Intermediate Postgres SQL Training: If There Is a Will, There Is a SQL Query That Will Get It Done

BY BEN FOUTS, MPH, DATA ANALYST, RCHC

SQL TRAINING SESSION, MAY 7, 2021

REDWOOD HEALTH COMMUNITY COALITION

Serving Sonoma, Napa, Marin & Yolo Counties

References

https://www.tutorialspoint.com/postgresql/index.htm

https://www.postgresql.org/docs/9.6/index.html

https://www.postgresqltutorial.com/

Google: Postgres + Command name

Prerequisites

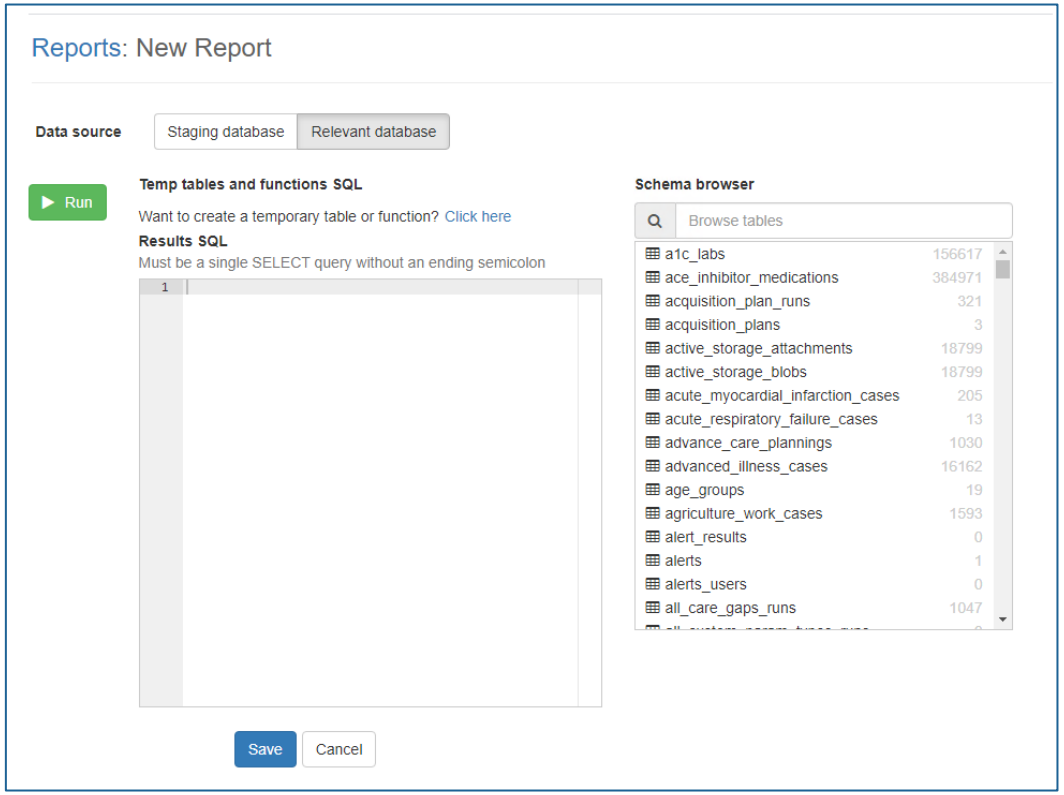
Participants in the session today should already know:

- Basic query commands like SELECT, FROM, WHERE, ORDER BY, GROUP BY
- How to read and understand a basic query
- How to write a basic query
- General familiarity with the tables in the Relevant Production database

What You Will Need for the Exercises

- DataGrip

- -- or-- Relevant



- We will be working on the “Production” database because all tables there have the same name and same fields

Other Notes

- I am not going to work on DataGrip or Relevant directly in this presentation because I do not want to accidently display any patient information.
- There will be various exercises where you will be given a problem to work through. Use your own version of DataGrip or Relevant. Or else, just sketch it out on a piece of paper.
- These exercises will be timed with a timer.

One Note on Terminology

I am using the following terminology:

- A “statement” is SQL code that operates on or returns a set of data
- A “clause” is a subunit of the statement
- A “command” is the action word in the statement

Example

- Statement:

```
SELECT mrn FROM patients WHERE last_name = 'Lopez'
```

- Clause:

```
SELECT mrn

FROM patients

WHERE last_name = 'Lopez'
```

- Command:

```
SELECT,   FROM,    WHERE
```

Intermediate Postgres Clauses

A DEEP DIVE INTO SIX COMMON COMMANDS AND HOW TO USE THEM

1. Getting the “Last” Record

WITH TWO METHODS

Getting the “Last” Record

- Last could mean the most recent (usually our need)
- The same approach is also used to find the “first” record
- There are two methods that will be explored in this presentation

What the data looks like

- For example, let's find the last A1c lab by patient
- Importer: a1c_labs

```
SELECT patient_id, performed_on, result
FROM a1c_labs
ORDER BY patient_id, performed_on DESC
```

patient_id	performed_on	result
10	2/11/2021	5.4
10	3/4/2020	6
10	3/5/2019	5.5
10	2/22/2019	4.9
11	3/1/2021	5.1
11	12/28/2020	6.6
11	8/30/2019	9.2

Method #1: Using a Row Number

```
SELECT
      patient_id,
      performed_on,
      result,
      ROW_NUMBER()
          OVER (
             PARTITION BY patient_id
             ORDER BY performed_on DESC ) AS row
FROM a1c_labs
```

Adapted from SQL code found in the Quality Measure “Diabetes: Hemoglobin A1c Control (<=9%) (UDS 2020 Table 7, inverted)”

```
SELECT
      patient_id,
      performed_on,
      result,
      ROW_NUMBER()
            OVER (
                  PARTITION BY patient_id
                  ORDER BY performed_on DESC ) AS row
FROM a1c_labs
```

Display a row number

Split the table into unique patient_id numbers. In other words, re-start the numbering with each unique patient_id

Order the rows within each block of patient_id by performed_on in descending order

Name the column “row”

If you wanted the first record, you would set the order as ascending, or ASC

Data Output

The data in the output is the same as on the original table, but it has been ordered and there is a new column “row”

patient_id	performed_on	result	row
10	2/11/2021	5.4	1
10	3/4/2020	6	2
10	3/5/2019	5.5	3
10	2/22/2019	4.9	4
11	3/1/2021	5.1	1
11	12/28/2020	6.6	2
11	8/30/2019	9.2	3

So, if this is made into a TEMPORARY TABLE, it can be JOINED to another table to display the performed_on date and result of the most recent lab by referencing row = 1

```
DROP TABLE IF EXISTS last_test;
CREATE TEMPORARY TABLE last_test AS
SELECT
      patient_id,
      performed_on,
      result,
      ROW_NUMBER()
            OVER (
                  PARTITION BY patient_id
                  ORDER BY performed_on DESC ) AS row
FROM a1c_labs;


SELECT
      patients.id,
      last_test.performed_on,
      last_test.result
FROM patients
      INNER JOIN last_test ON last_test.patient_id = patients.id
            AND last_test.row = 1
```

JOIN the TEMPORARY TABLE last_test to the table patients using the id fields AND where the row column in last_test is equal to 1

Exercise: Display the last systolic blood pressure by patient

Data looks like this:

```
SELECT patient_id, performed_on, systolic_blood_pressure
FROM blood_pressure_readings
```

patient_id	performed_on	systolic_blood_pressure
25	3/11/2021	140
25	1/22/2020	94
25	7/3/2019	153
26	1/16/2021	144
26	9/26/2020	128
26	8/22/2020	119
26	7/31/2019	130

From previous slides

Exercise: Display the last systolic blood pressure by patient

```
DROP TABLE IF EXISTS last_test;
CREATE TEMPORARY TABLE last_test AS
SELECT
    patient_id,
    performed_on,
    result,
    ROW_NUMBER()
        OVER (
            PARTITION BY patient_id
            ORDER BY performed_on DESC ) AS row
FROM a1c_labs;
```

```
SELECT patient_id, performed_on, systolic_blood_pressure
FROM blood_pressure_readings
```

```
SELECT
      patient_id,
      performed_on,
      systolic_blood_pressure,
      ROW_NUMBER()
            OVER (
                  PARTITION BY patient_id
                  ORDER BY performed_on DESC ) AS row
FROM blood_pressure_readings;
```

ANSWER

Next task: attach the last performed_on date in 2020 to a query with the table patients

From previous slides

Exercise: attach the last performed_on date in 2020 to a query with the table patients

```
DROP TABLE IF EXISTS last_test;
CREATE TEMPORARY TABLE last_test AS
SELECT
    patient_id,
    performed_on,
    result,
    ROW_NUMBER()
        OVER (
            PARTITION BY patient_id
            ORDER BY performed_on DESC ) AS row
FROM a1c_labs;


SELECT
    patients.id,
    last_test.performed_on,
    last_test.result
FROM patients
    INNER JOIN last_test ON last_test.patient_id = patients.id
        AND last_test.row = 1
```

```
SELECT patient_id, performed_on, systolic_blood_pressure
FROM blood_pressure_readings
```

ANSWER

```
DROP TABLE IF EXISTS last_bp;
CREATE TEMPORARY TABLE last_bp AS
SELECT
      patient_id,
      performed_on,
      systolic_blood_pressure,
      ROW_NUMBER()
            OVER (
                  PARTITION BY patient_id
                  ORDER BY performed_on DESC ) AS row
FROM blood_pressure_readings
WHERE performed_on BETWEEN '2020-01-01' AND '2020-12-31' ;

SELECT
      patients.id,
      last_bp.performed_on,
      last_bp.systolic_blood_pressure
FROM patients
      INNER JOIN last_bp ON last_bp.patient_id = patients.id
            AND last_bp.row = 1
```

Next task: display the **lowest** systolic blood pressure on the last date it was taken in 2020

Example of raw data:

patient_id	performed_on	systolic_blood_pressure
42	3/11/2021	100
42	3/11/2021	110
42	7/3/2019	153

ANSWER

Add another ORDER BY

```
DROP TABLE IF EXISTS last_bp;
CREATE TEMPORARY TABLE last_bp AS
SELECT
      patient_id,
      performed_on,
      systolic_blood_pressure,
      ROW_NUMBER()
            OVER (
                  PARTITION BY patient_id
                  ORDER BY performed_on DESC, systolic_blood_pressure ASC ) AS row
FROM blood_pressure_readings
WHERE performed_on BETWEEN '2020-01-01' AND '2020-12-31' ;

SELECT
      patients.id,
      last_bp.performed_on,
      last_bp.systolic_blood_pressure
FROM patients
      INNER JOIN last_bp ON last_bp.patient_id = patients.id
            AND last_bp.row = 1
```

Method #2: Using SELECT DISTINCT

- Similar to the Row Number method, but does not add a new column
- The required data is simply displayed. There are fewer rows than the original table

patient_id	performed_on	result
10	2/11/2021	5.4
10	3/4/2020	6
10	3/5/2019	5.5
10	2/22/2019	4.9
11	3/1/2021	5.1
11	12/28/2020	6.6
11	8/30/2019	9.2

patient_id	performed_on	result
10	2/11/2021	5.4
11	3/1/2021	5.1

For example, display last A1c lab by patient

```
SELECT DISTINCT ON (patient_id)
    patient_id,
    performed_on,
    result
FROM a1c_labs
ORDER BY patient_id, performed_on DESC
```

Will display one record, which is distinct on the field patient_id

It chooses this record based on the order of the fields patient_id and performed_on (in descending order)

Compare the code

```
SELECT DISTINCT ON (patient_id)
    patient_id,
    performed_on,
    result
FROM a1c_labs
ORDER BY patient_id, performed_on DESC
```

```
SELECT
        patient_id,
        performed_on,
        result,
        ROW_NUMBER()
                OVER (
                        PARTITION BY patient_id
                        ORDER BY performed_on DESC ) AS row
FROM a1c_labs
```

Next exercise: re-work the blood pressure query to display the last blood pressure date in 2020

```
DROP TABLE IF EXISTS last_bp;
CREATE TEMPORARY TABLE last_bp AS
SELECT
      patient_id,
      performed_on,
      systolic_blood_pressure,
      ROW_NUMBER()
            OVER (
                  PARTITION BY patient_id
                  ORDER BY performed_on DESC,
systolic_blood_pressure ASC ) AS row
FROM blood_pressure_readings
WHERE performed_on BETWEEN '2020-01-01' AND '2020-12-31' ;
```

```
SELECT DISTINCT ON (patient_id)
    patient_id,
    performed_on,
    result
FROM a1c_labs
ORDER BY patient_id, performed_on DESC
```

ANSWER

```
SELECT DISTINCT ON (patient_id)
      patient_id,
      performed_on,
      systolic_blood_pressure
FROM blood_pressure_readings
WHERE performed_on BETWEEN '2020-01-01' AND '2020-12-31'
ORDER BY patient_id, performed_on DESC;
```

Adapted from SQL code found in the Quality Measure “Controlling High Blood Pressure (UDS 2020 Table 7)”

Next task: display the lowest systolic blood pressure reading on the last date taken in 2020

```
SELECT DISTINCT ON (patient_id)
    patient_id,
    performed_on,
    systolic_blood_pressure
FROM blood_pressure_readings
WHERE performed_on BETWEEN '2020-01-01' AND '2020-12-31'
ORDER BY patient_id, performed_on DESC, systolic_blood_pressure ASC;
```

ANSWER

Add another ORDER BY

2. Comparing Two Records in a Sequence

Comparing Two Records in a Sequence

- You can compare the value of a field in one record to the value in a field in another record
- Normally done when the records can be sequenced, for example, according to date
- You can use this when you need to make a calculation
- Calculations are done on a single record, not between records. So, this method brings the value from another record in the sequence into a record so that a calculation can be made

The LAG Command

- Example: you want to know how many days have elapsed from one well-child visit to the next
- Importer: well_child_interventions
- Example of raw data:

```
SELECT patient_id, started_on
FROM well_child_interventions
ORDER BY patient_id, started_on
```

patient_id	started_on
10	9/21/2020
10	10/14/2020
10	11/11/2020
11	10/15/2020
11	11/5/2020
11	11/11/2020

```
SELECT
        patient_id,
        started_on AS visit_date,
        LAG(started_on, 1)
                OVER (
                        PARTITION BY patient_id
                        ORDER BY started_on ASC) AS previous_visit_date
FROM well_child_interventions
```

started_on is the field we are pulling into the record

Number of rows between current and target record

Divide the records into groups by unique patient_id numbers

Order the rows within each block of patient_id by started_on in ascending order

Name the column "previous_visit_date"

Adapted from SQL code found in the Quality Measure "Well-Child Visits in the First 15 Months of Life (QIP 2021)"

Resulting Data

The data in the statement output is the same as on the original table, but it has been ordered and there is a new column “previous_visit_date”

patient_id	visit_date	previous_visit_date
10	9/21/2020	
10	10/14/2020	9/21/2020
10	11/11/2020	10/14/2020
11	10/15/2020	
11	11/5/2020	10/15/2020
11	11/11/2020	11/5/2020

Null because there is no previous_visit_date prior to this record (it is the first record for that patient_id)

So, if this is made into a TEMPORARY TABLE, a calculation can be made to compare the visit_date field to the previous_visit_date field

```
DROP TABLE IF EXISTS compare;
CREATE TEMPORARY TABLE compare AS
SELECT
        patient_id,
        started_on AS visit_date,
        LAG(started_on, 1)
                OVER (
                    PARTITION BY well_child_interventions.patient_id
                    ORDER BY started_on ASC) AS previous_visit_date
FROM well_child_interventions
WHERE started_on BETWEEN '2020-01-01' AND '2020-12-31'

SELECT
    patient_id,
    visit_date,
    previous_visit_date,
    visit_date - previous_visit_date AS days_difference
FROM compare
ORDER BY patient_id, visit_date
```

patient_id	visit_date	previous_visit_date	days_difference
10	9/21/2020		
10	10/14/2020	9/21/2020	23
10	11/11/2020	10/14/2020	28
11	10/15/2020		
11	11/5/2020	10/15/2020	21
11	11/11/2020	11/5/2020	6

Exercise (this example is fabricated)

- Since 2020, a goal of your clinic is to have patients with diabetes get an A1c test every 4 months, but (hopefully) no more than 6 months apart
- You have a list of patients with diabetes (separate) and just want to calculate the difference between A1c tests since 1/1/2020
- Importer: a1c_labs
- Key fields: patient_id, performed_on

ANSWER

```
DROP TABLE IF EXISTS compare;
CREATE TEMPORARY TABLE compare AS
SELECT
        patient_id,
        performed_on AS lab_date,
        LAG(performed_on, 1)
                OVER (
                    PARTITION BY patient_id
                    ORDER BY performed_on ASC) AS previous_lab_date
        FROM a1c_labs
WHERE performed_on >= '2020-01-01';

SELECT
    patient_id,
    lab_date,
    previous_lab_date,
    lab_date - previous_lab_date AS days_difference,
    CASE WHEN previous_lab_date IS NULL
        THEN 'First record'
      WHEN (lab_date - previous_lab_date) > 60
        THEN 'Over 60 days’
        ELSE 'Under 60 days' END AS group_difference
FROM compare
ORDER BY patient_id, lab_date
```

Idea: this query can JOINED to a universe and then labs can be counted by group in the column group_difference (use a GROUP BY clause)

3. Examine the Overlap Between Two Time Periods

Examine the Overlap Between Two Time Periods

- Sometimes you might want to see if two spans of time overlap each other in any way
- In Relevant, some Importers are designed to display a time span. This is expressed as fields for a start date and an end date

Example #1: Diagnosis

The Importer diabetes_cases has three fields:

1. patient_id
2. started_on
3. ended_on

The time span between the started_on and ended_on dates represents when the patient had an active diagnosis

Example: Importer diabetes_cases

Typical set-up

- started_on: the earliest date that the patient had the diagnosis (for example, the diagnosis Onset Date or the date it was added to the Problem List)
- ended_on: the Resolved Date if the diagnosis is marked resolved. *Note that it is rare but still possible that a diagnosis gets resolved or otherwise removed from the Problem List. The SQL code should allow an ended_on date to indicate the patient no longer qualified for the diagnosis after that time.*

Example #2: Medication

The Importer antiplatelet_medications has three fields:

1. patient_id
2. started_on
3. ended_on

The time span between the started_on and ended_on dates represents when the patient was using the medication

Example: Importer antiplatelet_medications

Typical set-up

- started_on: the visit date when the medication was started or otherwise prescribed
- ended_on:
  - ✓ In eCW, this is the started_on date plus the “duration” of the prescription (i.e., how many days it lasts)
  - ✓ In NextGen, this is the date the medication was stopped (i.e., the date_stopped field)

Simple: Working With One Date Within a Time-Span

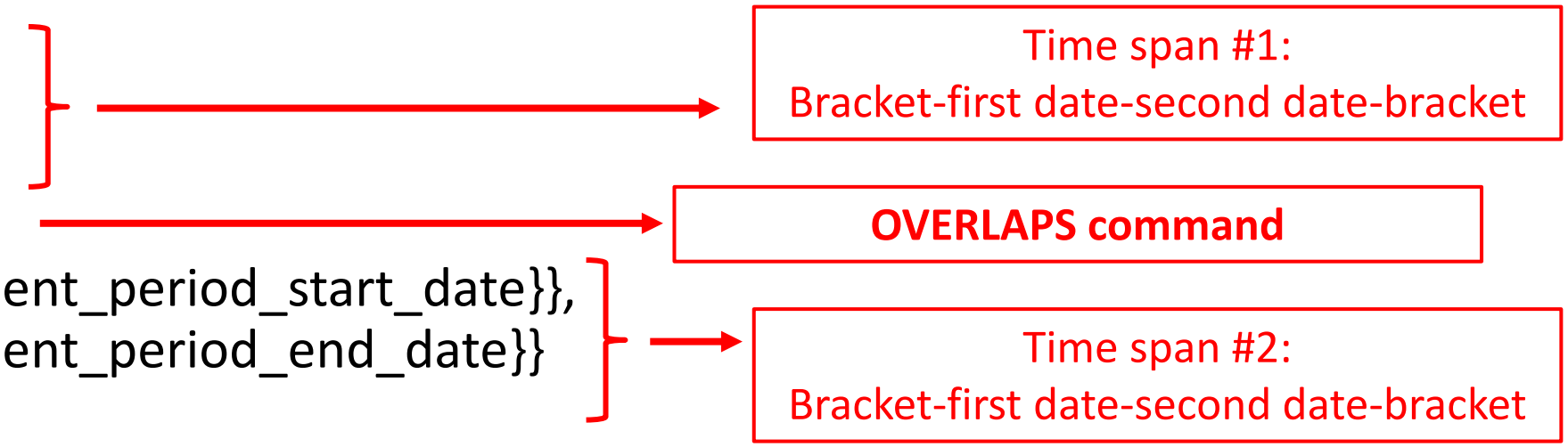
To get a list of patients who are currently on an anti-platelet medication:

```
SELECT DISTINCT patient_id
FROM antiplatelet_medications
WHERE NOW() BETWEEN started_on AND ended_on
```

Assumes that there is a date in the ended_on field of the record

OVERLAPS: Working with a Time-Span Overlapping another Time-Span (i.e., a Measurement Period)

```
SELECT DISTINCT patient_id
FROM antiplatelet_medications
WHERE
  (
    started_on,
    ended_on
  ) OVERLAPS (
    {{measurement_period_start_date}},
    {{measurement_period_end_date}}
  )
```



What You Typically See in Relevant

```
SELECT DISTINCT patient_id
FROM antiplatelet_medications
WHERE
  (
    started_on,
    COALESCE(ended_on + 1, {{measurement_period_end_date}} :: DATE + 1)
  ) OVERLAPS (
    {{measurement_period_start_date}},
    {{measurement_period_end_date}} :: DATE + 1
  )
```

First time span

Second time span

Adapted from SQL code found in the Quality Measure “Ischemic Vascular Disease (IVD): Use of Aspirin or Another Antiplatelet (UDS 2020 Table 6B)”

What You Typically See in Relevant

```
SELECT DISTINCT patient_id
FROM antiplatelet_medications
WHERE
    (
      started_on,
      COALESCE(ended_on + 1, {{measurement_period_end_date}} :: DATE + 1)
    ) OVERLAPS (
      {{measurement_period_start_date}},
      {{measurement_period_end_date}} :: DATE + 1
    )
```

COALESCE in case there is no ended_on date

If no ended_on date, assume it is active at the end of the measurement period

Add a day onto the end of the time spans so that it counts overlap on the last date

Date range definition

PERIOD START DATE <= comparison date < PERIOD END DATE

Says “LESS THAN” and not “LESS THAN OR EQUAL TO”

Adding a day to the end effectively makes this logic into:

PERIOD START DATE <= comparison date <= PERIOD END DATE + 1 day

Not needed for the beginning of the measurement period

Exercise

- You want to examine the PHQ-9 scores in 2020 for patients with depression at the time. You have another query (separate) that pulls the PHQ-9 scores, so you want to JOIN that with a patient-depression query you are making now
- Importer: depression_cases
- Key fields: patient_id, started_on, ended_on
- Output: an unduplicated list of patient IDs from patients who had depression at any time in 2020

Before starting: are there records with an ended_on date equal to NULL?

```
SELECT * FROM depression_cases
WHERE ended_on IS NULL
```

patient_id	started_on	ended_on
1055150	3/1/2018	
1055251	5/20/2019	
1056167	6/7/2018	
1079121	6/7/2018	

Therefore, you need to use the COALESCE command as we saw in a previous slide

Exercise

- You want to examine the PHQ-9 scores in 2020 for patients with depression at the time. You have another query (separate) that pulls the PHQ-9 scores, so you want to JOIN that with a patient-depression query you are making now
- Importer: depression_cases
- Key fields: patient_id, started_on, ended_on
- Output: an unduplicated list of patient IDs from patients who had depression at any time in 2020

```
SELECT DISTINCT patient_id
FROM antiplatelet_medications
WHERE
  (
   started_on,
   COALESCE(ended_on + 1,  {{measurement_period_end_date}} :: DATE + 1)
  ) OVERLAPS (
   {{measurement_period_start_date}},
   {{measurement_period_end_date}} :: DATE + 1
  )
```

ANSWER

```
SELECT DISTINCT patient_id
FROM depression_cases
WHERE
   (
    started_on,
    COALESCE(ended_on + 1,  '2020-12-31' :: DATE + 1)
   ) OVERLAPS (
    '2020-01-01' :: DATE ,
    '2020-12-31' :: DATE + 1
   )
```

NOTE that the OVERLAPS command is very sensitive to date values. Therefore, CAST the field as a date (:: DATE)

Since '2020-12-31' was formatted as a date, the program knows adding a one (+1) means adding one day. Another way of writing this is: + INTERVAL '1 DAY'

Next task: add measurement period parameters: {{measurement_period_start_date}}, {{measurement_period_end_date}}

ANSWER

```
SELECT DISTINCT patient_id, started_on, ended_on
FROM depression_cases
WHERE
   (
    started_on,
    COALESCE(ended_on + 1,  {{measurement_period_end_date}} :: DATE + 1)
   ) OVERLAPS (
    {{measurement_period_start_date}},
    {{measurement_period_end_date}} :: DATE + 1
   )
```

Even though your parameter might have already been formatted as a date field, I have seen a forced date format here anyway in the Relevant code (it does not hurt to add it)

4. Combining Query Output Using a UNION Query

Combining Query Output Using a UNION Query

- This is useful when you are getting the “same” kind of data from multiple tables
- Sometimes you cannot simply JOIN tables together because two or more tables contain the primary data you need

Example: Colorectal Cancer Screens

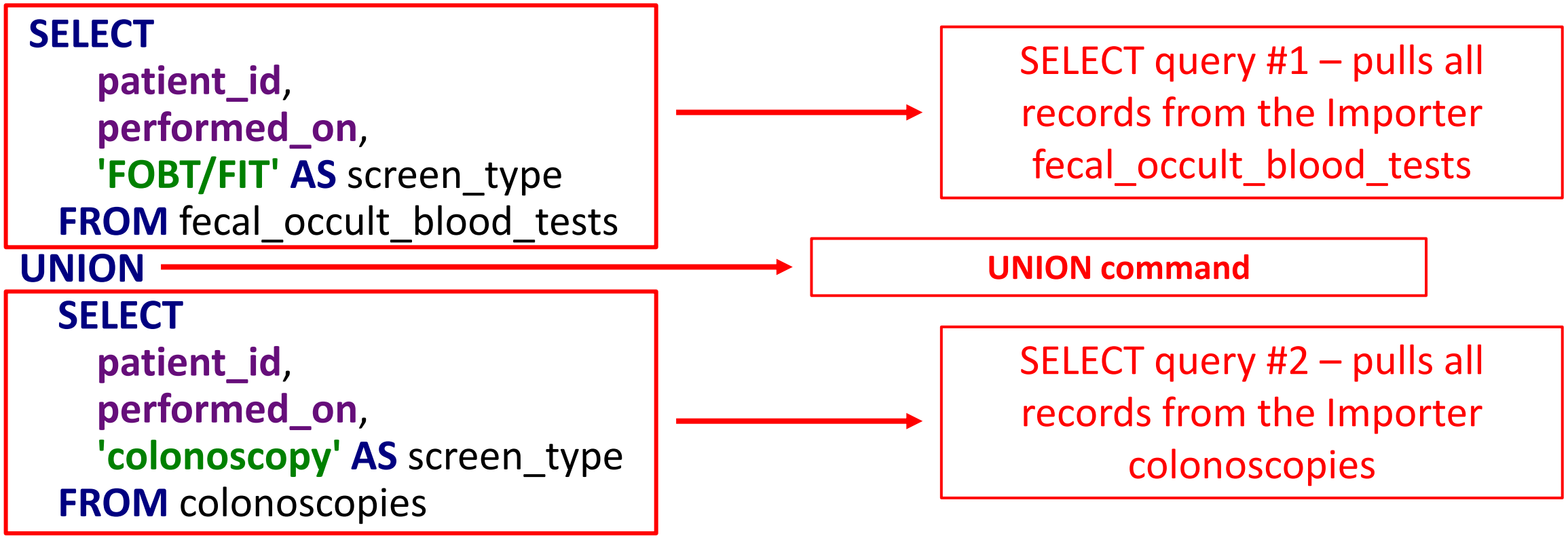
There are five types of colorectal cancer screening activities that come from five Importers:

1. fecal_occult_blood_tests
2. stool_dna_tests
3. sigmoidoscopies
4. colonoscopies
5. ct_colonographies

It would be complicated to try and combine these into one single query using JOINS

Example: Colorectal Cancer Screens

- What you really want is to get the results from each Importer and them combine them together
- The example we will look at will use two of the Importers:
  1. fecal_occult_blood_tests
  2. colonoscopies



Adapted from SQL code found in the Quality Measure “Colorectal Cancer Screening (UDS 2020 Table 6B)”

In this example, we are using the screen_type field so later we can tell the records apart (i.e., what the source was)

```
SELECT
      patient_id,
      performed_on,
      'FOBT/FIT' AS screen_type
  FROM fecal_occult_blood_tests
UNION
  SELECT
      patient_id,
      performed_on,
      'colonoscopy' AS screen_type
  FROM colonoscopies
```

Three columns with exactly the same names

Use a column alias if the column names in the underlying table are not the same

Exercise

- You want a list of all flu and MMR vaccines in 2020 along with all vaccine dates and a way to tell them apart
- Importers: flu_immunizations, mmr_immunizations
- Key fields: patient_id, applied_on
- Output: a list of these two vaccines from 2020 along with patient_id so that it can be JOINED later

Exercise

- You want a list of all flu and MMR vaccines in 2020 along with all vaccine dates and a way to tell them apart
- Importers: flu_immunizations, mmr_immunizations
- Key fields: patient_id, applied_on
- Output: a list of these two vaccines from 2020 along with patient_id so that it can be JOINED later

```
SELECT
    patient_id,
    performed_on,
    'FOBT/FIT' AS screen_type
  FROM fecal_occult_blood_tests
UNION
  SELECT
    patient_id,
    performed_on,
    'colonoscopy' AS screen_type
  FROM colonoscopies
```

ANSWER

```
SELECT
      patient_id,
      applied_on,
      'flu' AS vaccine_type
FROM flu_immunizations
WHERE applied_on BETWEEN '2020-01-01' AND '2020-12-31’


UNION


SELECT
      patient_id,
      applied_on,
      'MMR' AS vaccine_type
FROM mmr_immunizations
WHERE applied_on BETWEEN '2020-01-01' AND '2020-12-31'
```

Other Considerations

- The UNION command automatically removes any duplicate records in the final data set.
- For example, if you did not have the vaccine_type column, there would potentially be many duplicate records with only the patient_id and applied_on columns because many patients probably got both vaccines on the same day
- If you want the duplicates, use UNION ALL command

Other Considerations

- The columns must have the same names
- The columns must the same formatting. For example, you cannot have a column with name ‘column1’ that comes from a date column in your first SELECT query and a text column in your second SELECT query
- The columns must be in the same order
- You can add as many UNION statement as you want

5. Using WITH Versus CREATE TEMPORARY TABLE Commands

Using WITH Versus CREATE TEMPORARY TABLE Commands

- Sometimes it is easier to create a table in a query that temporarily stores data that you can use for a specific purpose
- The table is deleted after use, so it is not something that becomes part of the data model
- In Relevant, TEMPORARY TABLES are commonly used. However, the WITH statement can also be used for the same purpose
- Let’s look at both…

Temporary Table (This is Review)

```
DROP TABLE IF EXISTS numerator;
CREATE TEMPORARY TABLE numerator AS
SELECT DISTINCT ON (patient_id)
    patient_id,
    performed_on
FROM mammograms
WHERE mammograms.performed_on BETWEEN
        DATE {{measurement_period_end_date}} - INTERVAL '27 MONTHS'
        AND DATE {{measurement_period_end_date}}
ORDER BY patient_id, performed_on DESC
```

These commands are necessary to create the Temporary Table

The rest of the code is any kind of SELECT statement

So, instead of displaying the data, this query saves the data in a TEMPORARY TABLE so you can use it later in another SELECT query

WITH Command

```
WITH numerator AS (
    SELECT DISTINCT ON (patient_id)
        patient_id,
        performed_on
    FROM mammograms
    WHERE mammograms.performed_on BETWEEN
        DATE {{measurement_period_end_date}} - INTERVAL '27 MONTHS'
        AND DATE {{measurement_period_end_date}}
    ORDER BY patient_id, performed_on DESC
)
SELECT
    patients.mrn,
    numerator.performed_on AS last_mammogram
FROM patients
    INNER JOIN numerator ON numerator.patient_id = patients.id
```

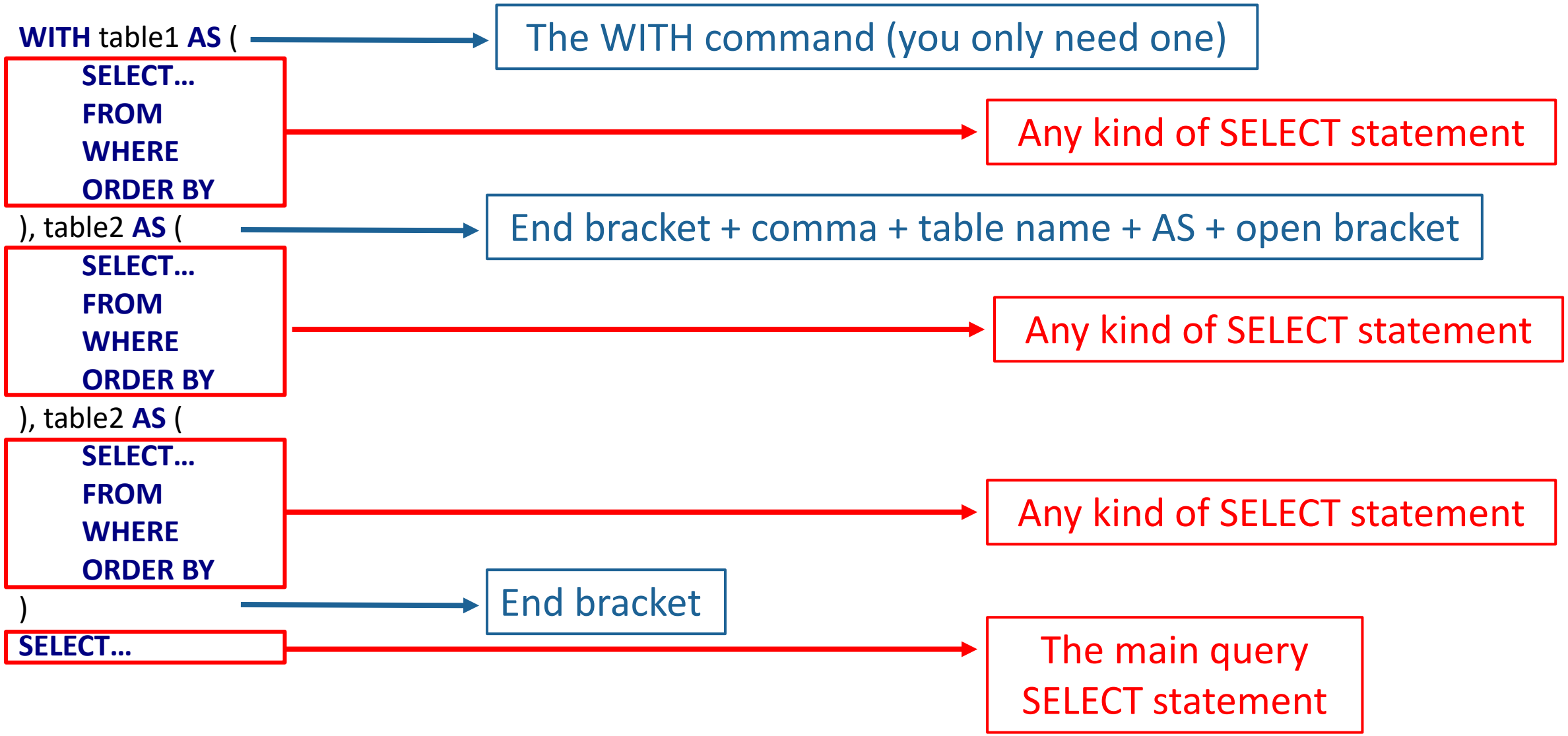
WITH + table name + AS + open bracket

Any kind of SELECT statement

End bracket

The table is available in the SELECT statement that follows only

You can create several tables after the WITH command



Exercise

- You want to list all patients with both flu and MMR vaccines (in history) along with the last vaccine date of each
- Importers: flu_immunizations, mmr_immunizations
- Key fields: patient_id, applied_on
- Output: a list of patients with both of these vaccines along with the patient_id, medical record number, and last vaccine dates

ANSWER

```
WITH last_flu_temp AS (
        SELECT DISTINCT ON (patient_id)
                patient_id,
                applied_on
        FROM flu_immunizations
        ORDER BY patient_id, applied_on DESC
), mmr_temp AS (
        SELECT DISTINCT ON (patient_id)
                patient_id,
                applied_on
        FROM mmr_immunizations
        ORDER BY patient_id, applied_on DESC
)
SELECT
   patients.id AS patient_id,
   last_flu_temp.applied_on AS last_flu_date,
   mmr_temp.applied_on AS last_mmr_date
FROM patients
   INNER JOIN last_flu_temp ON last_flu_temp.patient_id = patients.id
   INNER JOIN mmr_temp ON mmr_temp.patient_id = patients.id
```

6. Check to see if something EXISTS

AS A CONDITION OF THE WHERE STATEMENT

Check to see if something EXISTS

- Commonly found in the WHERE statement
- The core of it is a SELECT query. If the result of the query produces at least one row, the result of EXISTS is TRUE (in other words, it is TRUE that it exists)
- Therefore, in a WHERE statement, the record is chosen if the EXISTS condition is TRUE and the record is not chosen if the EXISTS condition is FALSE
- The content of the SELECT query is not important. What is important is that it returns any row of data

```
DROP TABLE IF EXISTS universe;
CREATE TEMPORARY TABLE universe AS
SELECT patients.id AS patient_id,
       patients.primary_care_giver_id,
       patients.primary_location_id,
       patients.health_center_id
FROM patients
-- age 18+, under 75 at beginning of measurement period
WHERE extract(YEAR FROM age({{measurement_period_start_date}}, patients.date_of_birth)) BETWEEN 18 AND 74
  -- visit in period
  AND EXISTS(
        SELECT
        FROM visits
                 INNER JOIN visit_set_memberships ON visit_set_memberships.visit_id = visits.id
        WHERE visits.patient_id = patients.id
          AND visits.visit_date :: DATE BETWEEN {{measurement_period_start_date}} AND {{measurement_perio
d_end_date}}
          AND visit_set_memberships.standard_visit_set_id = 'uds_medical')
  AND EXISTS(
        SELECT
        FROM diabetes_cases
        WHERE diabetes_cases.patient_id = patients.id
          AND (diabetes_cases.started_on, COALESCE(diabetes_cases.ended_on + 1, {{measurement_period_end_
date}} :: DATE + 1)) OVERLAPS
              ({{measurement_period_start_date}},
               {{measurement_period_end_date}} :: DATE + 1)
    )
```

Copied from the Quality Measure “Diabetes: Hemoglobin A1c Control (<=9%) (UDS 2020 Table 7, inverted)”

The EXISTS command + open bracket

This WHERE condition is key! visits.patient_id must be equal to patients.id in the main query outside of the EXISTS command

Any kind of SELECT statement

```
EXISTS(
    SELECT
    FROM visits
        INNER JOIN visit_set_memberships ON visit_set_memberships.visit_id = visits.id
    WHERE visits.patient_id = patients.id
        AND visits.visit_date :: DATE BETWEEN {{measurement_period_start_date}}
                AND {{measurement_period_end_date}}
        AND visit_set_memberships.standard_visit_set_id = 'uds_medical')
```

Note: some developers use SELECT 1 in the first line after the EXISTS command

```
DROP TABLE IF EXISTS universe;
CREATE TEMPORARY TABLE universe AS
SELECT patients.id AS patient_id,
       patients.primary_care_giver_id,
       patients.primary_location_id,
       patients.health_center_id
FROM patients
-- age 18+, under 75 at beginning of measurement period
WHERE extract(YEAR FROM age({{measurement_period_start_date}}, patients.date_of_birth)) BETWEEN 18 AND 74
  -- visit in period
  AND EXISTS(
        SELECT
        FROM visits
                 INNER JOIN visit_set_memberships ON visit_set_memberships.visit_id = visits.id
        WHERE visits.patient_id = patients.id
          AND visits.visit_date :: DATE BETWEEN {{measurement_period_start_date}} AND {{measurement_perio
d_end_date}}
          AND visit_set_memberships.standard_visit_set_id = 'uds_medical')
  AND EXISTS(
        SELECT
        FROM diabetes_cases
        WHERE diabetes_cases.patient_id = patients.id
          AND (diabetes_cases.started_on, COALESCE(diabetes_cases.ended_on + 1, {{measurement_period_end_
date}} :: DATE + 1)) OVERLAPS
              ({{measurement_period_start_date}},
               {{measurement_period_end_date}} :: DATE + 1)
    )
```

Exercise

- You want a list of patients who were pregnant (i.e., were observed to be pregnant) from January to March 31, 2021
- Importers: pregnancy_observations
- Key fields: patient_id, observed_on
- Output: a list of patients who meet this condition with only patient_id and medical record number displayed

Exercise

- You want a list of patients who were pregnant (i.e., were observed to be pregnant) from January to March 31, 2021
- Importers: pregnancy_observations
- Key fields: patient_id, observed_on
- Output: a list of patients who meet this condition with only patient_id and medical record number displayed

```
EXISTS(
    SELECT
    FROM visits
        INNER JOIN visit_set_memberships ON visit_set_memberships.visit_id = visits.id
    WHERE visits.patient_id = patients.id
        AND visits.visit_date :: DATE BETWEEN {{measurement_period_start_date}}
                AND {{measurement_period_end_date}}
```

ANSWER

```
SELECT
        patients.id,
        patients.mrn
FROM patients
WHERE EXISTS(
        SELECT 1
        FROM pregnancy_observations
        WHERE pregnancy_observations.patient_id = patients.id
                AND observed_on :: DATE BETWEEN '2021-01-01' AND '2021-03-31')
```

Is using EXISTS the only way to logically create this kind of query?

Query Optimization

Choosing the fastest option

- There are different ways to write a query
- Choose the one that works the fastest or uses the least amount of resources
- Some health centers have tables with millions of records (e.g., medicines, vitals, etc.)
- The nightly Relevant Pipeline takes longer and longer to complete as more records are added. Therefore, custom Quality Measures should be made as efficient as possible

WHERE Statement Options

- In the last Exercise, we learned about the EXISTS statement
- There are (at least) three other ways of executing the same logic

EXISTS statement

```
SELECT
        patients.id,
        patients.mrn
FROM patients
WHERE EXISTS(
        SELECT 1
        FROM pregnancy_observations
        WHERE pregnancy_observations.patient_id = patients.id
                AND observed_on :: DATE BETWEEN '2021-01-01' AND '2021-03-31')
```

INNER JOIN

```
SELECT DISTINCT
        patients.id,
        patients.mrn
FROM patients
INNER JOIN pregnancy_observations ON pregnancy_observations.patient_id = patients.id
WHERE observed_on :: DATE BETWEEN '2021-01-01' AND '2021-03-31'
```

Using an IN Clause

```
SELECT
        patients.id,
        patients.mrn
FROM patients
WHERE patients.id IN(SELECT patient_id
                     FROM pregnancy_observations
                     WHERE observed_on :: DATE BETWEEN '2021-01-01' AND '2021-03-31')
```

Using an ANY Clause

```
SELECT
        patients.id,
        patients.mrn
FROM patients
WHERE patients.id = ANY(SELECT patient_id
                        FROM pregnancy_observations
                        WHERE observed_on :: DATE BETWEEN '2021-01-01' AND '2021-03-31')
```

So, which is the best?

- Use **EXPLAIN ANALYSE** in DataGrip before the SELECT statement in order to get some statistics (next slide has an example)
- I ran all of them out of the RCHC Production database to process the most number of records
- Not very scientific, but the IN and ANY methods worked the fastest. INNER JOIN was the longest. All of them executed in under a second, so the observed differences may not be significant

```
74 ✔ EXPLAIN ANALYSE SELECT
75         patients.id,
76         patients.mrn
77   FROM patients
78   WHERE patients.id = ANY(SELECT patient_id
79   FROM pregnancy_observations
80       WHERE observed_on :: DATE BETWEEN '2021-01-01' AND '2021-03-31');
```

Output | Result 4

12 rows

	QUERY PLAN
1	Hash Join (cost=14282.76..51424.56 rows=16748 width=11) (actual time=68.871..421.876 rows=2552 loops=1)
2	Hash Cond: (patients.id = pregnancy_observations.patient_id)
3	-> Seq Scan on patients (cost=0.00..33122.06 rows=1037106 width=11) (actual time=0.008..193.617 rows=1037106 loops=1)
4	-> Hash (cost=14119.51..14119.51 rows=13060 width=4) (actual time=68.773..68.773 rows=2553 loops=1)
5	Buckets: 16384 Batches: 1 Memory Usage: 218kB
6	-> HashAggregate (cost=13988.91..14119.51 rows=13060 width=4) (actual time=67.691..68.307 rows=2553 loops=1)
7	Group Key: pregnancy_observations.patient_id
8	-> Seq Scan on pregnancy_observations (cost=0.00..13947.03 rows=16748 width=4) (actual time=0.011..63.198 rows=16893 loops=1)
9	Filter: ((observed_on >= '2021-01-01'::date) AND (observed_on <= '2021-03-31'::date))
10	Rows Removed by Filter: 666576
11	Planning time: 0.343 ms
12	Execution time: 422.238 ms

Use an Index

- Like an index at the back of a book you can use to quickly browse topics from a list. Or, like the Dewey Decimal System at a library
- Commonly used in Relevant
- Normally applied to TEMPORARY TABLES
- Can be seen in virtually all the Quality Measures because those handle large numbers of records
- An Index is optional and is not recommended for small tables

```
DROP TABLE IF EXISTS universe;
CREATE TEMPORARY TABLE universe AS
SELECT
        patients.id AS patient_id,
        patients.primary_care_giver_id AS provider_id,
        patients.primary_location_id   AS location_id,
        patients.health_center_id
FROM patients
WHERE EXISTS(
                SELECT
                FROM visits
                    INNER JOIN visit_set_memberships ON visits.id = visit_set_memberships.visit_id
                WHERE visits.visit_date :: DATE BETWEEN {{measurement_period_start_date}} AND
{{measurement_period_end_date}}
                    AND visit_set_memberships.standard_visit_set_id = 'uds_medical'
                    AND visits.patient_id = patients.id);
CREATE INDEX index_universe_on_patient_id ON universe (patient_id);
```

CREATE INDEX command + unique index name + ON + TEMPORARY TABLE name + index field

CREATE INDEX index_universe_on_patient_id **ON** universe (**patient_id**);

The CREATE INDEX command is separate from the SELECT statement. Therefore, there must be a semi-colon at the end of the SLECT statement right before the CREATE INDEX command and there must be a semi-colon after the whole CREATE INDEX statement

The index name can be anything, but should be unique. If it is not unique, you will get an error. Relevant uses the convention of "index" + table name + field name all connected by underscores

Table name

Field name, as it appears in the output. It should be the field that is normally used for look-up (like a "key" field)

```
CREATE TEMPORARY TABLE universe AS
```

```
SELECT patients.id AS patient_id,
```

```
    AND visits.patient_id = patients.id);
CREATE INDEX index_universe_on_patient_id ON universe (patient_id);
```

Record Efficiency

- Make the query process the least number of records possible. It takes processing time to find and analyze records, copy records into a TEMP TABLE, add an index, etc.
- For example, if you have a “universe” TEMPORARY TABLE which defines the measure denominator (even if it is an initial denominator), it can be used with an INNER JOIN on other tables

```
DROP TABLE IF EXISTS blood_pressure_reading_temp;
CREATE TEMPORARY TABLE blood_pressure_reading_temp AS
SELECT
    patient_id,
    performed_on,
    MIN(systolic_blood_pressure)  AS systolic_blood_pressure,
    MIN(diastolic_blood_pressure) AS diastolic_blood_pressure
FROM blood_pressure_readings;
```

```
DROP TABLE IF EXISTS blood_pressure_reading_temp;
CREATE TEMPORARY TABLE blood_pressure_reading_temp AS
SELECT
    blood_pressure_readings.patient_id,
    performed_on,
    MIN(systolic_blood_pressure)  AS systolic_blood_pressure,
    MIN(diastolic_blood_pressure) AS diastolic_blood_pressure
FROM blood_pressure_readings
INNER JOIN universe ON universe.patient_id
      = blood_pressure_readings.patient_id;
```

Record Efficiency

- You may encounter a case where you need to extract different kinds of records from the same table, and you need to do it several times.
- For example, in eCW Relevant, the table obf_pregnancy_data contains the delivery date, pregnancy outcome, and birth weight. The type of data is identified by item IDs
- Is there a scenario that is most efficient?

Let’s say that your health center has over 4 million records on obf_pregnancy_data and you need to make three separate TEMPORARY TABLES.

CREATE TEMPORARY TABLES as…

Delivery date

```
SELECT pregid, upd_date, choice
FROM obf_pregnancy_data
WHERE itemid = 475138;
```

Table 1

Pregnancy outcome

```
SELECT pregid, upd_date, choice
FROM obf_pregnancy_data
WHERE itemid = 412965;
```

Table 2

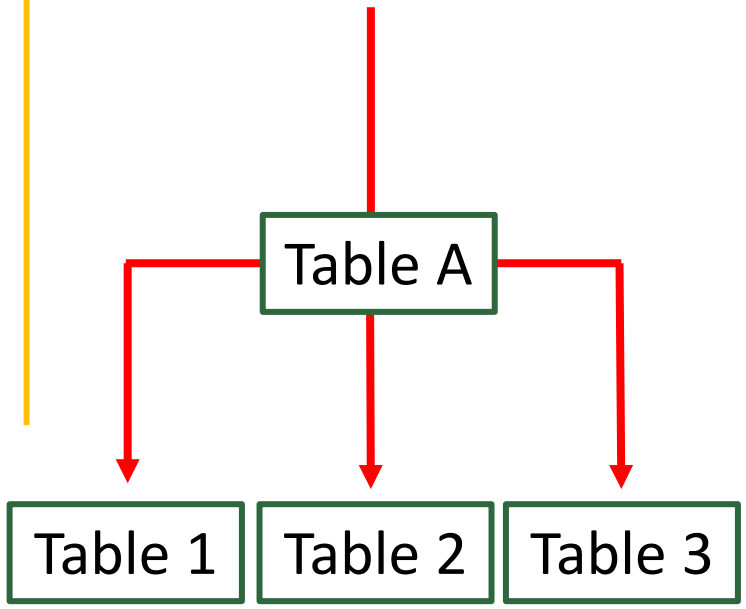
Birth weight

```
SELECT pregid, upd_date, choice
FROM obf_pregnancy_data
WHERE itemid = 494734;
```

Table 3

--OR-- All three, then split

```
SELECT pregid, upd_date, choice
FROM obf_pregnancy_data
WHERE itemid IN(412965,475138,494734);
```



The result is the same three tables with the same data in both cases

Look at your Data Pipeline

- Which Transformers and Importers are taking an exceptionally ling time to run
- You probably do not want to mess around with any that Relevant made, but are there custom ones that can be made more efficient?
- Queries for medications, visits, claims (and others) take a long time because those tables are huge

Quality Reports Operations Finance **Data Pipeline**

Overview Source Databases Acquisition Plans **Transformers** Importers Populations Risk Models Care Gaps Measures

Data Pipeline: Transformers

Status: Disabled Enabled All

Drag to re-order

+ New Transformer

Transformer	Last run ended	Last run took		
Create relevant_claims	Thursday, 05/06/21, 1:06 AM	**11 mins 47 secs**	✓	Run
Create relevant_appointments	Thursday, 05/06/21, 1:06 AM	14 secs	✓	Run
Create relevant_chronicpain	Thursday, 05/06/21, 1:06 AM	less than 1 sec	✓	Run
Build relevant_stool_dna_tests	Thursday, 05/06/21, 1:14 AM	**2 mins 55 secs**	✓	Run
Build lead_labs	Thursday, 05/06/21, 2:56 AM	**9 mins 21 secs**	✓	Run
Create relevant_last_appointment	Thursday, 05/06/21, 12:50 AM	20 secs	✓	Run
Create relevant_insurance_enrollments	Thursday, 05/06/21, 2:56 AM	4 secs	✓	Run
Build relevant_ace_inhibitor_medications and relevant_arb_medications	Thursday, 05/06/21, 2:55 AM	**9 mins 13 secs**	✓	Run

Questions?