER
first:	STRING
last:	STRING
dob:	DATE

transactions	
PK	transaction_id: BIGINT
purchased_at:	TIMESTAMP
shopper_id:	INTEGER
store_code:	BIGINT
till_number:	INTEGER

transaction_lines	
PK	line_id: BIGINT
transaction_id:	BIGINT
item_id:	BIGINT
qty:	INTEGER
value:	NUMERIC

items	
PK	item_id: BIGINT
description:	STRING
department_code:	INTEGER
section_code:	INTEGER
sub_section_code:	INTEGER

Cohort characteristic:
1st purchase date

Period to measure over:
Monthly

Measure, per cohort, per period:
% active users

Multiple steps:

1) Assign customers to cohorts.

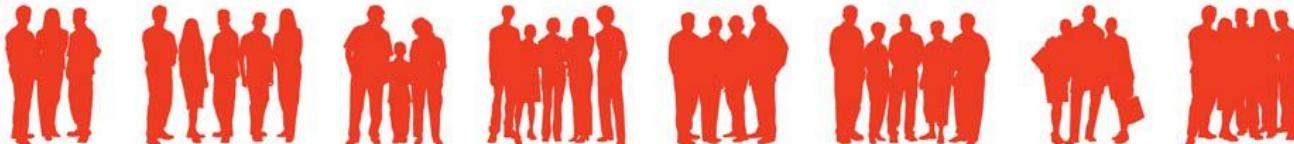
2) Count the number of customers active in each subsequent period.

3) Convert the counts to percents

4) Count the number of users per cohort. Add column to table.

-- Logically this can be done by:
 1) Extending all transaction records with the associated shopper's cohort date.
 2) Compute a new column that transforms the records purchased_at date → relative period (month) since the shopper's cohort date.

```
CREATE TABLE cohort_mth_cts AS
SELECT cohort_date,
       MONTHS_BETWEEN(
           DATE_TRUNC('month', purchased_at),
           DATE_TRUNC('month', cohort_date)
       ) as relative_period
FROM transactions
JOIN cohort_assignment
USING (shopper_id)
```



Example 7

Cohort Retention Analysis:

Supermarket customer
retention

shoppers	
PK	shopper_id: INTEGER
first:	STRING
last:	STRING
dob:	DATE

transactions	
PK	transaction_id: BIGINT
purchased_at:	TIMESTAMP
shopper_id:	INTEGER
store_code:	BIGINT
till_number:	INTEGER

transaction_lines	
PK	line_id: BIGINT
transaction_id:	BIGINT
item_id:	BIGINT
qty:	INTEGER
value:	Numeric

items	
PK	item_id: BIGINT
description:	STRING
department_code:	INTEGER
section_code:	INTEGER
sub_section_code:	INTEGER

Cohort characteristic:
1st purchase date

Period to measure over:
Monthly

Measure, per cohort, per period:
% active users

Multiple steps:

1) Assign customers to cohorts.

2) Count the number of
customers active in each
subsequent period.

3) Convert the counts to
percents

4) Count the number of users
per cohort. Add column to
table.

-- Logically this can be done by:
 1) Extending all transaction records with the associated shopper's cohort date.
 2) Compute a new column that transforms the records purchased_at date → relative period
 (month) since the shopper's cohort date.
 3) putting records in buckets based on the cohort_date and relative_period and working how many
 shoppers were active

```
CREATE TABLE cohort_mth_cts AS
SELECT cohort_date,
       MONTHS_BETWEEN(
           DATE_TRUNC('month', purchased_at),
           DATE_TRUNC('month', cohort_date)
       ) as relative_period,
       COUNT(DISTINCT shopper_id) as active_ct
FROM transactions
JOIN cohort_assignment
USING (shopper_id)
GROUP BY cohort_date, relative_period;
```



Example 7

Cohort Retention Analysis:

Supermarket customer retention

shoppers	
PK	shopper_id: INTEGER
first:	STRING
last:	STRING
dob:	DATE

transactions	
PK	transaction_id: BIGINT
purchased_at:	TIMESTAMP
shopper_id:	INTEGER
store_code:	BIGINT
till_number:	INTEGER

transaction_lines	
PK	line_id: BIGINT
transaction_id:	BIGINT
item_id:	BIGINT
qty:	INTEGER
value:	NUMERIC

items	
PK	item_id: BIGINT
description:	STRING
department_code:	INTEGER
section_code:	INTEGER
sub_section_code:	INTEGER

Cohort characteristic:
1st purchase date

Period to measure over:
Monthly

Measure, per cohort, per period:
% active users

Multiple steps:

1) Assign customers to cohorts.

2) Count the number of customers active in each subsequent period.

3) Convert the counts to percents

4) Count the number of users per cohort. Add column to table.

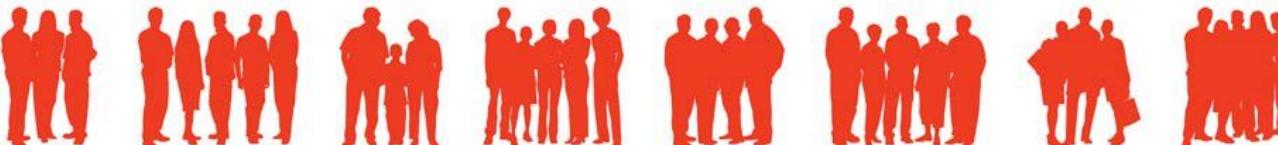
Demo. Let's see what we've created in Tableau!

```
CREATE TABLE cohort_mth_cts AS
SELECT cohort_date,
       MONTHS_BETWEEN(
           DATE_TRUNC('month', purchased),
           DATE_TRUNC('month', cohort_da)
       ) as relative_period,
       COUNT(DISTINCT shopper_id) as active
FROM transactions
JOIN cohort_assignment
USING (shopper_id)
GROUP BY cohort_date, relative_period;
```

cohort, period, active_val
(Jan 25, 1, 0.339)

NOTE: right now we have active counts, rather than %.

Cohort	Users	% Active users after App Launches -->																									
		Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9																
Jan 25	1,098	100%	23.9%	23.5%	18.7%	15.9%	16.3%	14.2%	14.5%	Retention over user lifetime																	
Jan 26	1,358	100%	31.1%	18.6%	14.3%	16.0%	14.9%	13.2%	12.9%																		
Jan 27	1,257	100%	27.2%	19.6%	14.5%	12.9%	13.4%	13.0%	10.8%	11.4%																	
Jan 28	1,587	100%	26.6%	17.9%	14.6%	14.8%	14.9%	13.7%	11.9%																		
Jan 29	1,758	100%	26.2%	20.4%	16.9%	14.3%	12.7%	12.5%																			
Jan 30	1,624	100%	26.4%	18.1%	13.7%	15.4%	11.8%																				
Jan 31	1,541	100%	23.9%	19.6%	15.0%	14.8%																					
Feb 01	868	100%	24.7%	16.9%	15.8%																						
Feb 02	1,143	Retention over product lifetime		18.5%																							
Feb 03	1,253																										
All Users	13,487	100%	27.0%	19.2%	15.4%	14.9%	14.0%	13.3%	12.5%	13.1%	12.2%																
											12.1%																



Example 7

Cohort Retention Analysis:

Supermarket customer retention

shoppers	
PK	shopper_id: INTEGER
first:	STRING
last:	STRING
dob:	DATE

transactions	
PK	transaction_id: BIGINT
purchased_at:	TIMESTAMP
shopper_id:	INTEGER
store_code:	BIGINT
till_number:	INTEGER

transaction_lines	
PK	line_id: BIGINT
transaction_id:	BIGINT
item_id:	BIGINT
qty:	INTEGER
value:	Numeric

items	
PK	item_id: BIGINT
description:	STRING
department_code:	INTEGER
section_code:	INTEGER
sub_section_code:	INTEGER

Cohort characteristic:
1st purchase date

Period to measure over:
Monthly

Measure, per cohort, per period:
% active users

Multiple steps:

- 1) Assign customers to cohorts.
- 2) Count the number of customers active in each subsequent period.
- 3) Convert the counts to percents
- 4) Count the number of users per cohort. Add column to table.

-- We could not normalize directly as we need to group by two different things
 numerator: cohort_date and relative_period
 denominator: cohort_date

CODE FROM PREVIOUS STEP

```
CREATE TABLE cohort_mth_cts AS
SELECT cohort_date,
       MONTHS_BETWEEN(
           DATE_TRUNC('month', purchased_at),
           DATE_TRUNC('month', cohort_date)
       ) as relative_period,
       COUNT(DISTINCT shopper_id) as active_ct
FROM transactions
JOIN cohort_assignment
USING (shopper_id)
GROUP BY cohort_date, relative_period;
```



Example 7

Cohort Retention Analysis:

Supermarket customer retention

shoppers	
PK	shopper_id: INTEGER
first:	STRING
last:	STRING
dob:	DATE

transactions	
PK	transaction_id: BIGINT
purchased_at:	TIMESTAMP
shopper_id:	INTEGER
store_code:	BIGINT
till_number:	INTEGER

transaction_lines	
PK	line_id: BIGINT
transaction_id:	BIGINT
item_id:	BIGINT
qty:	INTEGER
value:	Numeric

items	
PK	item_id: BIGINT
description:	STRING
department_code:	INTEGER
section_code:	INTEGER
sub_section_code:	INTEGER

Cohort characteristic:
1st purchase date

Period to measure over:
Monthly

Measure, per cohort, per period:
% active users

Multiple steps:

1) Assign customers to cohorts.

2) Count the number of customers active in each subsequent period.

3) Convert the counts to percents

4) Count the number of users per cohort. Add column to table.

-- We could not normalize directly as we need to group by two different things
 numerator: cohort_date and relative_period
 denominator: cohort_date

-- SOLUTION: Create a new table with just the denominator and JOIN it to the table cohort_mth_cts table, only keeping row for which the denominator logically matches with the numerator.

```
CREATE TABLE cohort_totals AS
SELECT cohort_date, COUNT(DISTINCT shopper_id) AS cohort_total
FROM cohort_assignment
GROUP BY cohort_date;
```



Example 7

Cohort Retention Analysis:

Supermarket customer retention

shoppers	
PK	shopper_id: INTEGER
first:	STRING
last:	STRING
dob:	DATE

transactions	
PK	transaction_id: BIGINT
purchased_at:	TIMESTAMP
shopper_id:	INTEGER
store_code:	BIGINT
till_number:	INTEGER

transaction_lines	
PK	line_id: BIGINT
transaction_id:	BIGINT
item_id:	BIGINT
qty:	INTEGER
value:	Numeric

items	
PK	item_id: BIGINT
description:	STRING
department_code:	INTEGER
section_code:	INTEGER
sub_section_code:	INTEGER

Cohort characteristic:
1st purchase date

Period to measure over:
Monthly

Measure, per cohort, per period:
% active users

Multiple steps:

1) Assign customers to cohorts.

2) Count the number of customers active in each subsequent period.

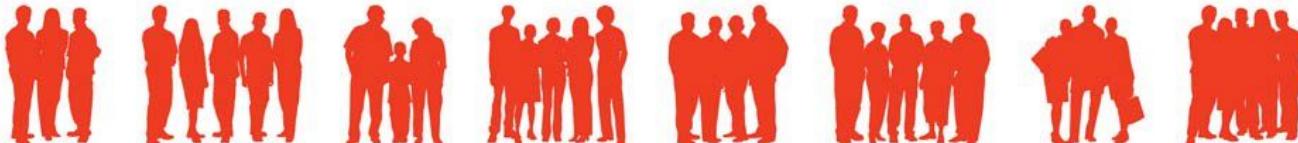
3) Convert the counts to percents

4) Count the number of users per cohort. Add column to table.

```
-- We could not normalize directly as we need to group by two different things
numerator: cohort_date and relative_period
denominator: cohort_date
-- SOLUTION: Create a new table with just the denominator and JOIN it to the table
cohort_mth_cts table, only keeping row for which the denominator logically
matches with the numerator.
```

```
CREATE TABLE cohort_totals AS
SELECT cohort_date, COUNT(DISTINCT shopper_id) AS cohort_total
FROM cohort_assignment
GROUP BY cohort_date;
```

```
CREATE TABLE cohort_mth_percent AS
SELECT cohort_date, relative_period, active_ct / cohort_total as active_percent
FROM cohort_mth_cts
JOIN cohort_totals
USING (cohort_date);
```



Example 7

Cohort Retention Analysis:

Supermarket customer retention

shoppers	
PK	shopper_id: INTEGER
first	STRING
last	STRING
dob	DATE

transactions	
PK	transaction_id: BIGINT
purchased_at	TIMESTAMP
shopper_id	INTEGER
store_code	BIGINT
till_number	INTEGER

transaction_lines	
PK	line_id: BIGINT
transaction_id	BIGINT
item_id	BIGINT
qty	INTEGER
value	NUMERIC

items	
PK	item_id: BIGINT
description	STRING
department_code	INTEGER
section_code	INTEGER
sub_section_code	INTEGER

Cohort characteristic:
1st purchase date

Period to measure over:
Monthly

Measure, per cohort, per period:
% active users

Multiple steps:

1) Assign customers to cohorts.

2) Count the number of customers active in each subsequent period.

3) Convert the counts to percents

4) Count the number of users per cohort. Add column to table.

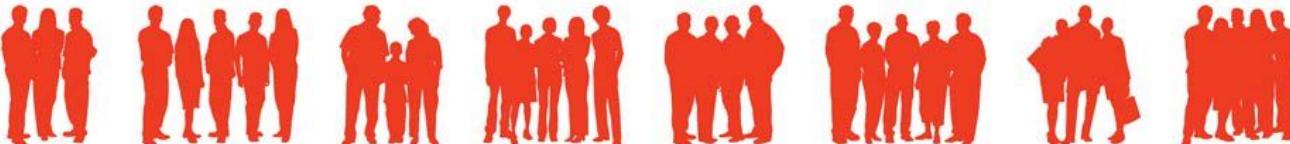
```
-- We could not normalize directly as we needed to divide the numerator: cohort_date and relative_period by the denominator: cohort_date
-- SOLUTION: Create a new table with just the relevant columns: cohort, period, active_val
--            cohort_mth_cts table, only keeping rows where the cohort matches with the numerator.
```

```
CREATE TABLE cohort_totals AS
SELECT cohort_date, COUNT(DISTINCT shopper_id) AS cohort_total
FROM cohort_assignment
GROUP BY cohort_date;
```

```
CREATE TABLE cohort_mth_percent AS
SELECT cohort_date, relative_period, active_ct / cohort_total as active_percent
FROM cohort_mth_cts
JOIN cohort_totals
USING (cohort_date);
```

Cohort	Users	% Active users after App Launches ->									
		Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9
Jan 25	1,099	100%	31.1%	23.5%	18.7%	15.9%	16.3%	14.2%	14.5%	12.9%	12.1%
Jan 26	1,358	100%	31.1%	18.6%	14.3%	16.0%	14.9%	13.2%	10.8%	11.4%	
Jan 27	1,257	100%	27.2%	19.6%	14.5%	12.9%	13.4%	13.0%			
Jan 28	1,587	100%	25.6%	17.9%	14.6%	14.8%	14.9%	13.7%	11.9%		
Jan 29	1,758	100%	26.2%	20.4%	16.9%	14.3%	12.7%	12.5%			
Jan 30	1,624	100%	26.4%	18.1%	13.7%	15.4%	11.8%				
Jan 31	1,541	100%	23.9%	19.6%	15.0%	14.8%					
Feb 01	868	100%	24.7%	16.9%	15.8%						
Feb 02	1,143	Retention over product lifetime									
Feb 03	1,253										
All Users	13,487	100%	27.0%	19.2%	15.4%	14.9%	14.0%	13.3%	12.5%	13.1%	12.2%

Your turn 8



Expanding table structure:

(row_val, col_val, cell_val)

Multiple steps:

- 1) Assign customers to cohorts.
- 2) Count the number of customers active in each subsequent period.
- 3) Convert the counts to percents
- 4) Count the number of users per cohort. Add column to table.

Adding column

To add a new **column** we need to add extra tuples.

One tuple per row value.

All tuples have same col_val, the name of the new column.

(row_id, new_col_id, cell_val)

Fixed values

Cohort	Users	% Active users after App Launches -->										Retention over user lifetime	Day 10
		Day 0	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9		
Jan 25	1,000	10%	31.9%	23.5%	18.7%	15.9%	16.3%	14.2%	14.5%	13.0%	11.4%	12.1%	
Jan 26	1,358	10%	31.1%	18.6%	14.3%	16.0%	14.9%	13.2%	12.9%				
Jan 27	1,257	10%	27.2%	19.6%	14.5%	12.9%	13.4%	13.0%	10.8%	11.4%			
Jan 28	1,587	10%	26.6%	17.9%	14.6%	14.8%	14.9%	13.7%	11.9%				
Jan 29	1,758	10%	26.2%	20.4%	16.9%	14.3%	12.7%	12.5%					
Jan 30	1,624	10%	26.4%	18.1%	13.7%	15.4%	11.8%						
Jan 31	1,541	10%	23.9%	19.6%	15.0%	14.8%							
Feb 01	868	10%	24.7%	16.9%	15.8%								
Feb 02	1,143	10%	18.5%										
Feb 03	1,253	10%											
All Users	10,401	10%	27.0%	19.2%	15.4%	14.9%	14.0%	13.3%	12.5%	13.1%	12.2%	12.1%	

row_id = cohort_date
col_id = 'total'
cell_val = total shoppers for the cohort_date

We've seen these values before....

Example 7

Expanding table structure:

(**row_val**, **col_val**, **cell_val**)

Column value will be fixed
to name of new column
being added.

Our Example Cont.

RECALL:

```
CREATE TABLE cohort_totals AS
SELECT cohort_date, COUNT(DISTINCT shopper_id) AS cohort_total
FROM cohort_assignment
GROUP BY cohort_date;
```

So the extra rows we want are the same as in cohort_totals with an extra fixed column...

```
SELECT
  FROM cohort_totals
```

- 1) Assign customers to cohorts.
- 2) Count the number of customers active in each subsequent period.
- 3) Convert the counts to percents
- 4) Count the number of users per cohort. Add column to table.

Example 7

Expanding table structure:

(**row_val**, **col_val**, **cell_val**)

Multiple steps:

- 1) Assign customers to cohorts.
- 2) Count the number of customers active in each subsequent period.
- 3) Convert the counts to percents
- 4) Count the number of users per cohort. Add column to table.

Column value will be fixed to name of new column being added.

Our Example Cont.

RECALL:

```
CREATE TABLE cohort_totals AS
SELECT cohort_date, COUNT(DISTINCT shopper_id) AS cohort_total
FROM cohort_assignment
GROUP BY cohort_date;
```

So the extra rows we want are the same as in cohort_totals with an extra fixed column...

```
SELECT cohort_date, 'total'::STRING, cohort_total
FROM cohort_totals
```

Example 7

Expanding table structure:

(**row_val**, **col_val**, **cell_val**)

Column value will be fixed to name of new column being added.

Multiple steps:

- 1) Assign customers to cohorts.
- 2) Count the number of customers active in each subsequent period.
- 3) Convert the counts to percents
- 4) Count the number of users per cohort. Add column to table.

Our Example Cont.

RECALL:

```
CREATE TABLE cohort_totals AS  
SELECT cohort_date, COUNT(DISTINCT shopper_id) AS cohort_total  
FROM cohort_assignment  
GROUP BY cohort_date;
```

So the extra rows we want are the same as in `cohort_totals` with an extra fixed column...

```
SELECT cohort_date, 'total', cohort_total  
FROM cohort_totals
```

Now how do we add them to our table `cohort_mth_percent` (represented in tuple format)?

Expanding table structure:

(row_val, col_val, cell_val)

Multiple steps:

1) Assign customers to cohorts.

2) Count the number of
customers active in each
subsequent period.

3) Convert the counts to
percents.

4) Count the number of users
per cohort. Add column to
table.

UNION

-- Two tables with the same columns can be merged using
-- the UNION command

UNION: Returns new table with all rows from both tables.
DUPLICATES are omitted.

UNION ALL: UNION, but duplicates kept.

EXCEPT: Returns rows in the first table that
do not exist in the second table.

Example 7

Expanding table structure:

(**row_val**, **col_val**, **cell_val**)

Multiple steps:

- 1) Assign customers to cohorts.
- 2) Count the number of customers active in each subsequent period.
- 3) Convert the counts to percents
- 4) Count the number of users per cohort. Add column to table.

Column value will be fixed to name of new column being added.

Our Example Cont.

RECALL:

```
CREATE TABLE cohort_totals AS
SELECT cohort_date, COUNT(DISTINCT shopper_id) AS cohort_total
FROM cohort_assignment
GROUP BY cohort_date;
```

So the extra rows we want are the same as in cohort_totals with an extra fixed column...

```
SELECT cohort_date AS row_id, 'total' AS col_id, cohort_total AS val
FROM cohort_totals
```

Now how do we add them to our table cohort_mth_percent (represented in tuple format)?

```
CREATE TABLE cohort_analysis AS
SELECT cohort_date as row_id, relative_period::STRING as col_id, active_percent as val
FROM cohort_mth_percent
UNION ALL
SELECT cohort_date AS row_id, 'total'::STRING AS col_id, cohort_total AS val
FROM cohort_totals;
```

Example 7

Expanding table structure:

(**row_val**, **col_val**, **cell_val**)

Multiple steps:

- 1) Assign customers to cohorts.
- 2) Count the number of customers active in each subsequent period.
- 3) Convert the counts to percents
- 4) Count the number of users per cohort. Add column to table.

Column value will be fixed to name of new column being added.

Our Example Cont.

RECALL:

```
CREATE TABLE cohort_totals AS  
SELECT cohort_date, COUNT(DISTINCT shopper_id) AS cohort_total  
FROM cohort_assignment  
GROUP BY cohort_date;
```

So the extra rows we want are the same as in

```
SELECT cohort_date AS row_id, 'total' AS col_id,  
cohort_total AS val  
FROM cohort_totals
```

UNION requires all column types to be the same in our relation.

In our **expanding table** our column header values used to be integers. Now we want to name a column header 'total'. All must be STRING.

Now how do we add them to our table cohort_mth_percent (represented in tuple format)?

```
CREATE TABLE cohort_analysis AS  
SELECT cohort_date as row_id, relative_period::STRING as col_id, active_percent as val  
FROM cohort_mth_percent  
UNION ALL  
SELECT cohort_date AS row_id, 'total'::STRING AS col_id, cohort_total AS val  
FROM cohort_totals;
```

Example 7

Expanding table structure:

(**row_val**, **col_val**, **cell_val**)

Column value will be fixed to name of new column being added.

Multiple steps:

- 1) Assign customers to cohorts.
- 2) Count the number of customers active in each subsequent period.
- 3) Convert the counts to percents
- 4) Count the number of users per cohort. Add column to table.

Our Example Cont.

RECALL:

```
CREATE TABLE cohort_totals AS  
SELECT cohort_date, COUNT(DISTINCT shopper_id) AS cohort_total  
FROM cohort_assignment  
GROUP BY cohort_date;
```

So the extra rows we want are the same as in

```
SELECT cohort_date AS row_id, 'total' AS col_id,  
cohort_total AS val  
FROM cohort_totals
```

UNION requires all column types to be the same in our relation.

In our **expanding table** our column header values used to be integers. Now we want to name a column header 'total'. All must be STRING.

Now how do we add them to our table cohort_mth_percent (represented in tuple format)?

```
CREATE TABLE cohort_analysis AS  
SELECT cohort_date as row_id, relative_period::STRING as col_id, active_percent as val  
FROM cohort_mth_percent  
UNION ALL  
SELECT cohort_date AS row_id, 'total'::STRING AS col_id, cohort_total AS val  
FROM cohort_totals;
```

column...

Let's recap.

Expanding table structure:
(row_val, col_val, cell_val)

Multiple steps:

1) Assign customers to cohorts.

2) Count the number of
customers active in each
subsequent period.

3) Convert the counts to
percents

4) Count the number of users
per cohort. Add column to
table.

```
CREATE TABLE cohort_assignment
AS
SELECT shopper_id,
       DATE_TRUNC('month',
MIN(purchased_at))::DATE AS cohort_date
FROM transactions
GROUP BY shopper_id;
```

```
CREATE TABLE cohort_mth_cts AS
SELECT
    cohort_date,
    MONTHS_BETWEEN(
        DATE_TRUNC('month',
                    purchased_at),
        DATE_TRUNC('month',
                    cohort_date)
    ) AS relative_period,
    COUNT(DISTINCT shopper_id)
        AS active_ct
FROM transactions
JOIN cohort_assignment
USING (shopper_id)
GROUP BY cohort_date,
relative_period;
```

```
CREATE TABLE cohort_totals AS
SELECT cohort_date,
COUNT(DISTINCT shopper_id) AS cohort_total
FROM cohort_assignment
GROUP BY cohort_date;
```

```
CREATE TABLE
cohort_mth_percent AS
SELECT cohort_date,
relative_period, active_ct /
cohort_total AS active_percent
FROM cohort_mth_cts
JOIN cohort_totals
USING (cohort_date)
```

```
CREATE TABLE cohort_analysis
AS
SELECT cohort_date AS row_id,
relative_period::STRING AS col_id,
active_percent AS val
FROM cohort_mth_percent
UNION ALL
SELECT cohort_date AS row_id,
'total'::STRING AS col_id,
cohort_total AS val
FROM cohort_totals
```

Let's recap.

Expanding table structure:
(row_val, col_val, cell_val)

Multiple steps:

1) Assign customers to cohorts.

2) Count the number of
customers active in each
subsequent period.

3) Convert the counts to
percents

4) Count the number of users
per cohort. Add column to
table.

```
CREATE TABLE cohort_assignment
AS
SELECT shopper_id,
       DATE_TRUNC('month',
MIN(purchased_at))::DATE AS cohort_date
FROM transactions
GROUP BY shopper_id;
```

```
CREATE TABLE cohort_mth_cts AS
SELECT
    cohort_date,
    MONTHS_BETWEEN(
        DATE_TRUNC('month',
                    purchased_at),
        DATE_TRUNC('month',
                    cohort_date)
    ) AS relative_period,
    COUNT(DISTINCT shopper_id) AS active_ct
FROM transactions
JOIN cohort_assignment
USING (shopper_id)
GROUP BY cohort_date,
relative_period;
```

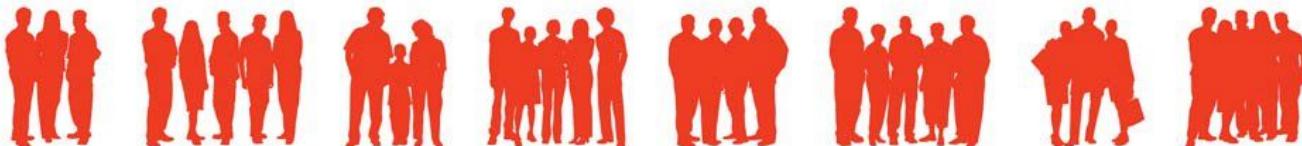
```
CREATE TABLE cohort_totals AS
SELECT cohort_date,
COUNT(DISTINCT shopper_id) AS cohort_total
FROM cohort_assignment
GROUP BY cohort_date;
```

```
CREATE TABLE
cohort_mth_percent AS
SELECT cohort_date,
relative_period, active_ct /
cohort_total AS active_percent
FROM cohort_mth_cts
JOIN cohort_totals
USING (cohort_date)
```

```
CREATE TABLE cohort_analysis
AS
SELECT cohort_date AS row_id,
relative_period::STRING AS col_id,
active_percent AS val
FROM cohort_mth_percent
UNION ALL
SELECT cohort_date AS row_id,
'total'::STRING AS col_id,
cohort_total AS val
FROM cohort_totals
```

Well, that was a lot of temporary tables.

So much for auto-updating (expanding tables)....



Let's recap.

Expanding table structure:
(row_val, col_val, cell_val)

Multiple steps:

1) Assign customers to cohorts.

2) Count the number of customers active in each subsequent period.

3) Convert the counts to percents

4) Count the number of users per cohort. Add column to table.

```
CREATE TABLE cohort_assignment
AS
SELECT shopper_id,
       DATE_TRUNC('month',
MIN(purchased_at))::DATE AS cohort_date
FROM transactions
GROUP BY shopper_id;
```

```
CREATE TABLE cohort_mth_cts AS
SELECT
    cohort_date,
    MONTHS_BETWEEN(
        DATE_TRUNC('month',
                    purchased_at),
        DATE_TRUNC('month',
                    cohort_date)
    ) AS relative_period,
    COUNT(DISTINCT shopper_id) AS active_ct
FROM transactions
JOIN cohort_assignment
USING (shopper_id)
GROUP BY cohort_date,
relative_period;
```

```
CREATE TABLE cohort_totals AS
SELECT cohort_date,
COUNT(DISTINCT shopper_id) AS cohort_total
FROM cohort_assignment
GROUP BY cohort_date;
```

```
CREATE TABLE
cohort_mth_percent AS
SELECT cohort_date,
relative_period, active_ct /
cohort_total AS active_percent
FROM cohort_mth_cts
JOIN cohort_totals
USING (cohort_date)
```

```
CREATE TABLE cohort_analysis
AS
SELECT cohort_date AS row_id,
relative_period::STRING AS col_id,
active_percent AS val
FROM cohort_mth_percent
UNION ALL
SELECT cohort_date AS row_id,
'total'::STRING AS col_id,
cohort_total AS val
FROM cohort_totals
```

Solution A: Subqueries.

→ Pretty unreadable

→ `cohort_totals` is used twice. Will have to repeat code.



Let's recap.

Expanding table structure:
(row_val, col_val, cell_val)

Multiple steps:

1) Assign customers to cohorts.

2) Count the number of customers active in each subsequent period.

3) Convert the counts to percents

4) Count the number of users per cohort. Add column to table.

```
CREATE TABLE cohort_assignment
AS
SELECT shopper_id,
       DATE_TRUNC('month',
MIN(purchased_at))::DATE AS cohort_date
FROM transactions
GROUP BY shopper_id;
```

```
CREATE TABLE cohort_mth_cts AS
SELECT
    cohort_date,
    MONTHS_BETWEEN(
        DATE_TRUNC('month',
                    purchased_at),
        DATE_TRUNC('month',
                    cohort_date)
    ) AS relative_period,
    COUNT(DISTINCT shopper_id) AS active_ct
FROM transactions
JOIN cohort_assignment
USING (shopper_id)
GROUP BY cohort_date,
relative_period;
```

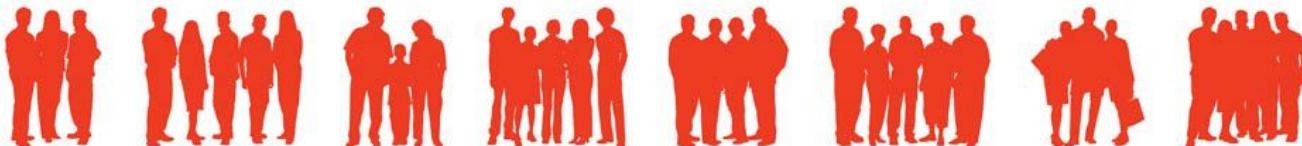
```
CREATE TABLE cohort_totals AS
SELECT cohort_date,
COUNT(DISTINCT shopper_id) AS cohort_total
FROM cohort_assignment
GROUP BY cohort_date;
```

```
CREATE TABLE
cohort_mth_percent AS
SELECT cohort_date,
relative_period, active_ct /
cohort_total AS active_percent
FROM cohort_mth_cts
JOIN cohort_totals
USING (cohort_date)
```

```
CREATE TABLE cohort_analysis
AS
SELECT cohort_date AS row_id,
relative_period::STRING AS col_id,
active_percent AS val
FROM cohort_mth_percent
UNION ALL
SELECT cohort_date AS row_id,
'total'::STRING AS col_id,
cohort_total AS val
FROM cohort_totals
```

Solution B: Common Table Expressions.

- Very useful.
- Simple.
- Let's learn them now!



Introducing **common table expressions**.

Allows the simple declaration of temporary tables that **exist for just that query**.

Keeps our logic as it was.

Conceptually separate tables.

ORDER MATTERS. CAN USE TABLES DEFINED ABOVE.

WITH

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
),
top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
  FROM orders
 WHERE region IN (SELECT region FROM top_regions)
 GROUP BY region, product;
```

Let's recap.

Expanding table structure:
(row_val, col_val, cell_val)

Multiple steps:

1) Assign customers to cohorts.

2) Count the number of customer's active each in each subsequent period.

3) Convert the counts to percents

4) Count the number of users per cohort. Add column to table.

```
CREATE TABLE cohort_assignment
AS
SELECT shopper_id,
       DATE_TRUNC('month',
MIN(purchased_at))::DATE AS cohort_date
FROM transactions
GROUP BY shopper_id;
```

```
CREATE TABLE cohort_mth_cts AS
SELECT
    cohort_date,
    MONTHS_BETWEEN(
        DATE_TRUNC('month',
                    purchased_at),
        DATE_TRUNC('month',
                    cohort_date)
    ) AS relative_period,
    COUNT(DISTINCT shopper_id)
        AS active_ct
FROM transactions
JOIN cohort_assignment
USING (shopper_id)
GROUP BY cohort_date,
relative_period;
```

```
CREATE TABLE cohort_totals AS
SELECT cohort_date,
COUNT(DISTINCT shopper_id) AS cohort_total
FROM cohort_assignment
GROUP BY cohort_date;
```

```
CREATE TABLE
cohort_mth_percent AS
SELECT cohort_date,
relative_period, active_ct /
cohort_total AS active_percent
FROM cohort_mth_cts
JOIN cohort_totals
USING (cohort_date)
```

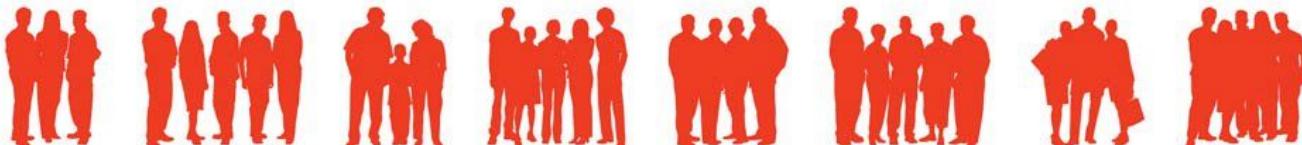
```
CREATE TABLE cohort_analysis
AS
SELECT cohort_date AS row_id,
relative_period::STRING AS col_id,
active_percent AS val
FROM cohort_mth_percent
UNION ALL
SELECT cohort_date AS row_id,
'total'::STRING AS col_id,
cohort_total AS val
FROM cohort_totals
```

Common Table Expression Syntax Example

```
WITH regional_sales AS (
    SELECT ... FROM ... WHERE ...
),
top_regions AS (
    SELECT ... FROM ...
)
SELECT ... ;
```

Example 7 Your turn 10

Using CTEs write the whole thing as one SQL query.



```
CREATE TABLE cohort_analytis_final AS
WITH    cohort_assignment AS (
            SELECT shopper_id,
                   DATE_TRUNC('month', MIN(purchased_at))::DATE as cohort_date
              FROM transactions
             GROUP BY shopper_id
        ),
        cohort_mth_cts AS (
            SELECT cohort_date,
                   MONTHS_BETWEEN( DATE_TRUNC('month', purchased_at), DATE_TRUNC('month', cohort_date) )
                           as relative_period,
                   COUNT(DISTINCT shopper_id) as active_ct
              FROM transactions
             JOIN cohort_assignment
               USING (shopper_id)
             GROUP BY cohort_date, relative_period
        ),
        cohort_totals AS (
            SELECT cohort_date, COUNT(DISTINCT shopper_id) AS cohort_total
              FROM cohort_assignment
             GROUP BY cohort_date
        ),
        cohort_mth_percent AS (
            SELECT cohort_date, relative_period, active_ct / cohort_total as active_percent
              FROM cohort_mth_cts
             JOIN cohort_totals
               USING (cohort_date)
        )
    SELECT cohort_date as row_id, relative_period::STRING as col_id, active_percent as val
      FROM cohort_mth_percent
     UNION ALL
    SELECT cohort_date AS row_id, 'total'::STRING AS col_id, cohort_total AS val
      FROM cohort_totals;
```

Yep. Simple. But
complicated.

Just takes practice and
step-by-step logical thought.

What we've done this
week:

- 7 Examples of KPIs in SQL
- Cohort Analysis
- UNION
- Expanding tables
- Visualizing expanding tables in Tableau

