# Session 9

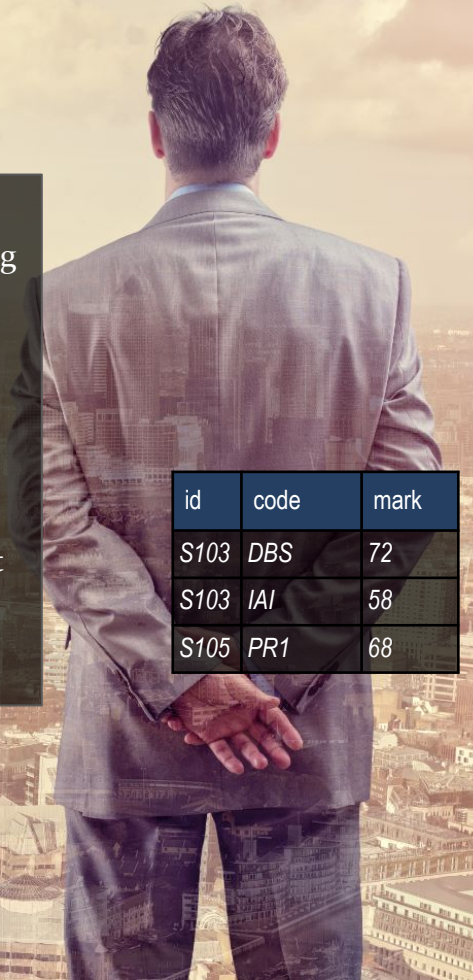
## Relational Databases (SQL III)


NYLAB: *Data at Scale*

# 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**

# 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
```

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 |

**Take home message:**

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

# 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.**

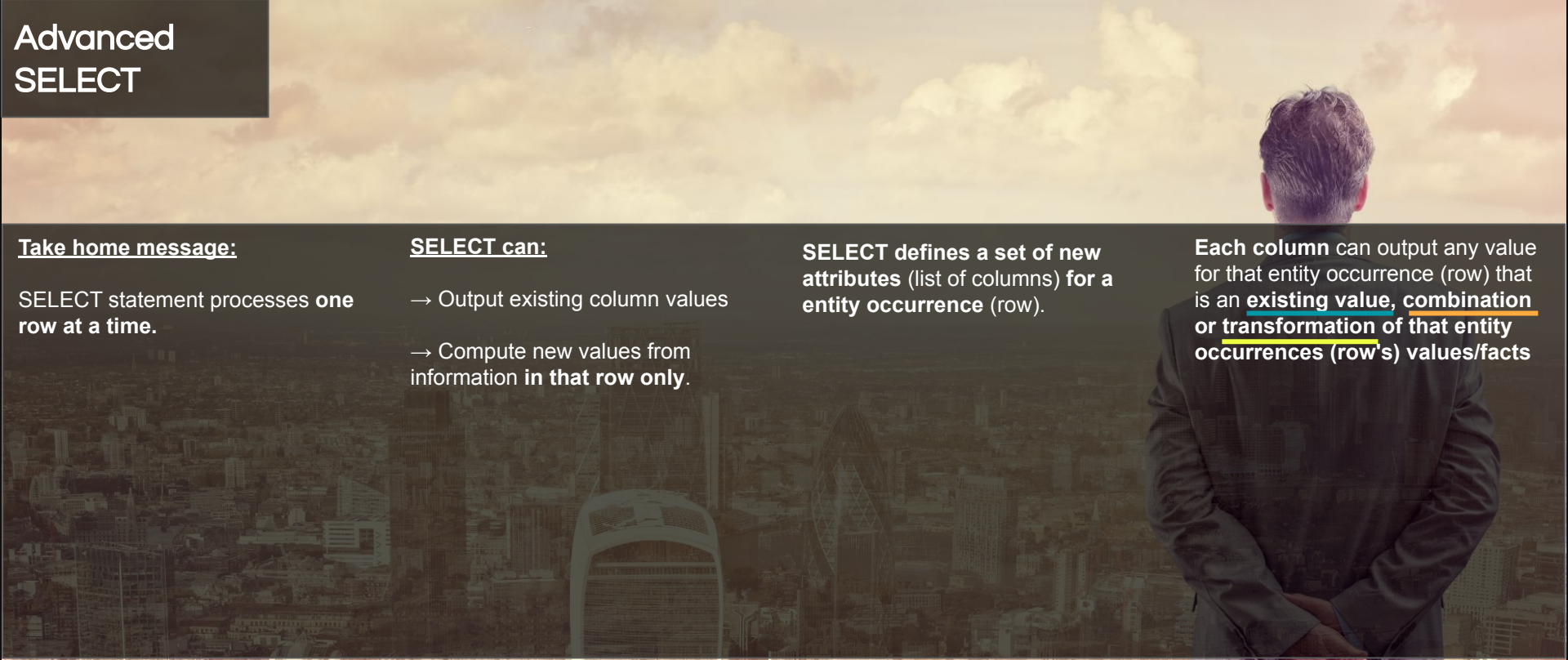
**SELECT can:**

→ 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

| id | code | mark1 | mark2 |
|---|---|---|---|
| S103 | DBS | 72 | 70 |
| S103 | IAI | 58 | 61 |
| 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 |

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

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

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

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.

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**

**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.**

**Example:**

**Transformation**

student

| 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 |

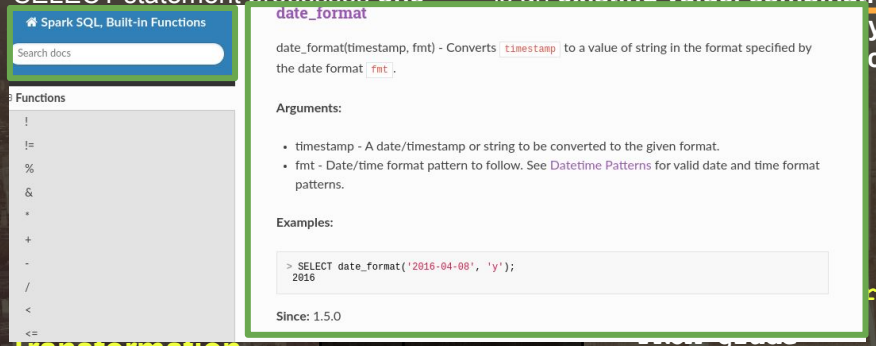
# Advanced SELECT

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**

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**

```
rmat(dob,  'EEEE') as day
```

**Transformation**

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

OUTPUT

| id | day |
|---|---|
| S103 | Wednesday |

# 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**

**Transform**

**Functions**
**parameter**

**postgres**
**or you can**

**Example:**

**Transformation**

student

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

```
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; 03; Q3; 3rd quarter |
| E | day-of-week | text | Tue; Tuesday |
| F | aligned day of week in month | number(1) | |
| a | am-pm-of-day | am-pm | |
| Z | zone-offset | offset-Z | ; -0800; -08:00; |
| ' | escape for text | delimiter | |
| '' | single quote | lite | |
| [ | optional section start | | |
| ] | optional section end | | |

why 4x Es?

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

# Advanced SELECT

```
SELECT id,
       'Born on a ' || date_format(dob,  'EEEE') || ', started studying on a ' || date_fromat(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_fromat(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_fromat(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.

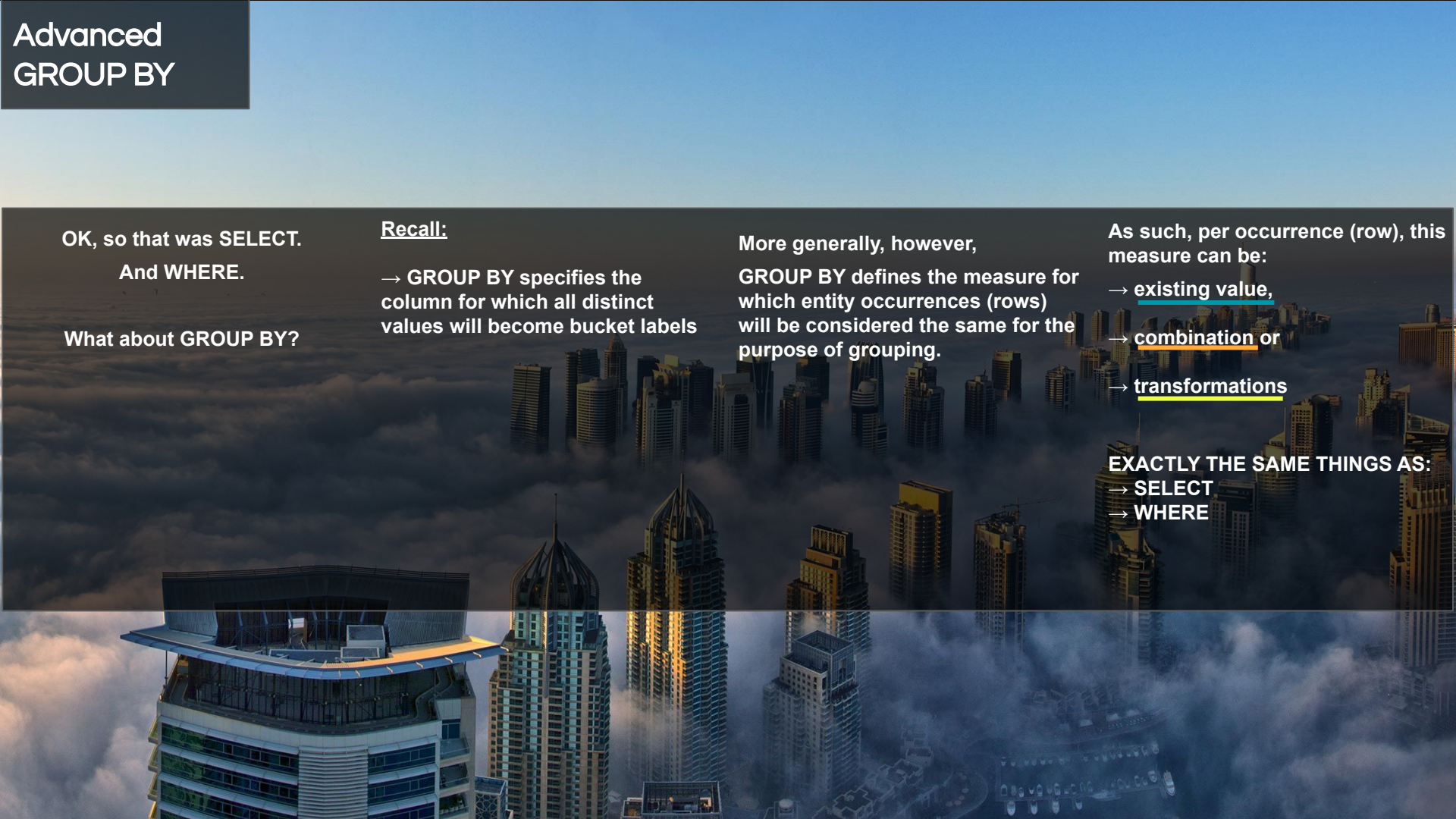
As such, per occurrence (row), this measure can be:

→ 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')
```

**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')
```

**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 |

**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**

student

| 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

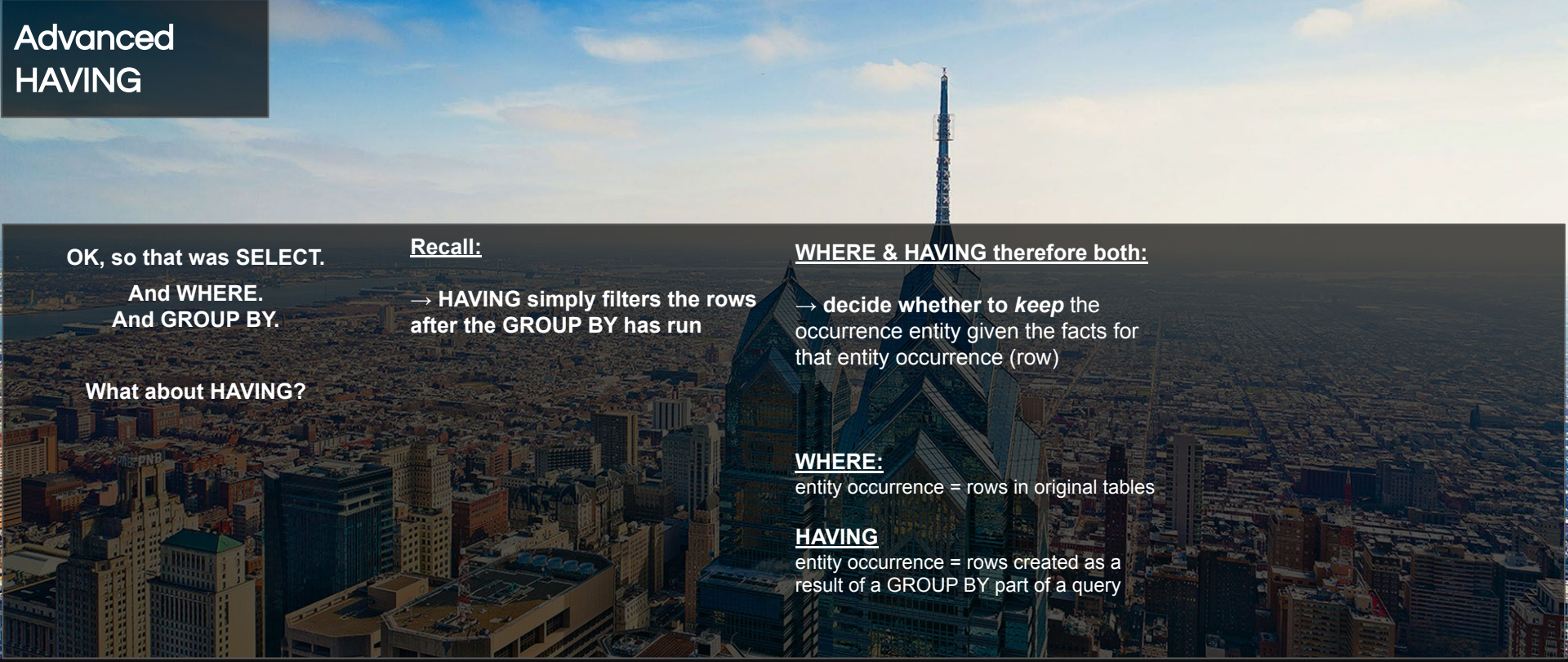
| id | dob | start_date |
|---|---|---|
| S103 | 1990-06-20 | 2001-07-02 |

Wednesday

| id | dob | start_date |
|---|---|---|
| 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!!!!!!**

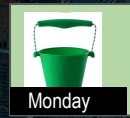
# Advanced HAVING

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

**Example:**
**Combination**
**(Transformation)**

student

| 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 | count(*) |
|---|---|
| Monday | 1 |
| Wednesday | 2 |

# 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.

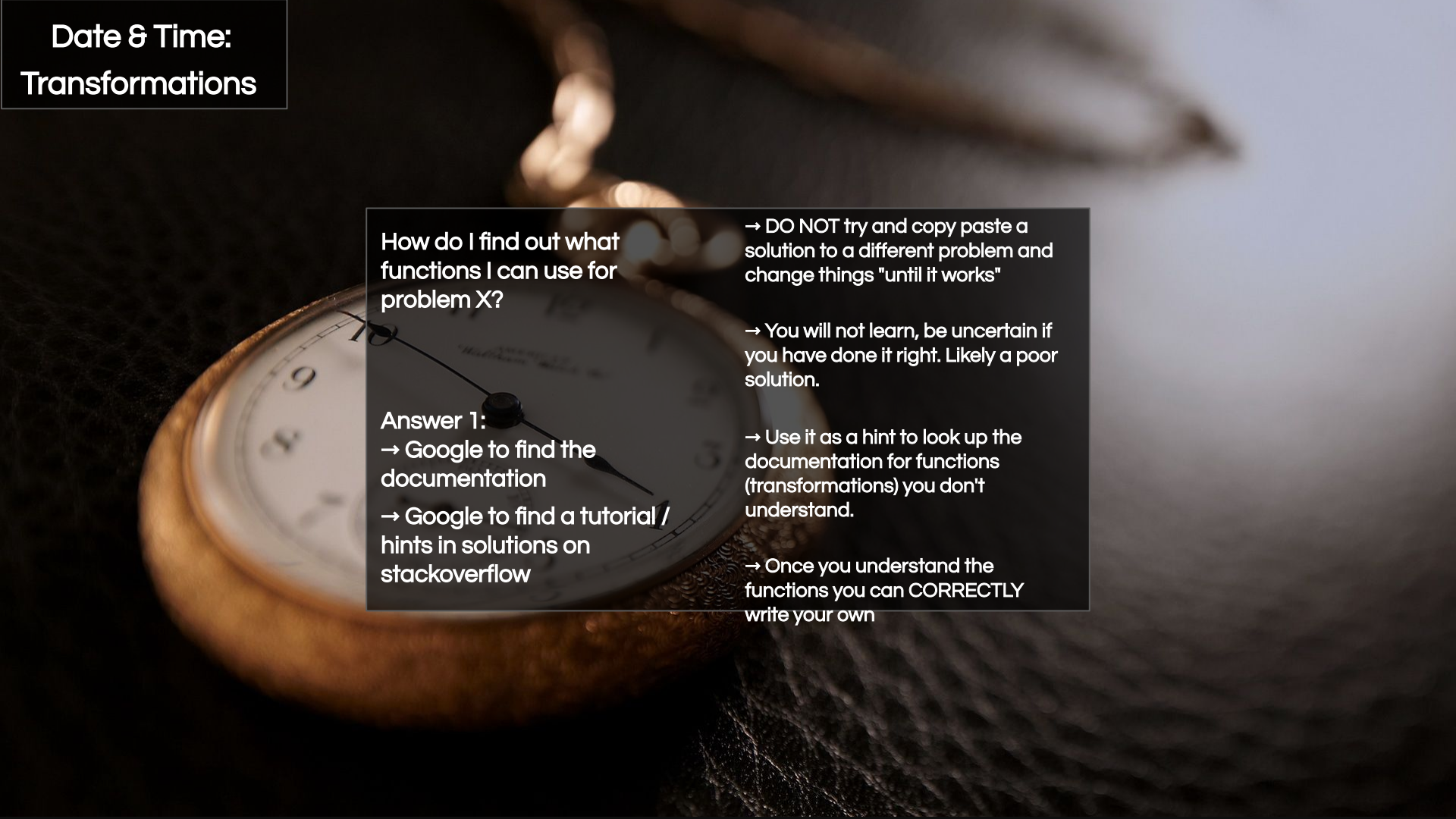
Why?

→ 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

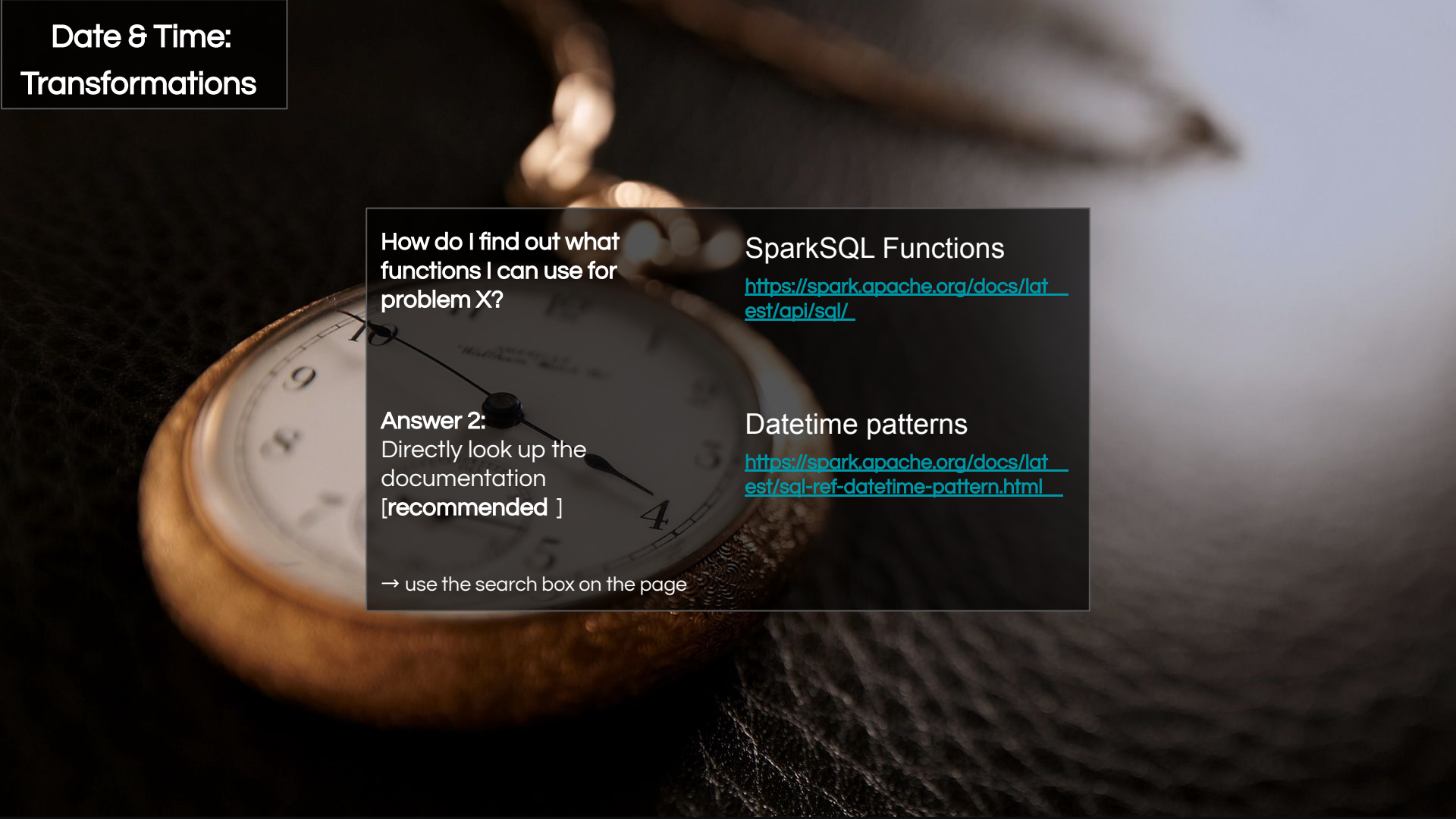
→ DO NOT try and copy paste a solution to a different problem and change things "until it works"

→ 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

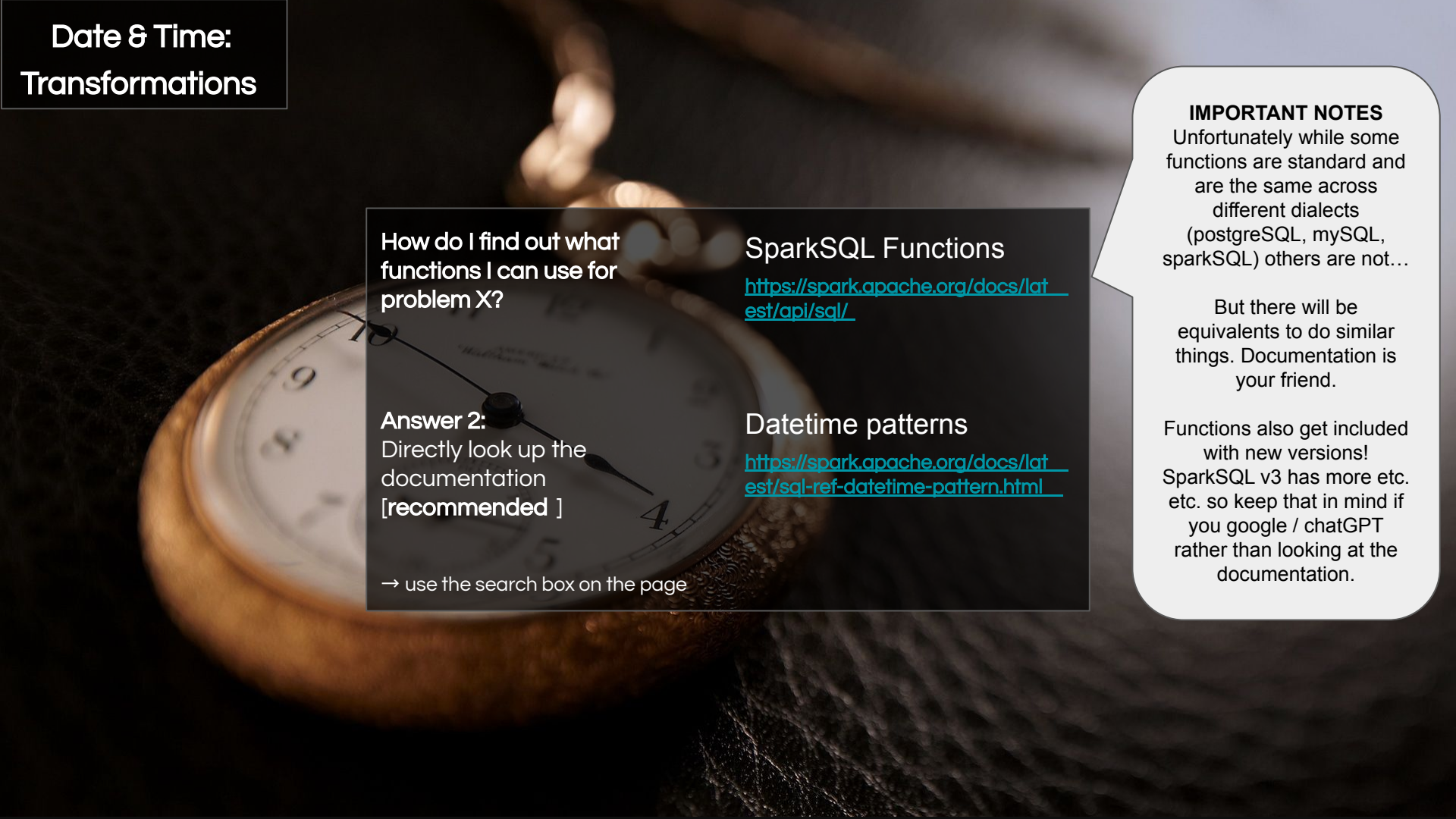
SparkSQL Functions

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

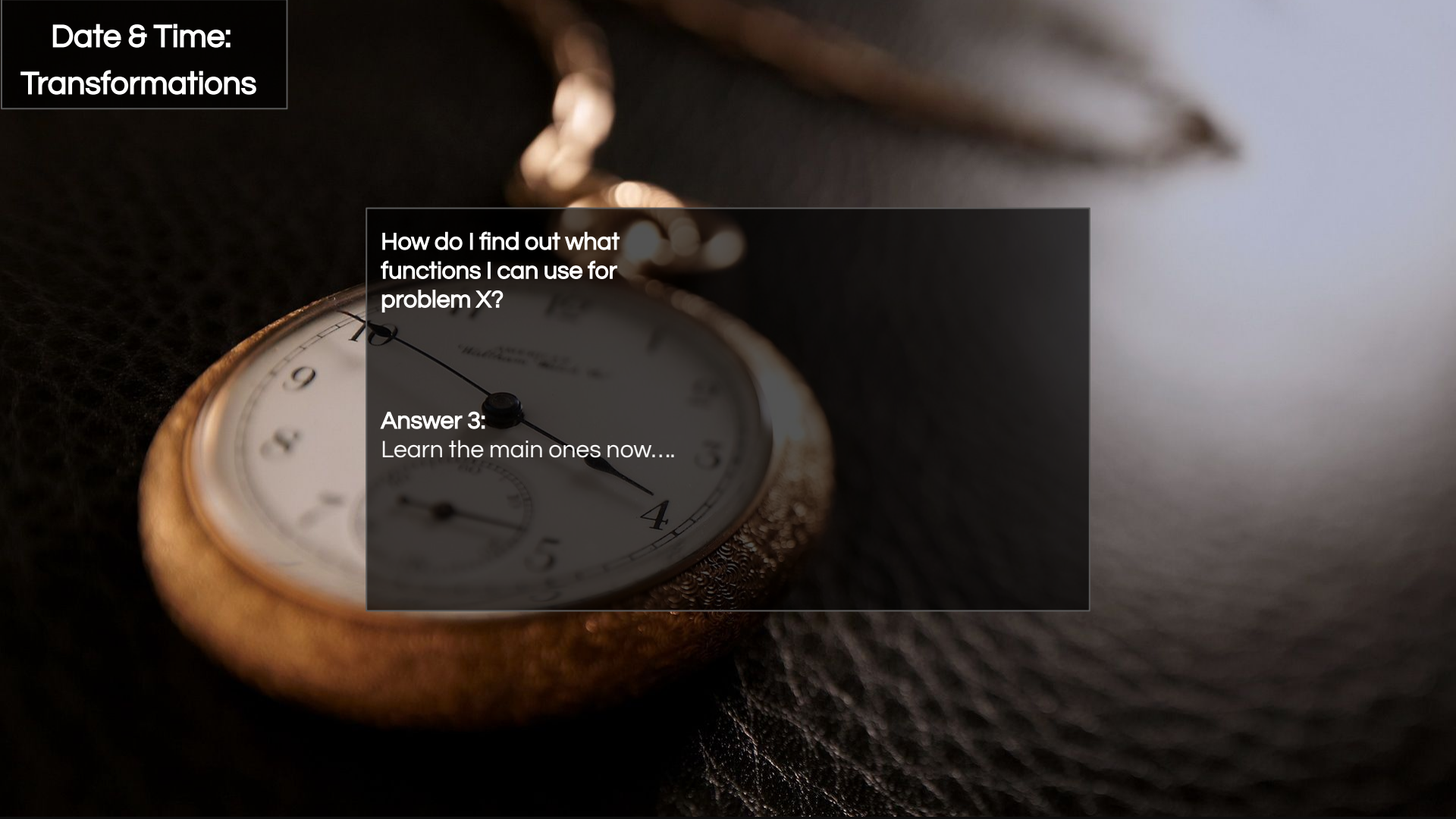
https://spark.apache.org/docs/latest/sql-ref-datetime-pattern.html

**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

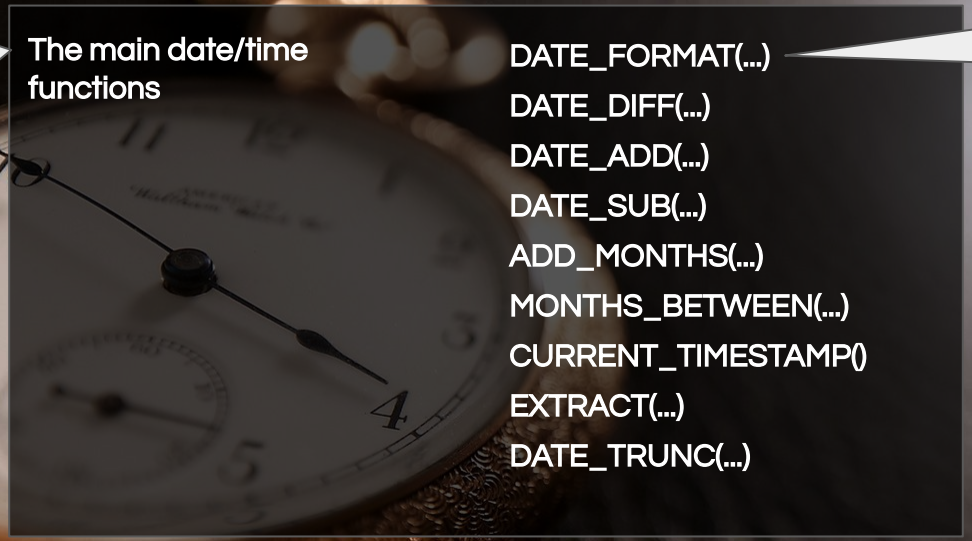


How do I find out what functions I can use for problem X?

**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 |
|---|---|---|---|
| `DATE_DIFF(endDate, startDate)` | Compute the difference between TIMESTAMP or DATE objects. Returns **the number of days.** | `SELECT DATE_DIFF('2007-06-06 10:00:00'::TIMESTAMP, '1706-03-30 09:00:00'::TIMESTAMP);` | **110006**<br>**(type = INTEGER)** |
| `MONTHS_BETWEEN(endDate,startDate)` | Compute the difference between TIMESTAMP or DATE objects. Returns **the number of months.** | `months_between('2007-03-02'::DATE, '2007-01-01'::DATE));` | **2.03225806**<br>**(type = DOUBLE)** |
| `DATE_SUB(startDate, numDays)` | Subtract a given number of days from a DATE.<br>Returns **a DATE**<br><br>WARNING: Using a TIMESTAMP here will currently cause a silent error! | `SELECT date_sub('2016-07-30'::DATE, 1);`<br><br>`SELECT date_sub('2016-07-30 23:00:00':TIMESTAMP, 1);` | **2016-07-29 (type = DATE)**<br><br>**NULL (silent error - do not use!!)** |

# Difference
(via functions)

## What it achieves:
Computes the difference between two dates/times.

| Function | Description | Example | Result |
|---|---|---|---|
| `DATE_DIFF(endDate, startDate)` | Compute the difference between TIMESTAMP or DATE objects. Returns **the number of days.** | `SELECT DATE_DIFF('2007-06-06 10:00:00'::TIMESTAMP, '1706-03-30 09:00:00'::TIMESTAMP);` | **110006**<br>**(type = INTEGER)** |
| `MONTHS_BETWEEN(endDate,startDate)` | Compute the difference between TIMESTAMP or DATE objects. Returns **the number of months.** | `months_between('2007-03-02'::DATE, '2007-01-01'::DATE));` | **2.03225806**<br>**(type = DOUBLE)** |
| `DATE_SUB(startDate, numDays)` | Subtract a given number of days from a DATE.<br>Returns **a DATE**<br><br>WARNING: Using a TIMESTAMP here will currently cause a silent error! | `SELECT date_sub('2016-07-30'::DATE, 1);`<br><br>`SELECT date_sub('2016-07-30 23:00:00':TIMESTAMP, 1);` | **2016-07-29 (type = DATE)**<br><br>**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 |
|---|---|---|---|
| `TIMESTAMP - TIMESTAMP` | Compute the difference between TIMESTAMP. Returns **an "INTERVAL DAY TO SECONDS".** Cast to BIGINT **to get seconds** between the timestamps.<br><br>or DATE objects. Returns **an INTERVAL DAY.** This is a special type | `SELECT '2007-06-06 10:00:00'::TIMESTAMP - '1706-03-30 09:00:00'::TIMESTAMP;`<br><br>`SELECT ('2007-06-06 10:00:00'::TIMESTAMP - '1706-03-30 09:00:00'::TIMESTAMP)::BIGINT` | **Unreadable Object (type = INTERVAL DAY TO SECONDS)**<br><br>**9504522000 (type = BIGINT)** |
| `DATE - DATE` | Compute the difference between DATE. Returns **an "INTERVAL DAY".** Cast to INTEGER **to get days** between the timestamps. | `SELECT '2007-06-06 10:00:00'::DATE - '1706-03-30 09:00:00'::DATE;`<br><br>`SELECT ('2007-06-06 10:00:00'::DATE - '1706-03-30 09:00:00'::DATE)::INTEGER` | **Unreadable Object (type = INTERVAL DAY TO SECONDS)**<br><br>**110006 (type = INTEGER)** |
| `TIMESTAMP - INTERVAL`<br>`DATE - INTERVAL`<br><br>`NOTE: BE CAREFUL OF THE RETURN TYPE!` | 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. | `SELECT '2016-07-30 10:00:00'::TIMESTAMP - INTERVAL 30 MINUTES`<br><br>`SELECT '2016-07-30'::DATE - INTERVAL 30 MINUTES`<br><br>`SELECT '2016-07-30'::DATE - INTERVAL 2 DAYS` | **2016-07-30 09:30:00 (type = TIMESTAMP)**<br><br>**2016-07-29 23:30:00 (type = TIMESTAMP)**<br><br>**2016-07-28 (type = DATE)** |

# Addition

## What it achieves:

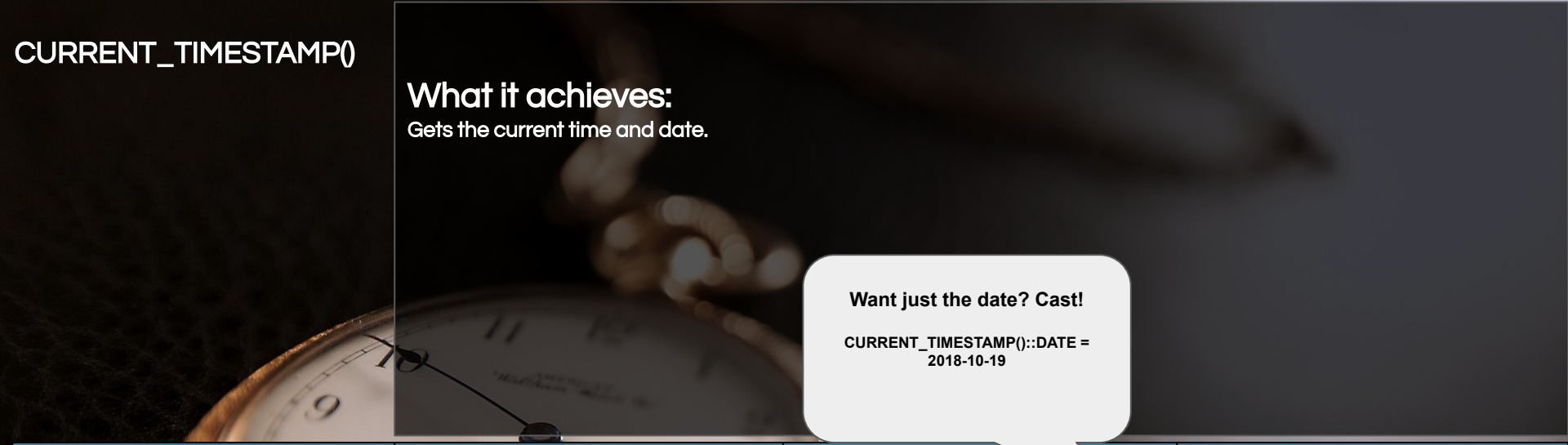
Adds an INTERVAL to a DATE/TIME/INTERVAL/TIMESTAMP

| Function | Description | Example | Result |
|---|---|---|---|
| `TIMESTAMP + INTERVAL`<br><br>`DATE + INTERVAL` | Adds an INTERVAL of time to a TIMESTAMP or DATE<br>**RETURNS a**<br>**DATE** if starting with DATE and adding days, months, years<br>**TIMESTAMP** otherwise | `SELECT '2007-06-06 10:00:00'::TIMESTAMP + INTERVAL 10 days;`<br><br>`SELECT '2007-06-06'::TIMESTAMP + INTERVAL 10 days;` | **2007-06-16 10:00:00 (type = TIMESTAMP)**<br><br>**2007-06-16 (type = DATE)** |
| `DATE_ADD(startDate, numDays)` | Add a given number of days from a DATE or TIMESTAMP.<br>Returns **a DATE** (TIMESTAMP will be truncated!)<br><br>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! | `SELECT date_add('2016-07-29'::DATE, 1);`<br><br>`SELECT date_add('2016-07-29 23:00:00'::TIMESTAMP, 1);` | **2016-07-30 (type = DATE)**<br><br>**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 |
|---|---|---|---|
| `CURRENT_TIMESTAMP()` | Returns the current date and time.<br>**RETURNS a TIMESTAMP** | `SELECT CURRENT_TIMESTAMP();` | **2023-10-21T15:11:46.071+0000**<br>**(type = TIMESTAMP)** |

# EXTRACT

## What it achieves:

Extracts and returns PART of a DATE or TIMESTAMP

| Function | Description | Example | Result |
|---|---|---|---|
| `EXTRACT(field FROM timestamp)` | Get subfield. Options: year, yearofweek, quarter, month, week, day, dayofweek, dayofweek_iso, doy, hour, minute, second See: https://spark.apache.org/docs/latest/api/sql/#extract | `extract(month from '2007-05-01'::TIMESTAMP)` | **5** |
| | | `extract(dow from '2007-05-01'::DATE)` | **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 |
|---|---|---|---|
| `DATE_TRUNC(field,timestamp)` | 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 | `SELECT DATE_TRUNC('month', '2007-05-15'::DATE)` | **2007-05-01 00:00:00** |
| | | `SELECT DATE_TRUNC('hour', '2007-05-01 09:30:05'::TIMESTAMP)` | **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).

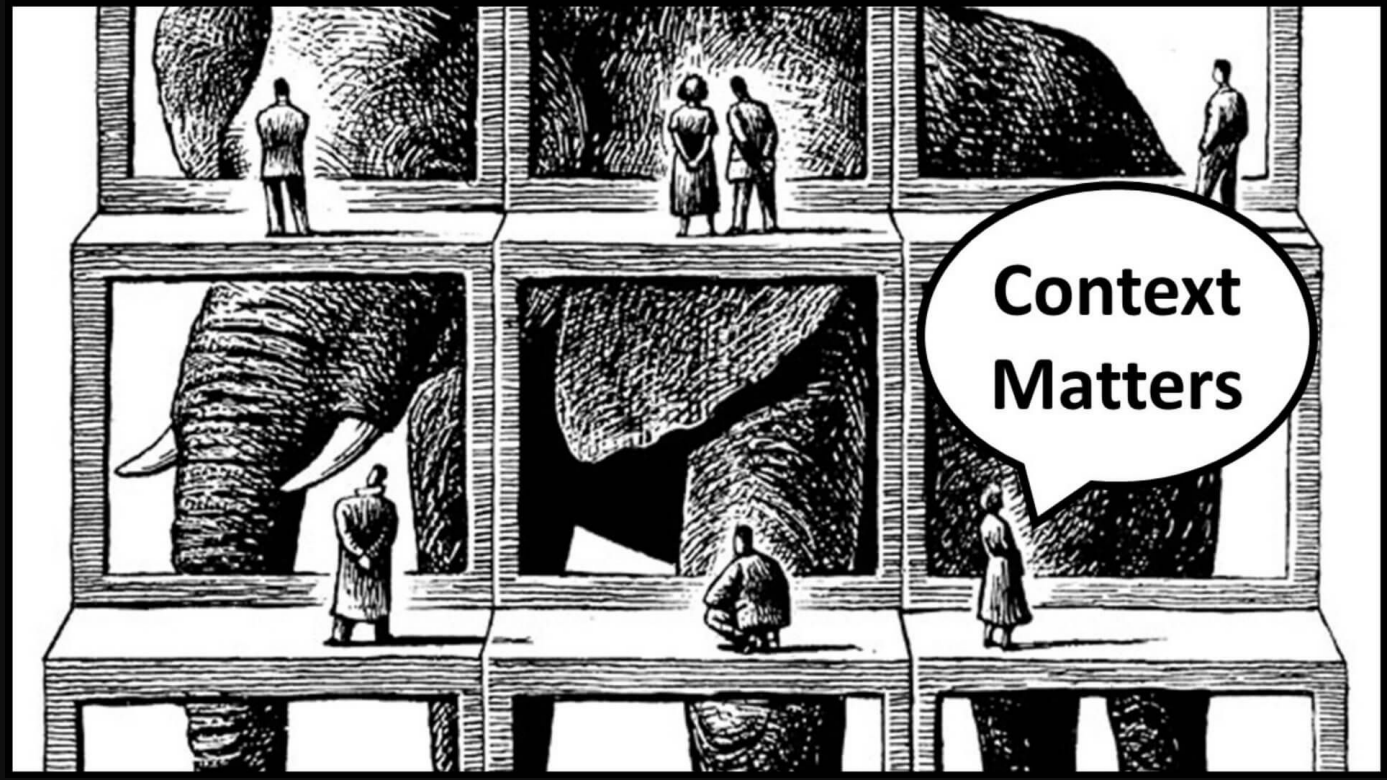
## Real world use:

Return the total sales per month.

```
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 |
|---|---|---|---|
| `DATE_TRUNC(field,timestamp)` | 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 | `SELECT DATE_TRUNC('month', '2007-05-15'::DATE)` | **2007-05-01 00:00:00** |
| | | `SELECT DATE_TRUNC('hour', '2007-05-01 09:30:05'::TIMESTAMP)` | **2007-05-01 09:00:00** |

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



***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...*

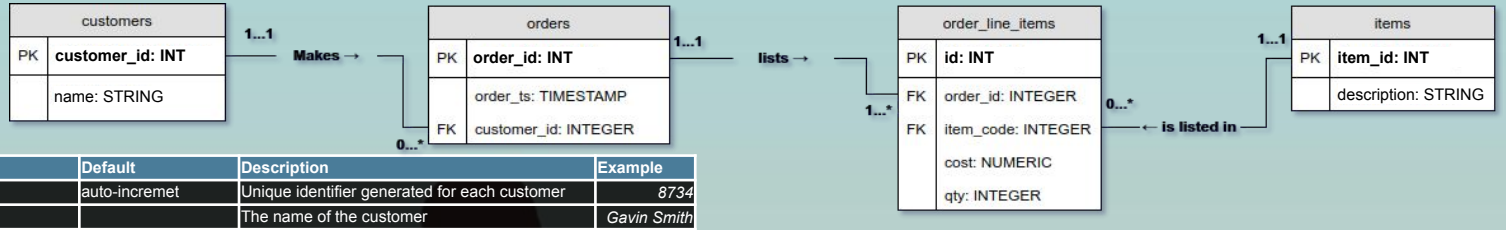
1) *List the customers by name who shop with us more than once a month?*

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* |

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

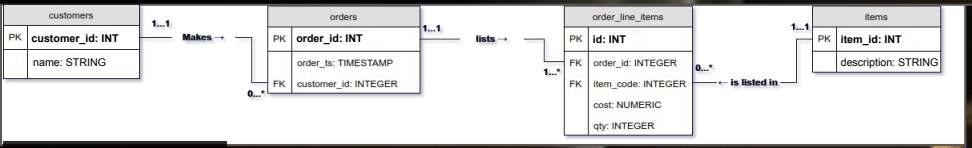
1) *List the customers by name who shop with us more than once a month while they were active customers?*

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 | *545* |
| **description** | STRING | NOT NULL | | The description of the item | *Mellowship Slinky Decaf* |

## 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

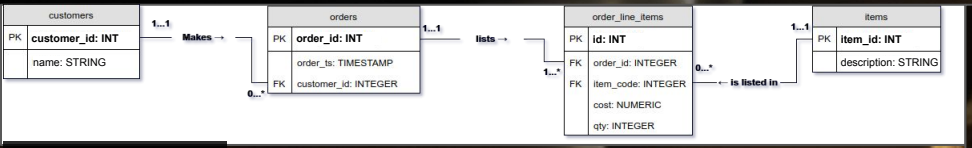
| Field Name | Data Type | Constraint | Default | Description | Example |
|---|---|---|---|---|---|
| **order_id** | INTEGER | Primary Key | auto-increment | Unique identifier generated for each order | *123456* |
| **order_ts** | TIMESTAMP | NOT NULL | now() | Timestamp of when the order was placed | *2017-01-01 13:01:02* |
| **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 |
|---|---|---|---|---|---|
| **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 | *545* |
| **description** | STRING | NOT NULL | | The description of the item | *Mellowship Slinky Decaf* |

## 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 13:01:02* |
| **customer_id** | INTEGER | NOT NULL | | The id of the customer that made the order. | *8734* |

***QUESTION***

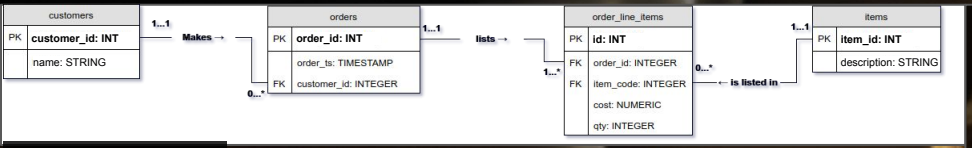
*List the customers by name who shop with us more than once a month while they were active customers?*

**How to do this?**

**I'd work how to do it in parts.**

## 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 | *545* |
| **description** | STRING | NOT NULL | | The description of the item | *Mellowship Slinky Decaf* |

## 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 13:01:02* |
| **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?*

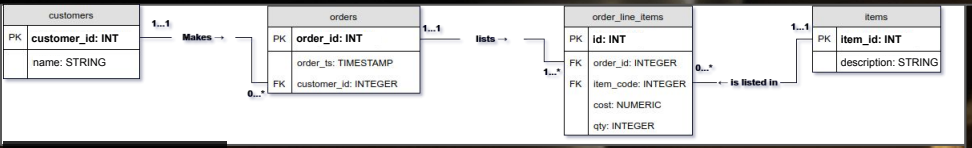
```
SELECT
FROM customers
WHERE
```

**How to do this?**

**I'd work how to do it in parts.**

## 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 | *545* |
| **description** | STRING | NOT NULL | | The description of the item | *Mellowship Slinky Decaf* |

## 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 13:01:02* |
| **customer_id** | INTEGER | NOT NULL | | The id of the customer that made the order. | *8734* |

***QUESTION***

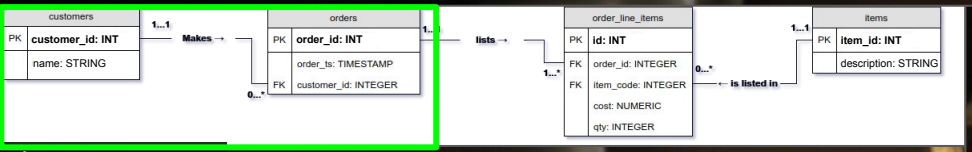
*List the customers by name who shop with us more than once a month while they were active customers?*

```
SELECT
FROM customers, orders
WHERE
```

**Ah! So we have a join to deal with!**

## 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 | *545* |
| **description** | STRING | NOT NULL | | The description of the item | *Mellowship Slinky Decaf* |

## 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 13:01:02* |
| **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?*

We have a one-to-many join.

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

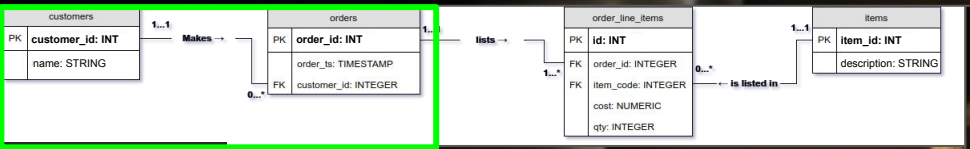
```
SELECT
FROM orders
JOIN customers
ON orders.customer_id =
customers.customer_id
```

## order_line_item

| Field Name | Data Type | Constraint | Default | Description | Example |
|---|---|---|---|---|---|
| **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* |

***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 |
|---|---|---|---|---|---|
| **item_id** | SERIAL | Primary Key | auto-increment | Unique identifier generated for each item | *545* |
| **description** | STRING | NOT NULL | | The description of the item | *Mellowship Slinky Decaf* |

## 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 13:01:02* |
| **customer_id** | 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
```

**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).

**So my logic:**

1) Each row: order + customer name

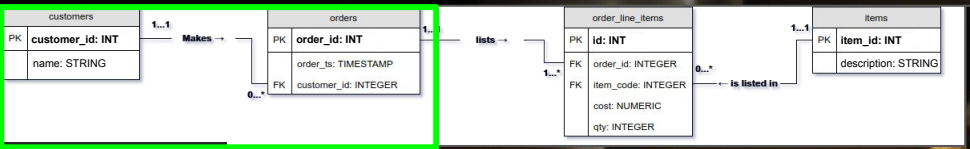
2) For each customer compute: number of months visited

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 |
|---|---|---|---|---|---|
| **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 | *545* |
| **description** | STRING | NOT NULL | | The description of the item | *Mellowship Slinky Decaf* |

## 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 13:01:02* |
| **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?*

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
```

**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).

**So my logic:**

1) Each row: order + customer name

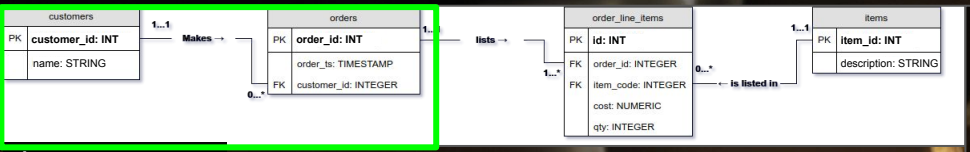
2) For each customer compute: number of months visited

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 |
|---|---|---|---|---|---|
| **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 | *545* |
| **description** | STRING | NOT NULL | | The description of the item | *Mellowship Slinky Decaf* |

## 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 13:01:02* |
| **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?*

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
```

**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).

**So my logic:**

1) Each row: order + customer name

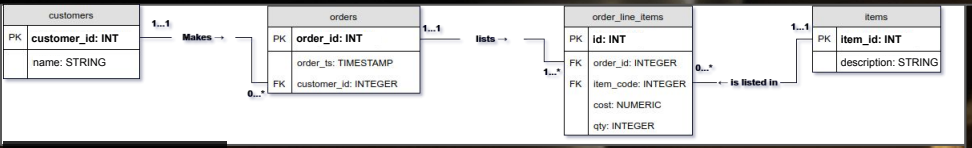
2) For each customer compute: number of months visited

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 |
|---|---|---|---|---|---|
| **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 | *545* |
| **description** | STRING | NOT NULL | | The description of the item | *Mellowship Slinky Decaf* |

## 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 13:01:02* |
| **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?*

**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

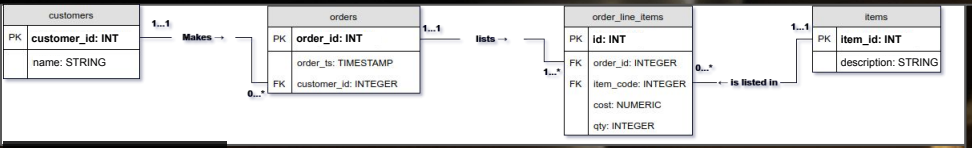
4) Then filter: (2) / (3) > 1

**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 |
|---|---|---|---|---|---|
| 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 | *545* |
| description | STRING | NOT NULL | | The description of the item | *Mellowship Slinky Decaf* |

## 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 13:01:02* |
| 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?*

**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 |
|---|---|---|---|---|---|
| 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_c | | | | | *545* |
| cost | | | | | *16.50* |
| qty | | | | | *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 |
|---|---|---|---|---|---|
| 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_c | | | | | 545 |
| cost | | | | | 16.50 |
| qty | | | | | 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 |
|---|---|---|---|---|---|
| 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_c | | | | | 545 |
| cost | | | | | 16.50 |
| qty | | | | | 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

**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 |
|---|---|---|---|---|---|
| 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_c | | | | | 545 |
| cost | | | | | 16.50 |
| qty | | | | | 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:**

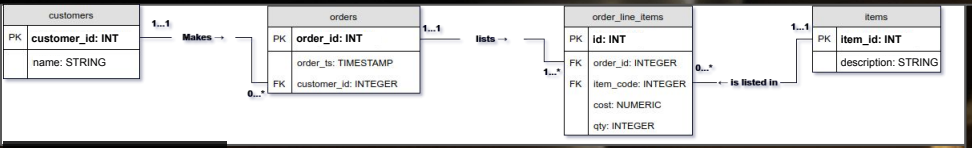
1) Each row: order + customer name
2) For each customer compute: number of months visited
3) For each customer compute: number of months active
4) Then filter: (2) / (3) > 1

**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 |
|---|---|---|---|---|---|
| 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 | *545* |
| description | STRING | NOT NULL | | The description of the item | *Mellowship Slinky Decaf* |

## 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 13:01:02* |
| 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?*

**So my logic:**

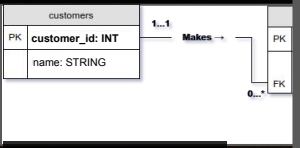
1) Each row: order + customer name
2) For each customer compute: number of months visited
3) For each customer compute: number of months active
4) Then filter: (2) / (3) > 1

**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) )
       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
```

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 | | s line item | 123456 |
| item_code | INTEGER | Foreign K NOT NU | | item. | 545 |
| cost | NUMERIC | NOT NU | | ne item. | 16.50 |
| qty | INTEGER | CHECK( | | ne item. | 1 |




items

| Field Name | Data Type | Constraint | mple |
|---|---|---|---|
| item_id | SERIAL | Primary Ke | 545 |
| description | STRING | NOT NULL | Mellowship inky Decaf |

customers

| Field Name | Data Type | Constraint | mple |
|---|---|---|---|
| customer_id | INTEGER | Primary Ke | 8734 |
| name | STRING | NOT NULL | avin Smith |

orders

| Field Name | Data Type | Constraint | mple |
|---|---|---|---|
| order_id | INTEGER | Primary Ke | 123456 |
| order_ts | TIMESTAMP | NOT NULL | 017-01-01 13:01:02 |
| customer_id | INTEGER | NOT NULL | 8734 |

And we're done!

***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.

***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) )
       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!

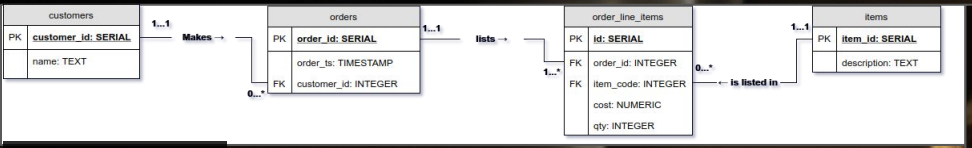
1) *List the customers by name who shop with us more than once a month while they were active customers?*

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 |
|---|---|---|---|---|---|
| **id** | SERIAL | 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 | *545* |
| **description** | TEXT | NOT NULL | | The description of the item | *Mellowship Slinky Decaf* |

## customers

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

## orders

| Field Name | Data Type | Constraint | Default | Description | Example |
|---|---|---|---|---|---|
| **order_id** | SERIAL | 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 13:01:02* |
| **customer_id** | 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
```

*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`**.

*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?*

```
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
```

*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`**.

*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?*

```
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
```

*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;
```

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
```

*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, name
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

# *We can do better! subqueries!*

```
SELECT COUNT(*)
FROM result
```

2

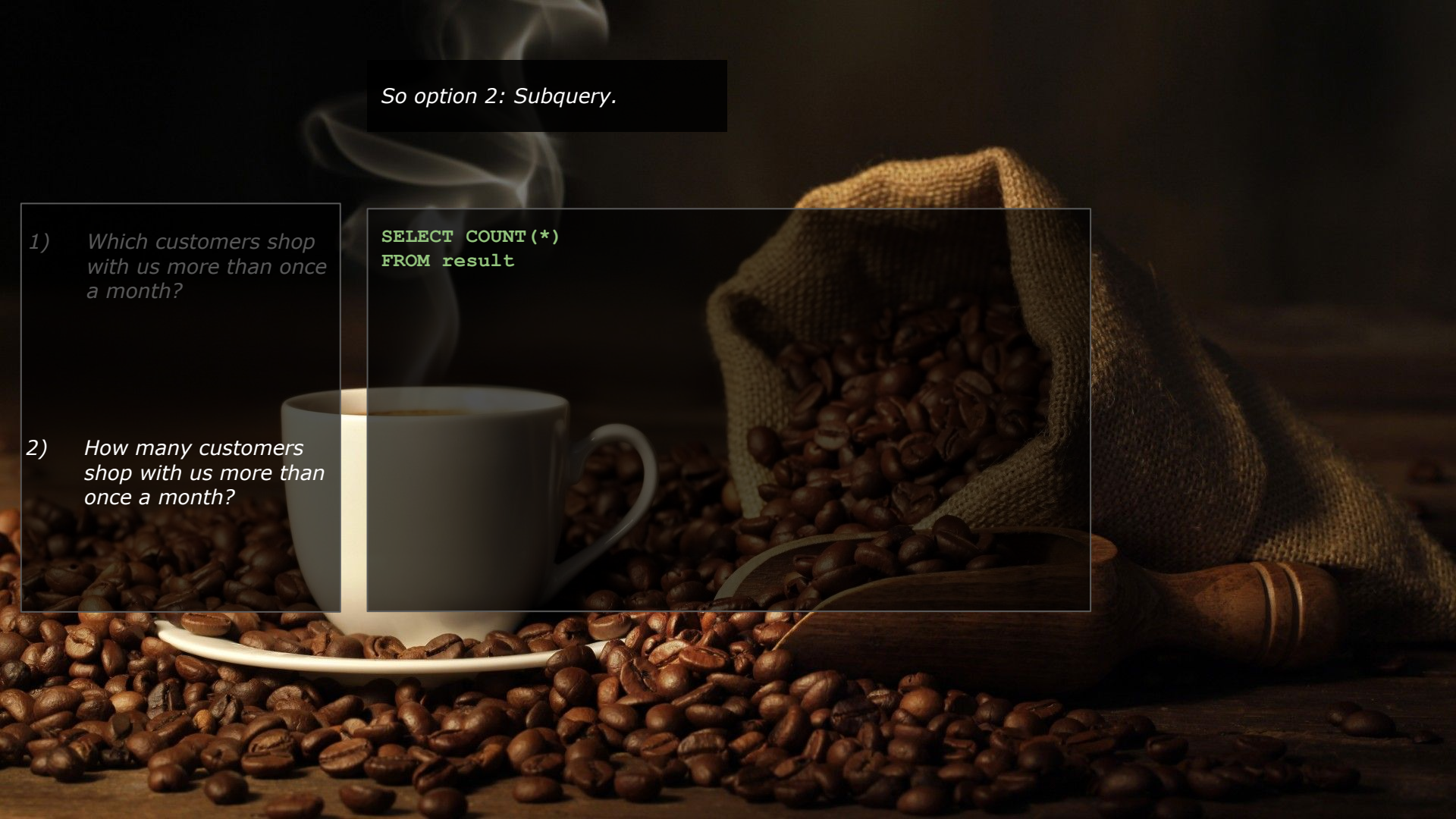
*So option 1:*
*Create a physical new table called result. Run a second query.*

*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 custome… shop with us more tha… 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_montly_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;
```


Exercise!