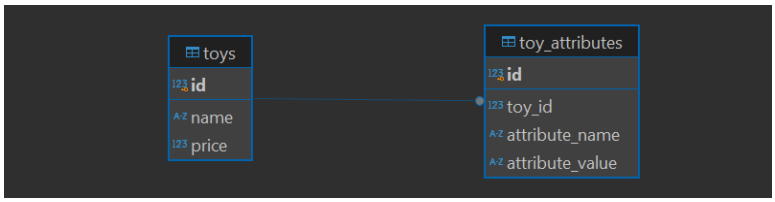
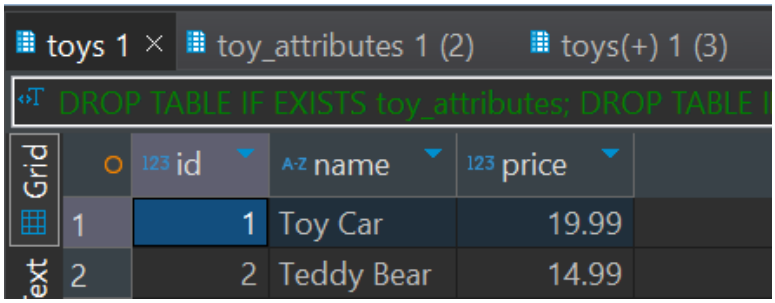
DATABASE 2 – LABS 1-5 Documentation

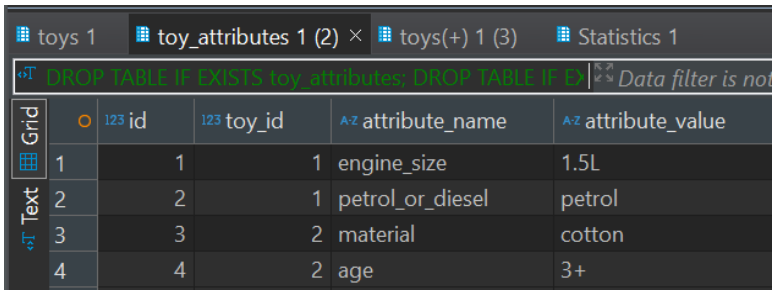
Student Name: Christopher Noblett *Student Number: C22454222*

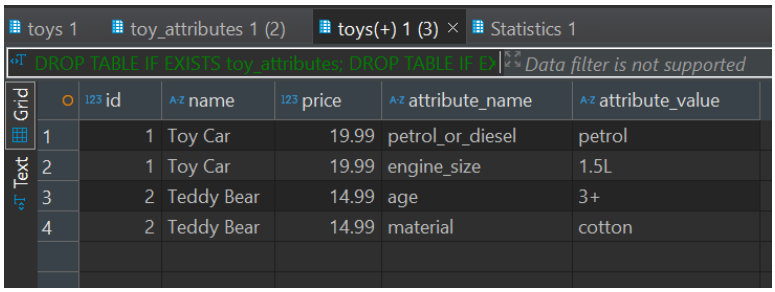
LAB 1

Exercise 1









1. Toys and Toy Attributes Tables

Toys Table

- Design Choice: The toys table serves as the primary entity, representing the main product. Using SERIAL for the id allows for automatic, unique identification of each toy, simplifying data entry and management. The attributes name and price are set to NOT NULL, ensuring that every toy has a name and price, which are essential for any retail application.

Toy Attributes Table

- Design Choice: The toy_attributes table is designed to store specific features of toys, enabling a flexible structure that accommodates varying attributes across different toys. The combination of toy_id and attribute_name as a unique constraint ensures that each toy can have unique attribute names, preventing duplicate entries and maintaining data integrity. The foreign key relationship with toys(id) allows for cascading deletes, ensuring that if a toy is deleted, its attributes are also removed, thus avoiding orphan records.

Data Insertion Choices

- Design Choice: The sample data inserted into both tables reflects realistic toy products and their attributes. This approach demonstrates how the tables function together, showcasing their capability to handle diverse toy features without altering the core structure.

Query Design Choices

- Design Choice: The selection queries (SELECT * FROM toys; and SELECT * FROM toy_attributes;) are straightforward and effective for displaying data. The join query (LEFT JOIN) demonstrates the relationship between toys and their attributes, allowing for comprehensive data retrieval that enhances user experience when searching for toys and their features.

Exercise 2


```
-- Create table for E2 (the "many" side of the relationship)
CREATE TABLE E2 (
  K2 INT PRIMARY KEY -- Primary key for E2
);
-- Create table for E1 (the "one" side of the relationship)
CREATE TABLE E1 (
  K1 INT PRIMARY KEY, -- Primary key for E1
  K2 INT NOT NULL, -- Foreign key referencing E2
  CONSTRAINT fk_e2 FOREIGN KEY (K2) REFERENCES E2(K2) ON DELETE CASCADE
);
```

Table Design for E1 and E2

1. E2 Table
    - Design Choice: The E2 table is created to represent the "many" side of the relationship. It has a primary key K2, which uniquely identifies each record in the table. This structure allows multiple records in E1 to reference a single record in E2, establishing the basis for the one-to-many relationship.
2. E1 Table
    - Design Choice: The E1 table represents the "one" side of the relationship. It has a primary key K1 for unique identification of each record and a foreign key K2 that references the K2 primary key in the E2 table. The inclusion of K2 as a foreign key enforces referential integrity, ensuring that each record in E1 must relate to a valid record in E2.

- ON DELETE CASCADE: This constraint ensures that if a record in E2 is deleted, all corresponding records in E1 will also be automatically deleted. This design choice prevents orphaned records in E1 and maintains data consistency across the tables.


```
-- Create table for E3 (the "(0,1)" side of the relationship)
CREATE TABLE E3 (
  K3 INT PRIMARY KEY -- Primary key for E3
);
-- Create table for E4 (the "(0,*)" side of the relationship)
CREATE TABLE E4 (
  K4 INT PRIMARY KEY, -- Primary key for E4
  K3 INT, -- Foreign key referencing E3, can be NULL
  CONSTRAINT fk_e3 FOREIGN KEY (K3) REFERENCES E3(K3) ON DELETE SET NULL
);
```

Table Design for E3 and E4

1. E3 Table
    - Design Choice: The E3 table represents the "(0,1)" side of the relationship, meaning that there can be either zero or one record in E3 related to each record in E4. It contains a primary key K3 that uniquely identifies each record in the table. This structure allows for flexibility in relationships, indicating that not all records in E4 need to reference a record in E3.
2. E4 Table
    - Design Choice: The E4 table represents the "(0,*)" side of the relationship, allowing for zero or more records in E4 that can reference a single record in E3. It has a primary key K4 for unique identification and a foreign key K3 that references E3. The K3 field can be NULL, which aligns with the "(0,1)" relationship, allowing records in E4 to exist without a corresponding record in E3.
    - ON DELETE SET NULL: This foreign key constraint specifies that if a referenced record in E3 is deleted, the K3 field in the corresponding records in E4 will be set to NULL rather than deleting the records in E4. This design choice preserves the existence of E4 records while indicating the absence of the relationship to E3, thus maintaining data integrity.


```
-- Create table for E5 (one side of the relationship)
CREATE TABLE E5 (
  K5 INT PRIMARY KEY -- Primary key for E5
);
-- Create table for E6 (the other side of the relationship)
CREATE TABLE E6 (
  K6 INT PRIMARY KEY -- Primary key for E6
);
-- Create a junction table to model the many-to-many relationship
CREATE TABLE E5_E6_Relationship (
```

```
  E5_id INT, -- Foreign key referencing E5
  E6_id INT, -- Foreign key referencing E6
  PRIMARY KEY (E5_id, E6_id), -- Composite primary key
  CONSTRAINT fk_e5 FOREIGN KEY (E5_id) REFERENCES E5(K5) ON DELETE CASCADE,
  CONSTRAINT fk_e6 FOREIGN KEY (E6_id) REFERENCES E6(K6) ON DELETE CASCADE
);
```

Table Design for E5, E6, and E5_E6_Relationship

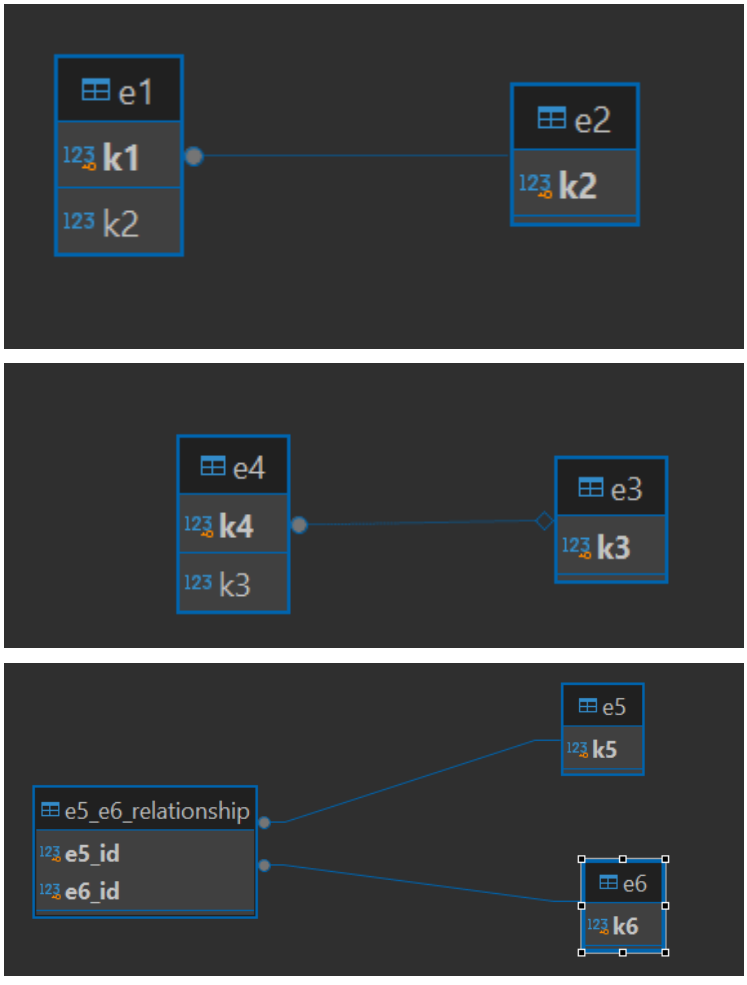
1. E5 Table
    - Design Choice: The E5 table serves as one side of the relationship and contains a primary key K5, which uniquely identifies each record in the table. This design choice establishes the entity E5 as a distinct entity that can participate in a many-to-many relationship with E6.
2. E6 Table
    - Design Choice: Similar to E5, the E6 table serves as the other side of the relationship and contains a primary key K6. This allows for the unique identification of each record in E6. Both E5 and E6 can have multiple records, which is essential for the many-to-many relationship.
3. E5_E6_Relationship Junction Table
    - Design Choice: The E5_E6_Relationship table is a junction (or associative) table designed to model the many-to-many relationship between E5 and E6. This table includes:
        - Foreign Keys: E5_id references K5 from E5, and E6_id references K6 from E6, establishing the connections between the two entities.
        - Composite Primary Key: The primary key consists of both E5_id and E6_id, ensuring that each combination of E5 and E6 is unique. This prevents duplicate relationships from being created.
        - ON DELETE CASCADE: Both foreign key constraints specify ON DELETE CASCADE, meaning that if a record in E5 or E6 is deleted, any corresponding records in the E5_E6_Relationship table will also be deleted automatically. This design choice maintains referential integrity by ensuring that relationships do not exist without their associated entities.


```
-- Create table for E8
CREATE TABLE E8 (
  K8 INT PRIMARY KEY -- Primary key for E8
);
-- Create table for E7 (the child in the one-to-one side of the relationship)
CREATE TABLE E7 (
  K7 INT PRIMARY KEY, -- Primary key for E7
  K8 INT NOT NULL, -- Foreign key referencing E8, must be NOT NULL to enforce the (1,1) constraint
  CONSTRAINT fk_e8 FOREIGN KEY (K8) REFERENCES E8(K8) ON DELETE CASCADE
);
```

Table Design for E7 and E8

1. E8 Table
    - Design Choice: The E8 table contains a primary key K8 that uniquely identifies each record in the table. This establishes E8 as a distinct entity that can be referenced by other entities. Since it’s the "one" side of a one-to-one relationship, its primary key will correspond directly to a record in the E7 table.

2. E7 Table
    - Design Choice: The E7 table represents the "one" side of the one-to-one relationship, with a primary key K7 that uniquely identifies each record. The key design choice here is the inclusion of the foreign key K8, which references the primary key of the E8 table. This enforces the one-to-one relationship between E7 and E8.
        - NOT NULL Constraint: The foreign key K8 is declared as NOT NULL, ensuring that each record in E7 must correspond to an existing record in E8. This enforces the (1,1) constraint of the relationship, meaning that for each record in E7, there must be a corresponding record in E8, and vice versa.
        - Foreign Key Constraint: The foreign key constraint (fk_e8) references K8 in the E8 table with an ON DELETE CASCADE action. This means that if a record in E8 is deleted, the corresponding record in E7 will also be automatically deleted. This design choice maintains referential integrity and prevents orphaned records in E7.



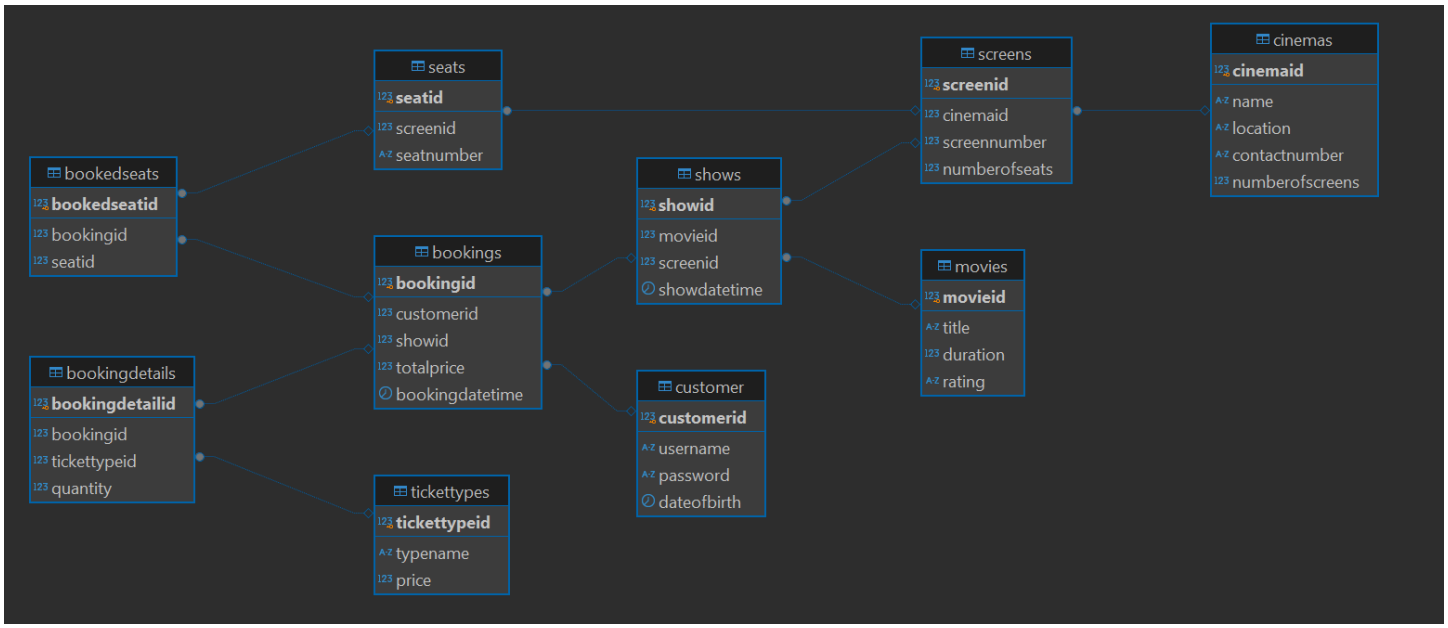
Relational Diagram Overview

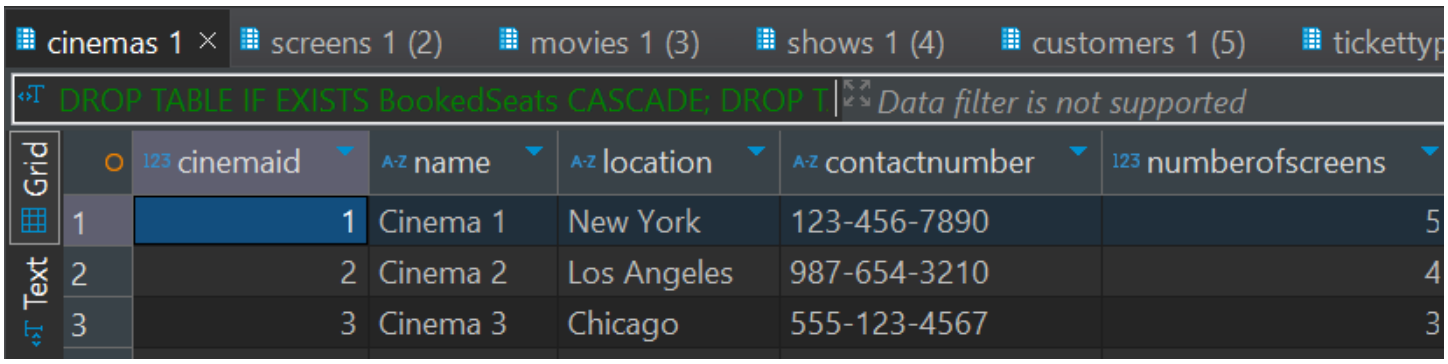
1. Entities and Attributes:
    - E1: Contains K1 (Primary Key) and K2 (Foreign Key referencing E2).
    - E2: Contains K2 (Primary Key).
    - E3: Contains K3 (Primary Key).
    - E4: Contains K4 (Primary Key) and K3 (Foreign Key referencing E3).
    - E5: Contains K5 (Primary Key).
    - E6: Contains K6 (Primary Key).
    - E7: Contains K7 (Primary Key) and K8 (Foreign Key referencing E8).
    - E8: Contains K8 (Primary Key).

Relationships:

1. E1 and E2:
    - Type: One-to-Many
    - Description: E1 can have multiple entries corresponding to a single entry in E2. The foreign key K2 in E1 references K2 in E2.
2. E3 and E4:
    - Type: One-to-Zero-or-Many
    - Description: Each entry in E3 can correspond to zero or more entries in E4. The foreign key K3 in E4 references K3 in E3, and it can be NULL, indicating that the relationship is optional.
3. E5 and E6:
    - Type: Many-to-Many
    - Description: There is a many-to-many relationship between E5 and E6. This is modeled using a junction table called E5_E6_Relationship, which contains foreign keys E5_id (referencing E5) and E6_id (referencing E6). The composite primary key ensures that each pair of E5 and E6 can be uniquely identified.
4. E7 and E8:
    - Type: One-to-One
    - Description: Each entry in E7 corresponds to exactly one entry in E8, and vice versa. The foreign key K8 in E7 references the primary key K8 in E8, enforcing the one-to-one constraint. The ON DELETE CASCADE action ensures referential integrity.

Exercise 3





	cinemaid	name	location	contactnumber	numberofscreens
1	1	Cinema 1	New York	123-456-7890	5
2	2	Cinema 2	Los Angeles	987-654-3210	4
3	3	Cinema 3	Chicago	555-123-4567	3

cinemas 1 | screens 1 (2) | movies 1 (3) | shows 1 (4) | customers 1 (5)

```
DROP TABLE IF EXISTS BookedSeats CASCADE; DROP T
```

Data filter is not supported

Grid | Text

	screenid	cinemaid	screennumber	numberofseats
1	1	1	1	200
2	2	1	2	250
3	3	1	3	300
4	4	1	4	200
5	5	1	5	250
6	6	2	1	250
7	7	2	2	300
8	8	2	3	200
9	9	2	4	250
10	10	3	1	200
11	11	3	2	250
12	12	3	3	300

cinemas 1 | screens 1 (2) | movies 1 (3) | shows 1 (4)

```
DROP TABLE IF EXISTS BookedSeats CASCADE; DROP T
```

Data fil

Grid | Text

	movieid	title	duration	rating
1	1	Movie 1	120	PG-13
2	2	Movie 2	150	R
3	3	Movie 3	90	PG

cinemas 1 | screens 1 (2) | movies 1 (3) | shows 1 (4) | customers

```
DROP TABLE IF EXISTS BookedSeats CASCADE; DROP T
```

Data filter is not supp

Text

	showid	movieid	screenid	showdatetime
1	1	1	1	2024-07-26 10:00:00.000
2	2	1	2	2024-07-26 13:00:00.000
3	3	1	3	2024-07-26 16:00:00.000
4	4	2	4	2024-07-26 19:00:00.000
5	5	2	5	2024-07-26 22:00:00.000
6	6	3	6	2024-07-26 10:00:00.000
7	7	3	7	2024-07-26 13:00:00.000
8	8	3	8	2024-07-26 16:00:00.000

cinemas 1 | screens 1 (2) | movies 1 (3) | shows 1 (4) | customers 1 (5)

```
DROP TABLE IF EXISTS BookedSeats CASCADE; DROP T
```

Data filter is not supported

Grid

Text

	customerid	username	password	dateofbirth
1	1	user1	password1	1990-01-01
2	2	user2	password2	1995-01-01
3	3	user3	password3	2000-01-01

cinemas 1 | screens 1 (2) | movies 1 (3) | shows

```
DROP TABLE IF EXISTS BookedSeats CASCADE; DROP T
```

Grid

Text

	tickettypeid	typename	price
1	1	Adult	15
2	2	Child	10
3	3	Senior	12

cinemas 1 | screens 1 (2) | movies 1 (3) | shows 1 (4) | customers 1 (5) | tickettypes 1 (6) | bookings 1 (7)

```
DROP TABLE IF EXISTS BookedSeats CASCADE; DROP T
```

Data filter is not supported

Grid | Text

	bookingid	customerid	showid	totalprice	bookingdatetime
1	1	1	1	30	2024-07-25 10:00:00.000
2	2	1	2	30	2024-07-25 13:00:00.000
3	3	2	3	20	2024-07-25 16:00:00.000
4	4	3	4	45	2024-07-25 19:00:00.000

cinemas 1 | screens 1 (2) | movies 1 (3) | shows 1 (4) | customer 1 (5)

```
DROP TABLE IF EXISTS BookedSeats CASCADE; DROP T
```

Data filter is not supported

Grid | Text

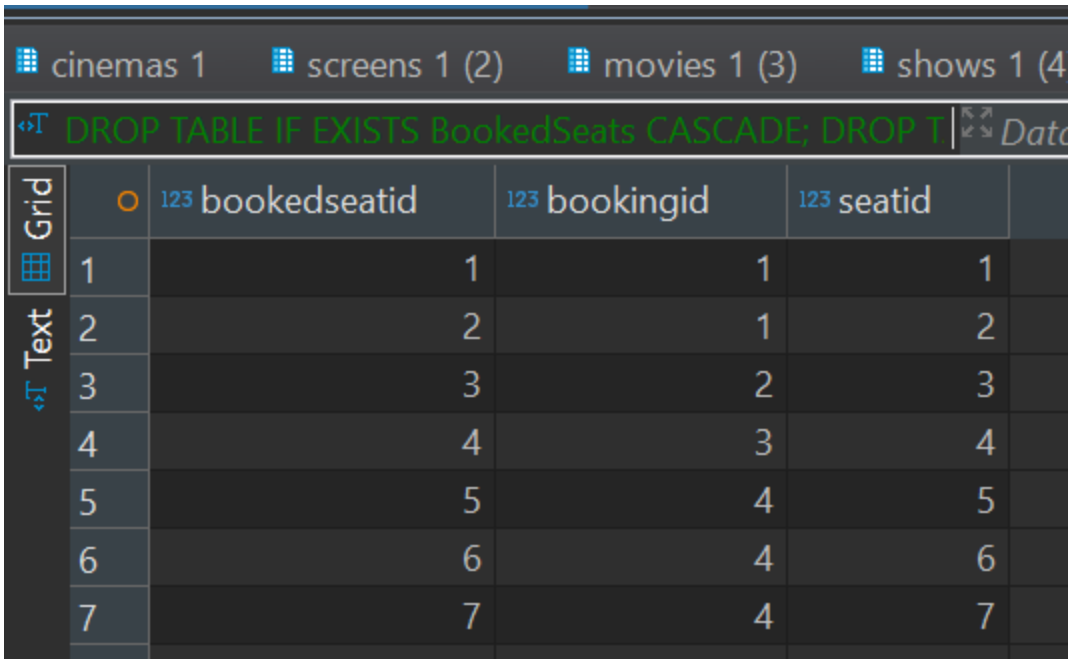
	bookingdetailid	bookingid	tickettypeid	quantity
1	1	1	1	2
2	2	2	2	1
3	3	3	3	1
4	4	4	1	3

cinemas 1 | screens 1 (2) | movies 1 (3) | shows 1 (4)

```
DROP TABLE IF EXISTS BookedSeats CASCADE; DROP T
```

Grid | Text

	seatid	screenid	seatnumber
1	1	1	A1
2	2	1	A2
3	3	1	B1
4	4	1	B2
5	5	2	A1
6	6	2	A2
7	7	2	B1
8	8	2	B2
9	9	3	A1
10	10	3	A2
11	11	3	B1
12	12	3	B2



	bookedseatid	bookingid	seatid
1	1	1	1
2	2	1	2
3	3	2	3
4	4	3	4
5	5	4	5
6	6	4	6
7	7	4	7

1. Cinemas Table

- Design Choice: The Cinemas table includes essential details such as name, location, contact number, and the number of screens. This design ensures that each cinema can be easily identified and contacted. The use of SERIAL for CinemaID allows for automatic and unique identification of each cinema without manual input errors.

2. Screens Table

- Design Choice: The Screens table is designed to relate directly to the Cinemas table through a foreign key (CinemaID). This relationship maintains data integrity and ensures that each screen is correctly associated with its respective cinema. The inclusion of both ScreenNumber and NumberOfSeats allows for effective management of screen resources and provides users with necessary information about seating capacity.

3. Movies Table

- Design Choice: The Movies table captures fundamental movie information, ensuring that critical attributes like title, duration, and rating are easily accessible. This design focuses on the need for comprehensive movie listings while ensuring that each movie has a unique identifier (MovieID), allowing for efficient queries and updates.

4. Shows Table

- Design Choice: The Shows table is crafted to connect both movies and screens, establishing a relationship that allows for detailed scheduling of showtimes. By incorporating foreign keys for MovieID and ScreenID, the design ensures that each show can be traced back to its respective movie and screen, thus maintaining referential integrity across related entities.

5. Customers Table

- Design Choice: The Customers table utilizes a unique constraint on the Username field to prevent duplicate accounts and ensure individual user identification. Storing password data as VARCHAR(255) allows for the

potential use of hashed passwords, enhancing security. The inclusion of DateOfBirth provides valuable demographic data for customer analysis.

6. TicketTypes Table

- Design Choice: The TicketTypes table is designed to categorize different ticket types and their prices, which facilitates flexible pricing strategies. By making TypeName and Price required fields, the design ensures that every ticket type is well-defined and valuable for both customers and the system's revenue management.

7. Bookings Table

- Design Choice: The Bookings table serves as a central link between customers and their show reservations. It uses foreign keys to reference CustomerID and ShowID, ensuring that every booking entry is connected to a specific customer and a specific show. Storing TotalPrice and BookingDateTime allows for easy tracking of transactions and facilitates reporting features.

8. BookingDetails Table

- Design Choice: The design of the BookingDetails table allows for detailed tracking of each booking transaction by linking to the Bookings and TicketTypes tables. This allows for the recording of quantities of each ticket type purchased, providing a granular view of sales which is critical for inventory management and sales analysis.

9. Seats Table

- Design Choice: The Seats table directly associates each seat with a screen, facilitating the management of seating arrangements. By using a foreign key to link to the Screens table, the design ensures that all seat entries are logically organized and that queries about seat availability can be performed efficiently.

10. BookedSeats Table

- Design Choice: The BookedSeats table captures the relationship between booked tickets and the seats assigned to those bookings. By linking BookingID and SeatID with foreign keys, the design effectively manages the many-to-many relationship between bookings and seats, ensuring that the system can accurately track which seats are booked for which shows while maintaining data integrity through cascading actions on deletions.

LAB 2

Exercise 1

```
-- "BUILDER".event_rankings source



CREATE OR REPLACE VIEW "BUILDER".event_rankings

AS SELECT participant_name,

   event_name,

   result,
```

```
  dense_rank() OVER (PARTITION BY event_name ORDER BY (
    CASE
      WHEN result_type::text = 'TIME'::text THEN result::double precision
      WHEN result_type::text = 'DISTANCE'::text THEN result * '-1'::integer::double precision
      ELSE NULL::double precision
    END)) AS rank
FROM paris;
```

This view ranks participants within each event based on their performance, accounting for both time and distance events:

- DENSE_RANK() assigns ranks, allowing ties to share the same rank without skipping ranks afterward.
- Partitioning by event_name ensures that rankings are independent within each event.
- Conditional CASE Expression handles different result_types:
    - For TIME events, results are ranked in ascending order (shorter times rank higher).
    - For DISTANCE events, results are ranked in descending order by multiplying by -1, so longer distances get a higher rank.
- Type Casting to double precision ensures consistent numerical precision, especially for any decimal values.

```
-- "BUILDER".top_5_irish_athletes source


CREATE OR REPLACE VIEW "BUILDER".top_5_irish_athletes
AS ( SELECT paris."da e",
    paris.event_name,
    paris.gender,
    paris.discipline_name,
    paris.participant_name,
    paris.participant_type,
    paris.participant_country_code,
    paris.participant_country,
    paris.result,
```

```
   paris.result_type
  FROM paris
 WHERE paris.participant_country_code::text = 'IRL'::text AND paris.result_type::text = 'DISTANCE'::text
 ORDER BY paris.result
 LIMIT 5)
UNION ALL
( SELECT paris."da e",
   paris.event_name,
   paris.gender,
   paris.discipline_name,
   paris.participant_name,
   paris.participant_type,
   paris.participant_country_code,
   paris.participant_country,
   paris.result,
   paris.result_type
  FROM paris
 WHERE paris.participant_country_code::text = 'IRL'::text AND paris.result_type::text = 'TIME'::text
 ORDER BY paris.result DESC
 LIMIT 5);
```

This view returns the top 5 Irish athletes by performance, separating distance and time events and adjusting the order accordingly:

- Filtering on participant_country_code limits results to Irish athletes ('IRL').

- Two ORDER-BY Limitations:
    - For DISTANCE events, it orders results from smallest to largest, capturing the top 5 best distances (greater distances would be ideal here).
    - For TIME events, it orders results in descending order to prioritize faster times as top performances.
- Union Operation combines these two result sets, providing the top 5 athletes for both types of events in one view.

This view makes it easy to identify top-performing Irish athletes across different scoring systems, providing a straightforward way to retrieve high-ranking athletes by event type.

```
CREATE OR REPLACE VIEW "BUILDER".woman_marathon_winner AS

WITH woman_marathon_winner AS (

   SELECT event_rankings_1.result

   FROM event_rankings event_rankings_1

   WHERE event_rankings_1.event_name::text = 'Women"s Marathon'::text

   ORDER BY event_rankings_1.result

   LIMIT 1

)

SELECT count(*) + 1 AS "position"

FROM event_rankings

WHERE event_name::text = 'Men"s Marathon'::text

AND result < (

   SELECT woman_marathon_winner.result

   FROM woman_marathon_winner

);
```

This view determines how the winner of the Women's Marathon would place relative to male marathon competitors:

1. Subquery for Winning Women's Time: Selects the top result for the Women's Marathon by ordering ascending (shortest time) and limiting to 1.
2. Position Calculation in Men's Marathon:
    - Compare the winning women's time to all male participants in the Men's Marathon.
    - Counts how many male participants had faster results and adds 1 to determine the hypothetical position of the women's winner among male competitors.

```
-- "BUILDER".medal_table source

CREATE OR REPLACE VIEW "BUILDER".medal_table

AS WITH event_rankings AS (

        SELECT paris.participant_name,

          paris.participant_country,

          paris.event_name,

          paris.result,

          paris.result_type,

          dense_rank() OVER (PARTITION BY paris.event_name ORDER BY (

            CASE

              WHEN paris.result_type::text = 'TIME'::text THEN paris.result

              WHEN paris.result_type::text = 'DISTANCE'::text THEN - paris.result

              ELSE NULL::real

            END)) AS ranking

         FROM paris

       )

 SELECT participant_country,

   count(

     CASE

       WHEN ranking = 1 THEN 1
```

```
        ELSE NULL::integer
    END) AS gold,
  count(
    CASE
        WHEN ranking = 2 THEN 1
        ELSE NULL::integer
    END) AS silver,
  count(
    CASE
        WHEN ranking = 3 THEN 1
        ELSE NULL::integer
    END) AS bronze
 FROM event_rankings
GROUP BY participant_country
ORDER BY (count(
    CASE
        WHEN ranking = 1 THEN 1
        ELSE NULL::integer
    END)) DESC, (count(
    CASE
        WHEN ranking = 2 THEN 1
        ELSE NULL::integer
    END)) DESC, (count(
    CASE
        WHEN ranking = 3 THEN 1
```

```
        ELSE NULL::integer

    END)) DESC, participant_country;
```

The "BUILDER".medal_table view generates a medal count by country:

1. event_rankings CTE for Ranking Logic: Using a CTE, the view ranks participants within each event, partitioned by event_name to ensure rankings reset per event. DENSE_RANK() is used to assign ranks without gaps for ties, and a CASE expression orders results by ascending time or descending distance to handle different result types.
2. Counting Medals by Position: The main query counts participants with a rank of 1 (gold), 2 (silver), or 3 (bronze) for each country, using conditional CASE statements within COUNT() to capture each medal type.
3. Country-Level Grouping and Ordering: The view groups by participant_country to consolidate medal counts per country, and orders results by gold, silver, and bronze counts in descending order for a traditional medal table ranking.

US 37 < Europe 54 total medals

Exercise 2

```
-- "BUILDER".stock_prices_with_moving_average source

CREATE OR REPLACE VIEW "BUILDER".stock_prices_with_moving_average
AS SELECT "Date",
    "Stock",
    "Close",
    avg("Close") OVER (PARTITION BY "Stock" ORDER BY "Date" ROWS 4 PRECEDING) AS
"5_day_moving_average"
   FROM stocks;
```

This view calculates a 5-day moving average for each stock:

- Moving Average Calculation: Uses a window function with ROWS 4 PRECEDING to include the current day and the prior 4 days, giving a 5-day rolling average of Close prices.
- Partitioning by Stock: Ensures that averages are computed separately for each stock, providing a continuous moving average specific to each stock's daily performance.

```
-- "BUILDER".stock_prices_2019 source

CREATE OR REPLACE VIEW "BUILDER".stock_prices_2019
AS SELECT "Stock",
    min("Close") FILTER (WHERE "Date"::text >= '2019-01-01'::text AND "Date"::text < '2020-01-01'::text) AS
"Min_Price_2019",
    max("Close") FILTER (WHERE "Date"::text >= '2019-01-01'::text AND "Date"::text < '2020-01-01'::text) AS
"Max_Price_2019"
   FROM stocks
  GROUP BY "Stock";
```

This view identifies the minimum and maximum closing prices for each stock in 2019:

- Filtering by Date: The FILTER clause restricts the calculation to dates within 2019, isolating the yearly high and low values.
- Grouping by Stock: By grouping, the view provides a summary of yearly price extremes per stock, useful for tracking 2019 performance at a glance.

```
-- "BUILDER".top_gainer_2019 source

CREATE OR REPLACE VIEW "BUILDER".top_gainer_2019
AS WITH stock_gains AS (
        SELECT stocks."Stock",
          (stocks."Close" - lag(stocks."Close") OVER (PARTITION BY stocks."Stock" ORDER BY stocks."Date")) /
lag(stocks."Close") OVER (PARTITION BY stocks."Stock" ORDER BY stocks."Date") * 100::double precision AS
"Gain_Percentage"
          FROM stocks
         WHERE stocks."Date"::text >= '2019-01-01'::text AND stocks."Date"::text < '2020-01-01'::text
       )
 SELECT "Stock",
    max("Gain_Percentage") AS "Max_Gain_Percentage"
   FROM stock_gains
  GROUP BY "Stock"
  ORDER BY (max("Gain_Percentage")) DESC
 LIMIT 1;
```

This view finds the stock with the highest daily percentage gain in 2019:

- Percentage Gain Calculation: A lag function computes daily percentage changes by comparing each closing price with the previous day's, allowing for daily gain tracking.
- Filtering and Ranking: Only data from 2019 is included, and stocks are ordered by their highest daily gain percentage, with the top stock identified by limiting results to 1.

```
-- "BUILDER".stock_daily_gains source

CREATE OR REPLACE VIEW "BUILDER".stock_daily_gains
AS SELECT "Date",
    "Stock",
       CASE
          WHEN "Close" > lag("Close") OVER (PARTITION BY "Stock" ORDER BY "Date") THEN 1
          WHEN "Close" < lag("Close") OVER (PARTITION BY "Stock" ORDER BY "Date") THEN 0
          ELSE NULL::integer
       END AS "GAIN"
   FROM stocks
  WHERE "Date"::text >= '2019-01-01'::text AND "Date"::text < '2020-01-01'::text;
```

This view marks each day in 2019 as a gain or loss for each stock:

- Gain Determination: Compares each day's Close price with the previous day using lag. A 1 indicates a gain, 0 a loss, and NULL when no change.
- Date Range Filter: Restricting data to 2019 limits the analysis to a single year, making it easier to track yearly daily trends for each stock.

```
-- "BUILDER".apple_gains_2019 source

CREATE OR REPLACE VIEW "BUILDER".apple_gains_2019
AS SELECT count(*) AS "Num_Gain_Days"
   FROM stock_daily_gains
  WHERE "Stock"::text = 'AAPL'::text AND "GAIN" = 1 AND "Date"::text >= '2019-01-01'::text AND "Date"::text <
'2020-01-01'::text;
```

This view counts the days Apple stock (AAPL) gained in 2019:

- Filter by Stock and Gain Indicator: Filters for AAPL where GAIN = 1, only counting positive change days, which gives a quick summary of how often Apple stock increased in 2019.

```
-- "BUILDER".monthly_stock_gains source

CREATE OR REPLACE VIEW "BUILDER".monthly_stock_gains
AS WITH monthly_stock_prices AS (
         SELECT stocks."Stock",
           date_trunc('month'::text, stocks."Date"::date::timestamp with time zone) AS "Month",
           first_value(stocks."Close") OVER (PARTITION BY stocks."Stock", (date_trunc('month'::text,
stocks."Date"::date::timestamp with time zone)) ORDER BY (stocks."Date"::date)) AS "First_Day_Close",
           last_value(stocks."Close") OVER (PARTITION BY stocks."Stock", (date_trunc('month'::text,
stocks."Date"::date::timestamp with time zone)) ORDER BY (stocks."Date"::date)) AS "Last_Day_Close"
          FROM stocks
         WHERE stocks."Date"::date >= '2019-01-01'::date AND stocks."Date"::date < '2020-01-01'::date
       )
 SELECT "Stock",
    "Month",
    ("Last_Day_Close" - "First_Day_Close") / "First_Day_Close" * 100::double precision AS "Monthly_Gain_Percentage"
   FROM monthly_stock_prices;
```

This view calculates the monthly percentage gain for each stock in 2019:

- First and Last Close of Each Month: The first_value and last_value functions capture each month's opening and closing prices.
- Monthly Gain Calculation: Subtracts the month's first closing price from the last, divides by the first, and converts to a percentage, providing a clear month-by-month view of stock performance trends.

LAB 3

Exercise 1

```
-- Drop existing tables and functions
DROP TABLE IF EXISTS LOGTEAM;
DROP TABLE IF EXISTS EUROLEAGUE;
DROP TABLE IF EXISTS MATCHES;
DROP TABLE IF EXISTS TEAMS;
DROP FUNCTION IF EXISTS log_team_insertion_and_add_to_euroleague();
-- Create TEAMS table
CREATE TABLE TEAMS (
  TeamName VARCHAR(20) PRIMARY KEY,
  TeamCountry VARCHAR(20) NOT NULL,
  CONSTRAINT check_country CHECK (TeamCountry IN ('Spain', 'England'))
);
-- Create MATCHES table
CREATE TABLE MATCHES (
  MatchID SERIAL PRIMARY KEY,
  TeamA_Name VARCHAR(20) REFERENCES TEAMS(TeamName),
  TeamB_Name VARCHAR(20) REFERENCES TEAMS(TeamName),
  Goal_A INTEGER NOT NULL CHECK (Goal_A >= 0),
  Goal_B INTEGER NOT NULL CHECK (Goal_B >= 0),
  Competition VARCHAR(20) NOT NULL,
  CONSTRAINT check_competition CHECK (Competition IN ('Champions League', 'Europa League', 'Premier League',
'LaLiga'))
);
-- Create EUROLEAGUE table
CREATE TABLE EUROLEAGUE (
  TeamName VARCHAR(20) PRIMARY KEY REFERENCES TEAMS(TeamName),
  Points INTEGER NOT NULL,
  Goals_scored INTEGER NOT NULL,
  Goals_conceded INTEGER NOT NULL,
  Difference INTEGER NOT NULL
);
-- Create LOGTEAM table to store insertion timestamps
CREATE TABLE LOGTEAM (
  LogID SERIAL PRIMARY KEY,
  TeamName VARCHAR(20) NOT NULL,
  InsertionTimestamp TIMESTAMP NOT NULL
);
-- Create a trigger function to log insertions into TEAMS and add to EUROLEAGUE
```

```sql
CREATE OR REPLACE FUNCTION log_team_insertion_and_add_to_euroleague()
RETURNS TRIGGER AS $$
BEGIN
  -- Log the insertion
  INSERT INTO LOGTEAM (TeamName, InsertionTimestamp)
  VALUES (NEW.TeamName, CURRENT_TIMESTAMP);
  -- Insert into EUROLEAGUE if not already present
  INSERT INTO EUROLEAGUE (TeamName, Points, Goals_scored, Goals_conceded, Difference)
  VALUES (NEW.TeamName, 0, 0, 0, 0)
  ON CONFLICT (TeamName) DO NOTHING;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
-- Create a trigger to log insertions into TEAMS and add to EUROLEAGUE
CREATE TRIGGER log_team_insertion_and_add_to_euroleague_trigger
AFTER INSERT ON TEAMS
FOR EACH ROW
EXECUTE FUNCTION log_team_insertion_and_add_to_euroleague();
-- Trigger to check team countries for Premier League and La Liga matches
CREATE OR REPLACE FUNCTION check_team_countries()
RETURNS TRIGGER AS $$
DECLARE
  team_a_country VARCHAR(20);
  team_b_country VARCHAR(20);
BEGIN
  SELECT TeamCountry INTO team_a_country FROM TEAMS WHERE TeamName = NEW.TeamA_Name;
  SELECT TeamCountry INTO team_b_country FROM TEAMS WHERE TeamName = NEW.TeamB_Name;
  IF NEW.Competition = 'Premier League' AND (team_a_country != 'England' OR team_b_country != 'England') THEN
     RAISE EXCEPTION 'Both teams must be from England for Premier League matches';
  ELSIF NEW.Competition = 'LaLiga' AND (team_a_country != 'Spain' OR team_b_country != 'Spain') THEN
     RAISE EXCEPTION 'Both teams must be from Spain for LaLiga matches';
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER check_team_countries_trigger
BEFORE INSERT ON MATCHES
FOR EACH ROW
EXECUTE FUNCTION check_team_countries();
-- Trigger to check match count and update EUROLEAGUE
CREATE OR REPLACE FUNCTION check_match_count_and_update_euroleague()
RETURNS TRIGGER AS $$
DECLARE
  match_count_a INT;
  match_count_b INT;
BEGIN
  -- Check match count for TeamA
  SELECT COUNT(*) INTO match_count_a FROM MATCHES
  WHERE TeamA_Name = NEW.TeamA_Name OR TeamB_Name = NEW.TeamA_Name;
```

```
  -- Check match count for TeamB
  SELECT COUNT(*) INTO match_count_b FROM MATCHES
  WHERE TeamA_Name = NEW.TeamB_Name OR TeamB_Name = NEW.TeamB_Name;
  IF match_count_a >= 4 OR match_count_b >= 4 THEN
     RAISE EXCEPTION 'A team cannot have more than 4 matches';
  END IF;
  -- Update EUROLEAGUE table
  IF NEW.Goal_A > NEW.Goal_B THEN
     -- TeamA wins
     UPDATE EUROLEAGUE SET
        Points = Points + 3,
        Goals_scored = Goals_scored + NEW.Goal_A,
        Goals_conceded = Goals_conceded + NEW.Goal_B,
        Difference = Difference + (NEW.Goal_A - NEW.Goal_B)
     WHERE TeamName = NEW.TeamA_Name;
     UPDATE EUROLEAGUE SET
        Goals_scored = Goals_scored + NEW.Goal_B,
        Goals_conceded = Goals_conceded + NEW.Goal_A,
        Difference = Difference - (NEW.Goal_A - NEW.Goal_B)
     WHERE TeamName = NEW.TeamB_Name;
  ELSIF NEW.Goal_B > NEW.Goal_A THEN
     -- TeamB wins
     UPDATE EUROLEAGUE SET
        Points = Points + 3,
        Goals_scored = Goals_scored + NEW.Goal_B,
        Goals_conceded = Goals_conceded + NEW.Goal_A,
        Difference = Difference + (NEW.Goal_B - NEW.Goal_A)
     WHERE TeamName = NEW.TeamB_Name;
     UPDATE EUROLEAGUE SET
        Goals_scored = Goals_scored + NEW.Goal_A,
        Goals_conceded = Goals_conceded + NEW.Goal_B,
        Difference = Difference - (NEW.Goal_B - NEW.Goal_A)
     WHERE TeamName = NEW.TeamA_Name;
  ELSE
     -- Draw
     UPDATE EUROLEAGUE SET
        Points = Points + 1,
        Goals_scored = Goals_scored + NEW.Goal_A,
        Goals_conceded = Goals_conceded + NEW.Goal_B
     WHERE TeamName IN (NEW.TeamA_Name, NEW.TeamB_Name);
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER check_match_count_and_update_euroleague_trigger
BEFORE INSERT ON MATCHES
FOR EACH ROW
EXECUTE FUNCTION check_match_count_and_update_euroleague();
INSERT INTO TEAMS (TeamName, TeamCountry) VALUES
```

```
('Arsenal', 'England'),
('Manchester City', 'England'),
('Manchester United', 'England'),
('Chelsea', 'England'),
('Real Madrid', 'Spain'),
('Barcelona', 'Spain'),
('Atletico Madrid', 'Spain'),
('Sevilla', 'Spain');
INSERT INTO MATCHES (TeamA_Name, TeamB_Name, Goal_A, Goal_B, Competition) VALUES
('Arsenal', 'Manchester City', 2, 1, 'Premier League'),
('Real Madrid', 'Barcelona', 3, 3, 'LaLiga'),
('Manchester United', 'Chelsea', 0, 2, 'Premier League'),
('Atletico Madrid', 'Sevilla', 1, 0, 'LaLiga');
SELECT * FROM TEAMS;
SELECT * FROM LOGTEAM;
SELECT * FROM MATCHES;
SELECT * FROM EUROLEAGUE;
```

1. log_team_insertion_and_add_to_euroleague Trigger/Function

- Purpose: This trigger logs each team insertion into the LOGTEAM table and automatically adds the team to the EUROLEAGUE table if not already present.
- Logging Insertions: The function inserts a record in LOGTEAM with the team name and current timestamp, providing a history of team additions.
- Automatic Addition to EUROLEAGUE: To simplify maintenance, teams are added with default statistics to the EUROLEAGUE table (Points, Goals_scored, Goals_conceded, Difference) set to zero. Using ON CONFLICT DO NOTHING ensures no duplicates.

2. check_team_countries Trigger/Function

- Purpose: This trigger checks the validity of team countries for specific competitions, ensuring league consistency.
- Country Check for Premier League and LaLiga: Before insertion into MATCHES, the function verifies that teams in Premier League matches are from England and teams in LaLiga matches are from Spain, raising an exception if the condition is not met. This maintains data integrity by enforcing country-specific restrictions for these leagues.

3. check_match_count_and_update_euroleague Trigger/Function

- Purpose: This trigger limits the number of matches per team to 4 and updates EUROLEAGUE standings after each match based on the outcome.
- Match Count Check: Before inserting a match, the function counts previous matches for each team. If a team already has 4 matches, an exception is raised to prevent additional entries, ensuring compliance with the match limit rule.
- Updating Standings Based on Results:
    - For wins, the function adds 3 points and updates goals scored/conceded and the goal difference for both teams.
    - For draws, each team receives 1 point, and goals scored/conceded are updated without a goal difference change.
    - The comprehensive update logic keeps EUROLEAGUE statistics accurate without needing external data adjustments.

Exercise 2

```
DROP TABLE IF EXISTS OLD_PATIENT_DATA;
DROP TABLE IF EXISTS PATIENTS;
CREATE TABLE PATIENTS (
  patient_id SERIAL PRIMARY KEY,
  date DATE NOT NULL,
  patient_fname VARCHAR(20) NOT NULL,
  patient_lname VARCHAR(20) NOT NULL,
  age INTEGER NOT NULL,
  weight NUMERIC(5,2) NOT NULL,
  height NUMERIC(5,2) NOT NULL,
  bmi NUMERIC(4,2) NOT NULL
);
CREATE OR REPLACE FUNCTION calculate_bmi()
RETURNS TRIGGER AS $$
BEGIN
  NEW.bmi := NEW.weight / ((NEW.height/ 100) * (NEW.height /100));
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER update_bmi
BEFORE INSERT OR UPDATE ON PATIENTS
FOR EACH ROW
EXECUTE FUNCTION calculate_bmi();
CREATE TABLE OLD_PATIENT_DATA (
  patient_id INTEGER,
  record_id SERIAL,
  date DATE NOT NULL,
  age INTEGER NOT NULL,
  weight NUMERIC(5,2) NOT NULL,
  height NUMERIC(5,2) NOT NULL,
  bmi NUMERIC(4,2) NOT NULL,
  PRIMARY KEY (patient_id, record_id),
  FOREIGN KEY (patient_id) REFERENCES PATIENTS(patient_id)
);
CREATE OR REPLACE FUNCTION store_old_data()
RETURNS TRIGGER AS $$
DECLARE
  next_record_id INTEGER;
BEGIN
  SELECT COALESCE(MAX(record_id), 0) + 1 INTO next_record_id
  FROM old_patient_data
  WHERE patient_id = OLD.patient_id;
  INSERT INTO old_patient_data (patient_id, record_id, date, age, weight, height, bmi)
  VALUES (OLD.patient_id, next_record_id, OLD.date, OLD.age, OLD.weight, OLD.height, OLD.bmi);
  RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER store_old_data_trigger
BEFORE UPDATE ON PATIENTS
FOR EACH ROW
EXECUTE FUNCTION store_old_data();
INSERT INTO patients (date, patient_fname, patient_lname, age, weight, height)
VALUES ('2024-01-01', 'John','Blog', 48, 71, 175);
INSERT INTO patients (date, patient_fname, patient_lname, age, weight, height)
VALUES('2024-03-01', 'Mary', 'Stuart', 24, 56, 172);
UPDATE patients
SET weight = 72
WHERE patient_id = 1;
UPDATE patients
SET height = 173
WHERE patient_id = 2;
SELECT * FROM patients;
SELECT * FROM old_patient_data;
```

1. calculate_bmi Trigger/Function

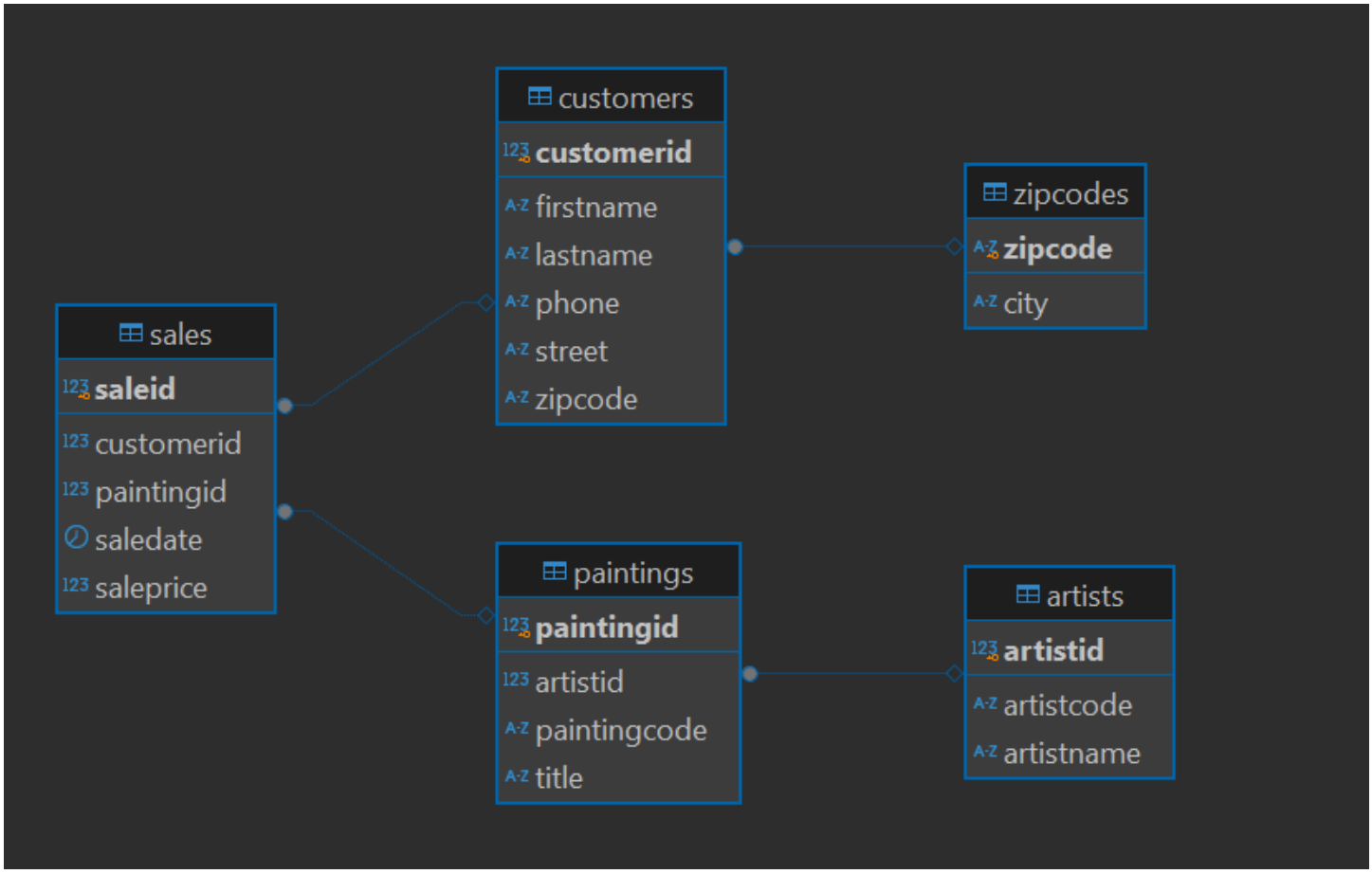
- Purpose: This trigger automatically calculates the Body Mass Index (BMI) for each patient based on their weight and height upon insertion or update.
- BMI Calculation Logic: The function uses the formula weight / ((height/100) * (height/100)), ensuring the BMI is calculated correctly in metric units (kilograms and meters).
- Automatic Execution: By defining this function to run before any insert or update on the PATIENTS table, it maintains data integrity by ensuring that the BMI field is always updated in real-time without manual intervention.

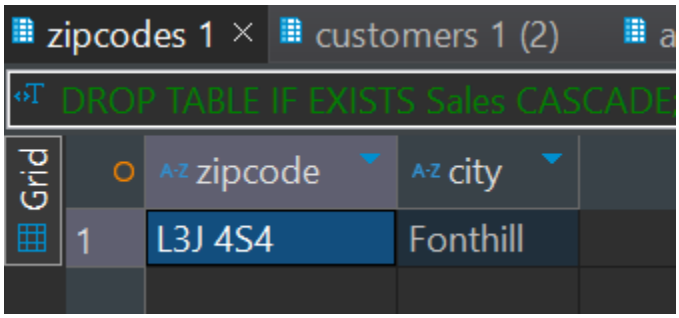
2. store_old_data Trigger/Function

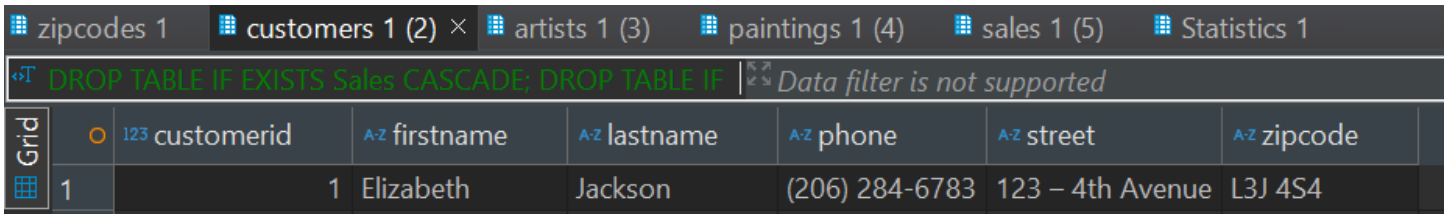
- Purpose: This trigger archives the old patient data into the OLD_PATIENT_DATA table before any update occurs on the PATIENTS table.
- Data Archiving: The function captures the previous state of a patient's record (including date, age, weight, height, and bmi) to preserve historical data, which is crucial for tracking patient changes over time.
- Dynamic Record ID Generation: The function calculates the next record_id for the archived data by selecting the maximum existing record ID for the patient. This ensures unique record identifiers in the OLD_PATIENT_DATA table.
- Triggered on Updates: This trigger runs before any update on the PATIENTS table, ensuring that any changes to patient data are recorded, thus enabling auditability of patient information.

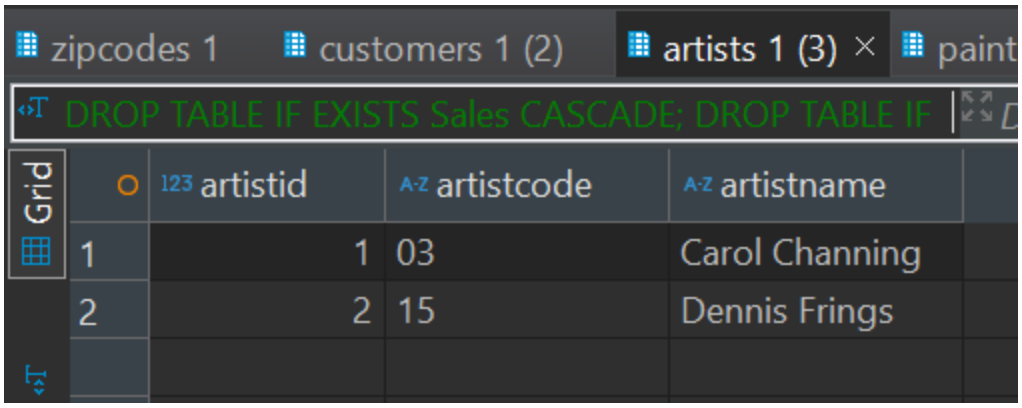
These triggers and functions work together to ensure that patient records are not only accurate and up-to-date but also that historical data is preserved for future reference, which is essential in medical data management.

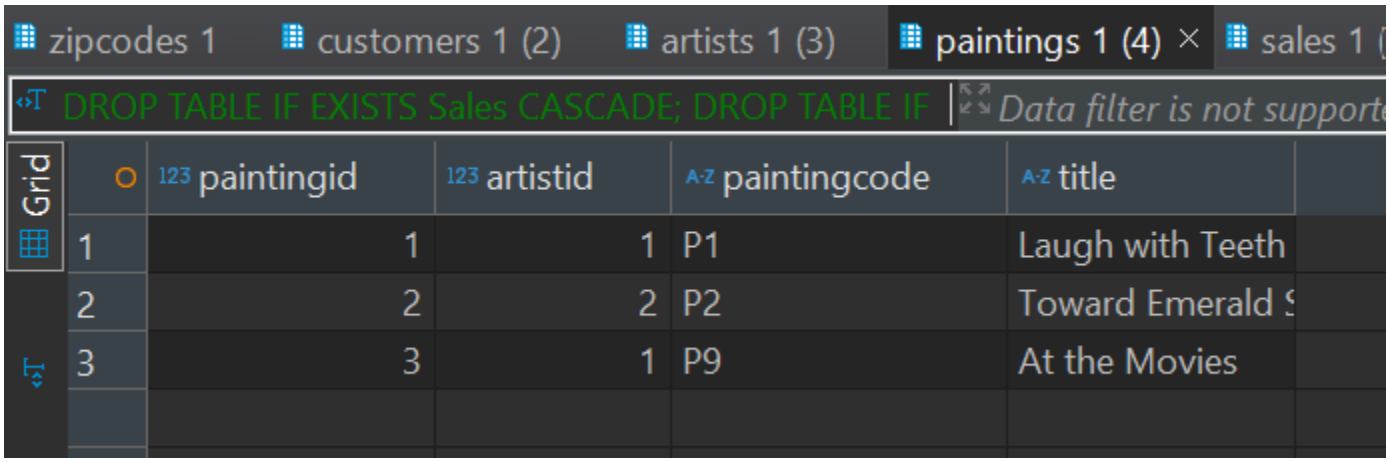
Exercise 1

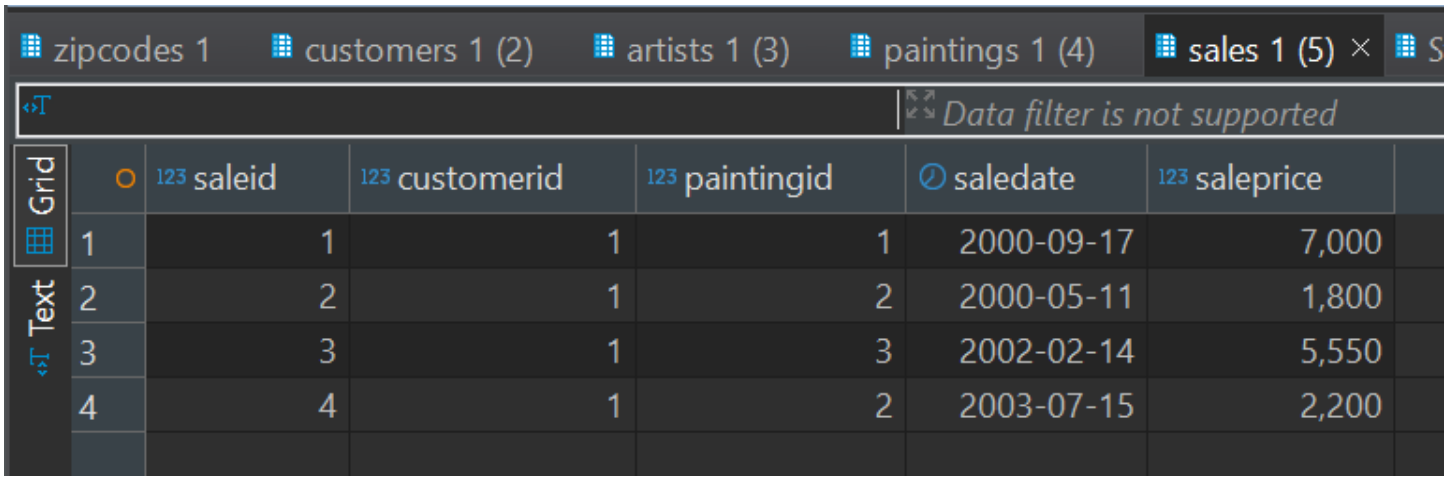

customers
customerid
firstname
lastname
phone
street
zipcode
zipcodes
zipcode
city
sales
saleid
customerid
paintingid
saledate
saleprice
paintings
paintingid
artistid
paintingcode
title
artists
artistid
artistcode
artistname


zipcodes 1
customers 1 (2)
DROP TABLE IF EXISTS Sales CASCADE
Grid
zipcode
city
1
L3J 4S4
Fonthill


zipcodes 1
customers 1 (2)
artists 1 (3)
paintings 1 (4)
sales 1 (5)
Statistics 1
DROP TABLE IF EXISTS Sales CASCADE; DROP TABLE IF
Data filter is not supported
Grid
customerid
firstname
lastname
phone
street
zipcode
1
1
Elizabeth
Jackson
(206) 284-6783
123 – 4th Avenue
L3J 4S4







1. ZipCodes Table

- Purpose: This table stores zip codes and their associated cities.
- Primary Key: ZipCode is defined as the primary key to ensure that each entry is unique and can be efficiently queried.
- City Field: The City field allows for the identification of the city corresponding to each zip code, facilitating geographic-related queries.

2. Customers Table

- Purpose: To store information about customers who purchase paintings.
- Primary Key: CustomerID serves as the primary key, uniquely identifying each customer.
- Foreign Key: The ZipCode field is a foreign key referencing the ZipCodes table. This establishes a relationship between customers and their geographical locations, enforcing data integrity and ensuring that any zip code entered exists in the ZipCodes table.
- Customer Details: Fields like FirstName, LastName, Phone, and Street provide comprehensive contact information for each customer.

3. Artists Table

- Purpose: This table holds information about the artists who create paintings.
- Primary Key: ArtistID is the primary key, ensuring that each artist can be uniquely identified.
- Artist Code: The ArtistCode field serves as a unique identifier for artists, potentially for use in inventory or cataloging systems.
- Artist Name: The ArtistName field contains the name of the artist, allowing for easy reference and querying.

4. Paintings Table

- Purpose: To store information regarding individual paintings created by artists.
- Primary Key: PaintingID acts as the primary key to uniquely identify each painting.
- Foreign Key: ArtistID references the Artists table, linking each painting to its creator. This relationship supports queries that require artist-related information for specific paintings.
- Painting Code and Title: Fields like PaintingCode and Title help in identifying and categorizing paintings, providing a way to manage the artworks in the database.

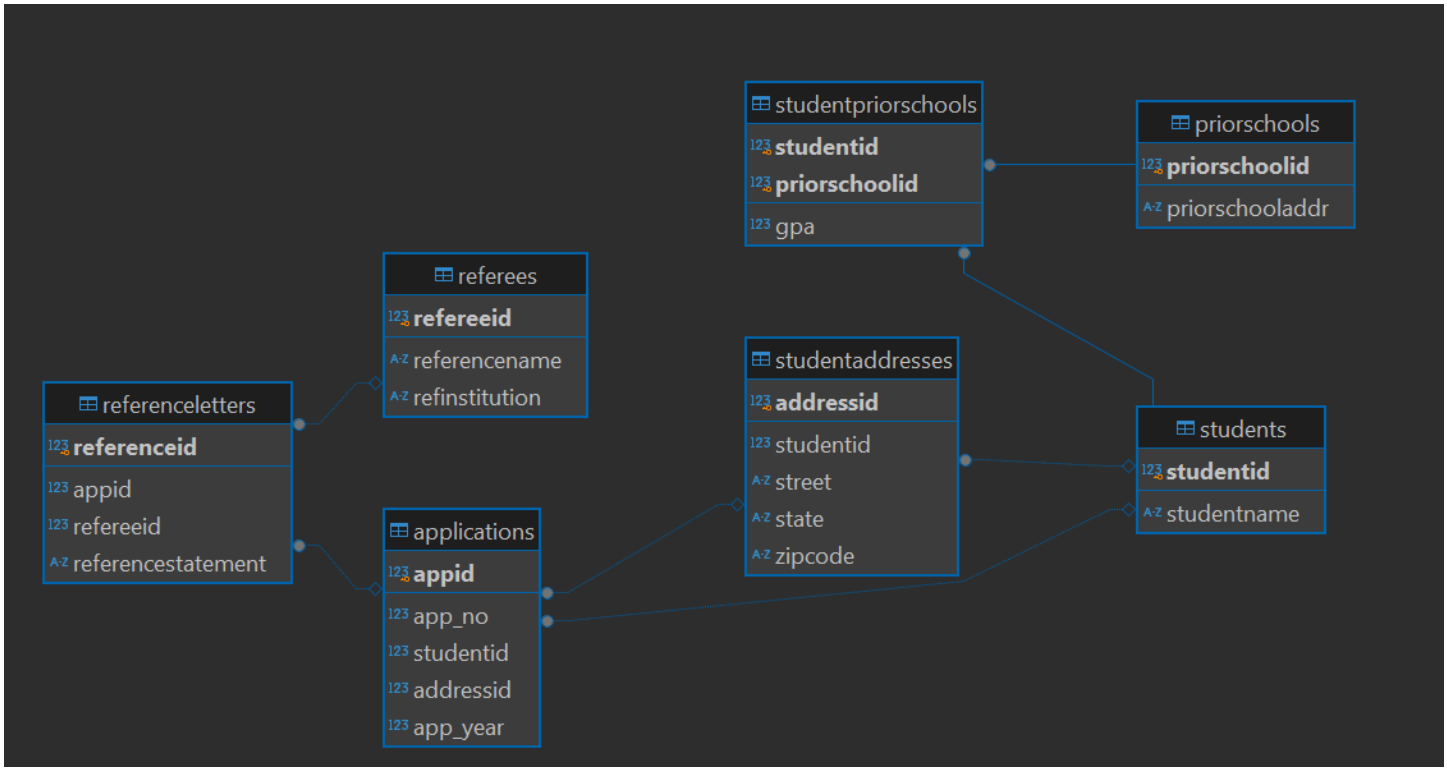
5. Sales Table

- Purpose: This table records sales transactions, capturing details about which paintings were sold, to whom, and for how much.
- Primary Key: SaleID is the primary key, ensuring each sale is uniquely identified.
- Foreign Keys:
    - CustomerID references the Customers table, linking sales to the respective customer.
    - PaintingID references the Paintings table, associating each sale with the specific painting sold. These relationships maintain referential integrity across the database.
- Unique Constraint: The combination of PaintingID and SaleDate has a unique constraint, ensuring that a painting cannot be sold more than once on the same date. This prevents duplicate entries and helps in tracking the sales history accurately.
- Sale Details: The SaleDate and SalePrice fields capture essential information regarding the transaction, allowing for analysis of sales trends over time.

Overall Design Considerations:

- Normalization: The structure adheres to normalization principles, minimizing data redundancy by separating entities (customers, artists, paintings, sales) into distinct tables. This also facilitates easier updates and maintenance.
- Referential Integrity: By using foreign keys, the design enforces relationships between tables, ensuring that data remains consistent and valid.
- Comprehensive Data Capture: The schema is designed to capture all relevant data about the art sales process, from customer information to the details of paintings and their respective sales, allowing for detailed reporting and analysis.

Exercise 2




```
DROP TABLE IF EXISTS Apps_NOT_Normalized cascade;
Create Table Apps_NOT_Normalized(
 App_No integer,
 StudentID integer,
 StudentName varchar(50),
 Street varchar(100),
 State varchar(30),
 ZipCode varchar(7),
 App_Year integer,
 ReferenceName varchar(100),
 RefInstitution  varchar(100),
 ReferenceStatement varchar(500),
 PriorSchoolId integer,
 PriorSchoolAddr varchar(100),
```

```
 GPA numeric(2)
);
insert into Apps_NOT_Normalized values(1,1,'Mark','Grafton Street','New York','NY234',2003,'Dr. Jones','Trinity
College','Good guy',1,'Castleknock',65);
insert into Apps_NOT_Normalized values(1,1,'Mark','Grafton Street','New York','NY234',2004,'Dr. Jones','Trinity
College','Good guy',1,'Castleknock',65);
insert into Apps_NOT_Normalized values(2,1,'Mark','White Street','Florida','Flo435',2007,'Dr. Jones','Trinity
College','Good guy',1,'Castleknock',65);
insert into Apps_NOT_Normalized values(2,1,'Mark','White Street','Florida','Flo435',2007,'Dr. Jones','Trinity
College','Good guy',2,'Loreto College',87);
insert into Apps_NOT_Normalized values(3,1,'Mark','White Street','Florida','Flo435',2012,'Dr. Jones','U Limerick','Very
Good guy',1,'Castleknock',65);
insert into Apps_NOT_Normalized values(3,1,'Mark','White Street','Florida','Flo435',2012,'Dr. Jones','U Limerick','Very
Good guy',2,'Loreto College',87);
insert into Apps_NOT_Normalized values(2,2,'Sarah','Green Road','California','Cal123',2010,'Dr.
Byrne','DIT','Perfect',1,'Castleknock',90);
insert into Apps_NOT_Normalized values(2,2,'Sarah','Green Road','California','Cal123',2010,'Dr.
Byrne','DIT','Perfect',3,'St. Patrick',76);
insert into Apps_NOT_Normalized values(2,2,'Sarah','Green Road','California','Cal123',2011,'Dr.
Byrne','DIT','Perfect',1,'Castleknock',90);
insert into Apps_NOT_Normalized values(2,2,'Sarah','Green Road','California','Cal123',2011,'Dr.
Byrne','DIT','Perfect',3,'St. Patrick',76);
insert into Apps_NOT_Normalized values(2,2,'Sarah','Green Road','California','Cal123',2012,'Dr.
Byrne','UCD','Average',1,'Castleknock',90);
insert into Apps_NOT_Normalized values(2,2,'Sarah','Green Road','California','Cal123',2012,'Dr.
Byrne','UCD','Average',3,'St. Patrick',76);
insert into Apps_NOT_Normalized values(2,2,'Sarah','Green Road','California','Cal123',2012,'Dr.
Byrne','UCD','Average',4,'DBS',66);
insert into Apps_NOT_Normalized values(2,2,'Sarah','Green Road','California','Cal123',2012,'Dr.
Byrne','UCD','Average',5,'Harvard',45);
insert into Apps_NOT_Normalized values(1,3,'Paul','Red Crescent','Carolina','Ca455',2012,'Dr. Jones','Trinity
College','Poor',1,'Castleknock',45);
insert into Apps_NOT_Normalized values(1,3,'Paul','Red Crescent','Carolina','Ca455',2012,'Dr. Jones','Trinity
College','Poor',3,'St. Patrick',67);
insert into Apps_NOT_Normalized values(1,3,'Paul','Red Crescent','Carolina','Ca455',2012,'Dr. Jones','Trinity
College','Poor',4,'DBS',23);
insert into Apps_NOT_Normalized values(1,3,'Paul','Red Crescent','Carolina','Ca455',2012,'Dr. Jones','Trinity
College','Poor',5,'Harvard',67);
insert into Apps_NOT_Normalized values(3,3,'Paul','Yellow Park','Mexico','Mex1',2008,'Prof.
Cahill','UCC','Excellent',1,'Castleknock',45);
insert into Apps_NOT_Normalized values(3,3,'Paul','Yellow Park','Mexico','Mex1',2008,'Prof.
Cahill','UCC','Excellent',3,'St. Patrick',67);
insert into Apps_NOT_Normalized values(3,3,'Paul','Yellow Park','Mexico','Mex1',2008,'Prof.
Cahill','UCC','Excellent',4,'DBS',23);
insert into Apps_NOT_Normalized values(3,3,'Paul','Yellow Park','Mexico','Mex1',2008,'Prof.
Cahill','UCC','Excellent',5,'Harvard',67);
insert into Apps_NOT_Normalized values(1,4,'Jack','Dartry Road','Ohio','Oh34',2009,'Prof. Lillis','DIT','Fair',3,'St.
Patrick',29);
insert into Apps_NOT_Normalized values(1,4,'Jack','Dartry Road','Ohio','Oh34',2009,'Prof. Lillis','DIT','Fair',4,'DBS',88);
```

```
insert into Apps_NOT_Normalized values(1,4,'Jack','Dartry Road','Ohio','Oh34',2009,'Prof.
Lillis','DIT','Fair',5,'Harvard',66);
insert into Apps_NOT_Normalized values(2,5,'Mary','Malahide Road','Ireland','IRE',2009,'Prof. Lillis','DIT','Good
girl',3,'St. Patrick',44);
insert into Apps_NOT_Normalized values(2,5,'Mary','Malahide Road','Ireland','IRE',2009,'Prof. Lillis','DIT','Good
girl',4,'DBS',55);
insert into Apps_NOT_Normalized values(2,5,'Mary','Malahide Road','Ireland','IRE',2009,'Prof. Lillis','DIT','Good
girl',5,'Harvard',66);
insert into Apps_NOT_Normalized values(2,5,'Mary','Malahide Road','Ireland','IRE',2009,'Prof. Lillis','DIT','Good
girl',1,'Castleknock',74);
insert into Apps_NOT_Normalized values(1,5,'Mary','Black Bay','Kansas','Kan45',2005,'Dr. Byrne','DIT','Perfect',3,'St.
Patrick',44);
insert into Apps_NOT_Normalized values(1,5,'Mary','Black Bay','Kansas','Kan45',2005,'Dr.
Byrne','DIT','Perfect',4,'DBS',55);
insert into Apps_NOT_Normalized values(1,5,'Mary','Black Bay','Kansas','Kan45',2005,'Dr.
Byrne','DIT','Perfect',5,'Harvard',66);
insert into Apps_NOT_Normalized values(3,6,'Susan','River Road','Kansas','Kan45',2011,'Prof.
Cahill','UCC','Messy',1,'Castleknock',88);
insert into Apps_NOT_Normalized values(3,6,'Susan','River Road','Kansas','Kan45',2011,'Prof.
Cahill','UCC','Messy',3,'St. Patrick',77);
insert into Apps_NOT_Normalized values(3,6,'Susan','River Road','Kansas','Kan45',2011,'Prof.
Cahill','UCC','Messy',4,'DBS',56);
insert into Apps_NOT_Normalized values(3,6,'Susan','River Road','Kansas','Kan45',2011,'Prof.
Cahill','UCC','Messy',2,'Loreto College',45);
select * from Apps_NOT_Normalized;
-- Drop existing tables if they exist
DROP TABLE IF EXISTS StudentPriorSchools cascade;
DROP TABLE IF EXISTS ReferenceLetters cascade;
DROP TABLE IF EXISTS Applications cascade;
DROP TABLE IF EXISTS StudentAddresses cascade;
DROP TABLE IF EXISTS Students cascade;
DROP TABLE IF EXISTS Referees cascade;
DROP TABLE IF EXISTS PriorSchools cascade;
-- Students table (1NF, 2NF, 3NF)
CREATE TABLE Students (
  StudentID INTEGER PRIMARY KEY,
  StudentName VARCHAR(50)
);
-- StudentAddresses table (1NF, 2NF, 3NF)
CREATE TABLE StudentAddresses (
  AddressID SERIAL PRIMARY KEY,
  StudentID INTEGER,
  Street VARCHAR(100),
  State VARCHAR(30),
  ZipCode VARCHAR(7),
  FOREIGN KEY (StudentID) REFERENCES Students(StudentID)
);
-- Applications table (1NF, 2NF, 3NF)
CREATE TABLE Applications (
```

```
  AppID SERIAL PRIMARY KEY,
  App_No INTEGER,
  StudentID INTEGER,
  AddressID INTEGER,
  App_Year INTEGER,
  FOREIGN KEY (StudentID) REFERENCES Students(StudentID),
  FOREIGN KEY (AddressID) REFERENCES StudentAddresses(AddressID),
  UNIQUE (App_No, App_Year)
);
-- Referees table (1NF, 2NF, 3NF)
CREATE TABLE Referees (
  RefereeID SERIAL PRIMARY KEY,
  ReferenceName VARCHAR(100),
  RefInstitution VARCHAR(100),
  UNIQUE (ReferenceName, RefInstitution)
);
-- References table (1NF, 2NF, 3NF)
CREATE TABLE ReferenceLetters (
  ReferenceID SERIAL PRIMARY KEY,
  AppID INTEGER,
  RefereeID INTEGER,
  ReferenceStatement VARCHAR(500),
  FOREIGN KEY (AppID) REFERENCES Applications(AppID),
  FOREIGN KEY (RefereeID) REFERENCES Referees(RefereeID)
);
-- PriorSchools table (1NF, 2NF, 3NF)
CREATE TABLE PriorSchools (
  PriorSchoolId INTEGER PRIMARY KEY,
  PriorSchoolAddr VARCHAR(100)
);
-- StudentPriorSchools table (1NF, 2NF, 3NF)
CREATE TABLE StudentPriorSchools (
  StudentID INTEGER,
  PriorSchoolId INTEGER,
  GPA NUMERIC(2),
  PRIMARY KEY (StudentID, PriorSchoolId),
  FOREIGN KEY (StudentID) REFERENCES Students(StudentID),
  FOREIGN KEY (PriorSchoolId) REFERENCES PriorSchools(PriorSchoolId)
);
-- Insert data into Students table
INSERT INTO Students (StudentID, StudentName)
SELECT DISTINCT StudentID, StudentName
FROM Apps_NOT_Normalized;
-- Insert data into StudentAddresses table
INSERT INTO StudentAddresses (StudentID, Street, State, ZipCode)
SELECT DISTINCT StudentID, Street, State, ZipCode
FROM Apps_NOT_Normalized;
-- Insert data into Applications table
INSERT INTO Applications (App_No, StudentID, AddressID, App_Year)
```

```
SELECT DISTINCT a.App_No, a.StudentID, sa.AddressID, a.App_Year
FROM Apps_NOT_Normalized a
JOIN StudentAddresses sa ON a.StudentID = sa.StudentID AND a.Street = sa.Street AND a.State = sa.State AND
a.ZipCode = sa.ZipCode;
-- Insert data into Referees table
INSERT INTO Referees (ReferenceName, RefInstitution)
SELECT DISTINCT ReferenceName, RefInstitution
FROM Apps_NOT_Normalized;
-- Insert data into References table
INSERT INTO ReferenceLetters (AppID, RefereeID, ReferenceStatement)
SELECT DISTINCT a.AppID, r.RefereeID, an.ReferenceStatement
FROM Apps_NOT_Normalized an
JOIN Applications a ON an.App_No = a.App_No AND an.StudentID = a.StudentID AND an.App_Year = a.App_Year
JOIN Referees r ON an.ReferenceName = r.ReferenceName AND an.RefInstitution = r.RefInstitution;
-- Insert data into PriorSchools table
INSERT INTO PriorSchools (PriorSchoolId, PriorSchoolAddr)
SELECT DISTINCT PriorSchoolId, PriorSchoolAddr
FROM Apps_NOT_Normalized;
-- Insert data into StudentPriorSchools table
INSERT INTO StudentPriorSchools (StudentID, PriorSchoolId, GPA)
SELECT DISTINCT StudentID, PriorSchoolId, GPA
FROM Apps_NOT_Normalized;
-- Select all data from Students table
SELECT * FROM Students;
-- Select all data from StudentAddresses table
SELECT * FROM StudentAddresses;
-- Select all data from Applications table
SELECT * FROM Applications;
-- Select all data from Referees table
SELECT * FROM Referees;
-- Select all data from References table
SELECT * FROM ReferenceLetters;
-- Select all data from PriorSchools table
SELECT * FROM PriorSchools;
-- Select all data from StudentPriorSchools table
SELECT * FROM StudentPriorSchools;
```

	123 studentid	A-Z studentname
1	1	Mark
2	2	Sarah
3	4	Jack
4	6	Susan
5	5	Mary
6	3	Paul

	addressid	studentid	street	state	zipcode
1	1	2	Green Road	California	Cal123
2	2	6	River Road	Kansas	Kan45
3	3	3	Red Crescent	Carolina	Ca455
4	4	4	Dartry Road	Ohio	Oh34
5	5	3	Yellow Park	Mexico	Mex1
6	6	5	Black Bay	Kansas	Kan45
7	7	1	White Street	Florida	Flo435
8	8	1	Grafton Stree	New York	NY234
9	9	5	Malahide Roa	Ireland	IRE

apps_not_normalized 1 | students 1 (2) | studentaddresses 1 (3) | applications 1 (4) ×

DROP TABLE IF EXISTS Apps_NOT_Normalized cascade; (| Data filter is not supported

Grid | Text

	appid	app_no	studentid	addressid	app_year
1	1	1	1	8	2,003
2	2	1	1	8	2,004
3	3	1	3	3	2,012
4	4	1	4	4	2,009
5	5	1	5	6	2,005
6	6	2	1	7	2,007
7	7	2	2	1	2,010
8	8	2	2	1	2,011
9	9	2	2	1	2,012
10	10	2	5	9	2,009
11	11	3	1	7	2,012
12	12	3	3	5	2,008
13	13	3	6	2	2,011

apps_not_normalized 1 | students 1 (2) | studentaddresses 1 (3) | applications 1 (4) | referees 1 (5)

```
DROP TABLE IF EXISTS Apps_NOT_Normalized cascade;
```

Data filter is not supported

Grid

	refereeid	referencename	refinstitution
1	1	Dr. Jones	Trinity College
2	2	Dr. Byrne	DIT
3	3	Dr. Byrne	UCD
4	4	Prof. Cahill	UCC
5	5	Dr. Jones	U Limerick
6	6	Prof. Lillis	DIT

apps_not_normalized 1 | students 1 (2) | studentaddresses 1 (3) | applications 1 (4) | referees 1 (5) | referenceletters 1 (6)

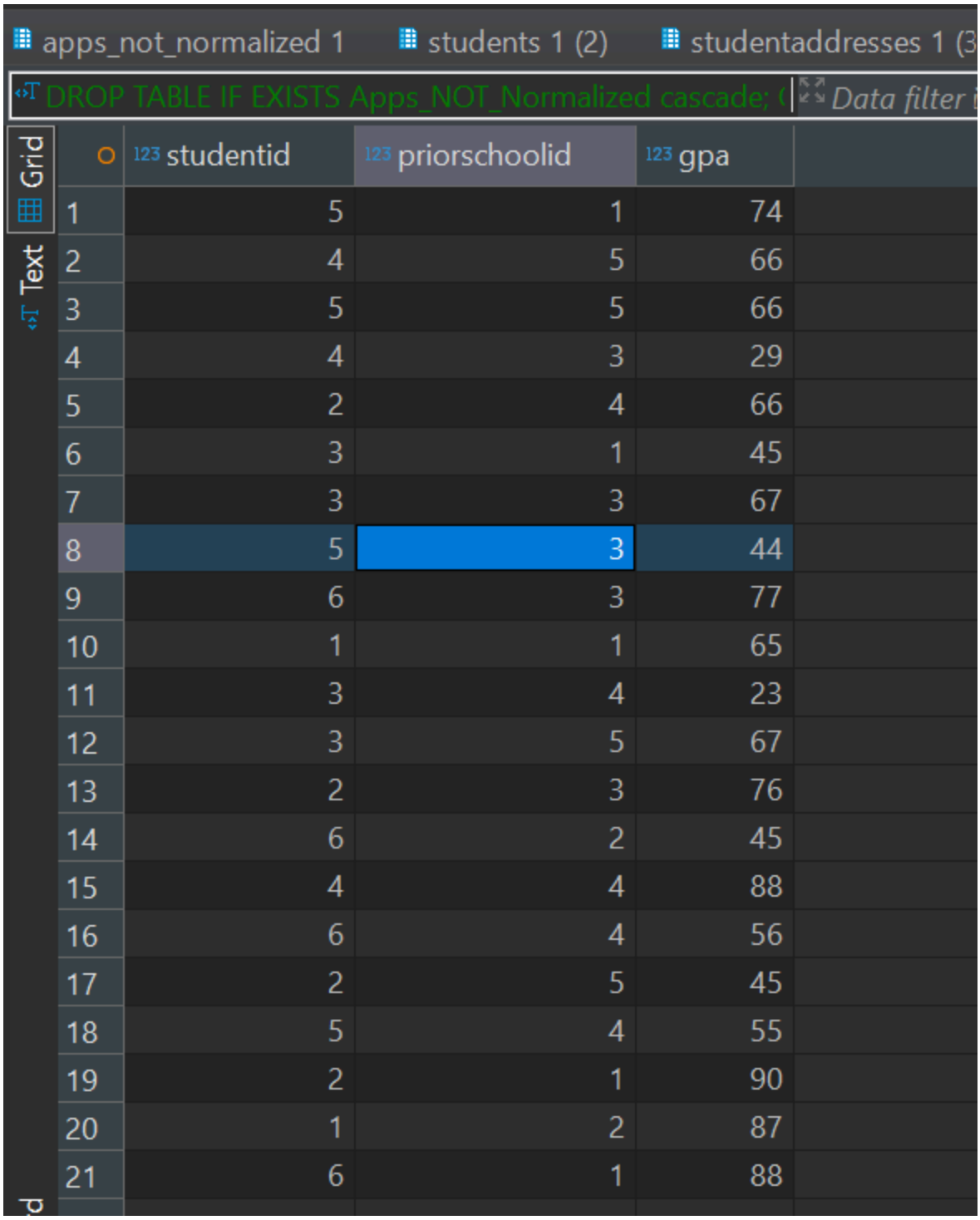
```
DROP TABLE IF EXISTS Apps_NOT_Normalized cascade;
```

Data filter is not supported

Text

	referenceid	appid	refereeid	referencestatement
1	1	1	1	Good guy
2	2	2	1	Good guy
3	3	3	1	Poor
4	4	4	6	Fair
5	5	5	2	Perfect
6	6	6	1	Good guy
7	7	7	2	Perfect
8	8	8	2	Perfect
9	9	9	3	Average
10	10	10	6	Good girl
11	11	11	5	Very Good guy
12	12	12	4	Excellent
13	13	13	4	Messy

	priorschoolid	priorschooladdr
1	1	Castleknock
2	2	Loreto College
3	3	St. Patrick
4	5	Harvard
5	4	DBS



	studentid	priorschoolid	gpa
1	5	1	74
2	4	5	66
3	5	5	66
4	4	3	29
5	2	4	66
6	3	1	45
7	3	3	67
8	5	3	44
9	6	3	77
10	1	1	65
11	3	4	23
12	3	5	67
13	2	3	76
14	6	2	45
15	4	4	88
16	6	4	56
17	2	5	45
18	5	4	55
19	2	1	90
20	1	2	87
21	6	1	88

1. Students Table

- Purpose: This table stores unique student information.
- Primary Key: StudentID serves as the primary key, ensuring each student can be uniquely identified.
- Student Name: The StudentName field captures the name of the student, providing a direct way to reference them.

2. StudentAddresses Table

- Purpose: To manage student address information separately from student records.
- Primary Key: AddressID is defined as a SERIAL primary key, allowing for unique identification of each address.
- Foreign Key: StudentID references the Students table, linking addresses to specific students. This enforces referential integrity and ensures that addresses can only belong to valid students.
- Address Fields: Street, State, and ZipCode store complete address details, allowing for multiple addresses per student without redundancy.

3. Applications Table

- Purpose: To track applications submitted by students.
- Primary Key: AppID is a SERIAL primary key, uniquely identifying each application record.
- Foreign Keys:
    - StudentID references the Students table, associating each application with the correct student.
    - AddressID references the StudentAddresses table, linking applications to specific addresses.
- Application Number and Year: App_No and App_Year capture specific details about each application, with a unique constraint on their combination to prevent duplicate applications in a given year.

4. Referees Table

- Purpose: To store information about referees who provide references for student applications.
- Primary Key: RefereeID is a SERIAL primary key for unique identification of each referee.
- Unique Constraint: The combination of ReferenceName and RefInstitution is constrained to ensure that a specific referee at a particular institution is only recorded once, avoiding duplicates.

5. ReferenceLetters Table

- Purpose: To link referees to specific applications and their statements.
- Primary Key: ReferenceID is a SERIAL primary key for unique identification of reference records.
- Foreign Keys:
    - AppID references the Applications table, linking reference letters to the specific application.
    - RefereeID references the Referees table, connecting the reference letter to the correct referee.
- Reference Statement: The ReferenceStatement field captures the content of the reference letter.

6. PriorSchools Table

- Purpose: To manage information about prior schools attended by students.
- Primary Key: PriorSchoolId serves as the primary key to uniquely identify each school.
- Prior School Address: PriorSchoolAddr captures the address of the prior school, providing context for the student's educational background.

7. StudentPriorSchools Table

- Purpose: To associate students with their prior schools and their GPA from each school.
- Composite Primary Key: The combination of StudentID and PriorSchoolId forms a composite primary key, ensuring that each student can only be linked to a specific prior school once.
- Foreign Keys:
    - StudentID references the Students table, linking to the student.
    - PriorSchoolId references the PriorSchools table, connecting to the school.
- GPA Field: The GPA field records the student's GPA at their prior school, capturing important academic performance data.

Overall Design Considerations:

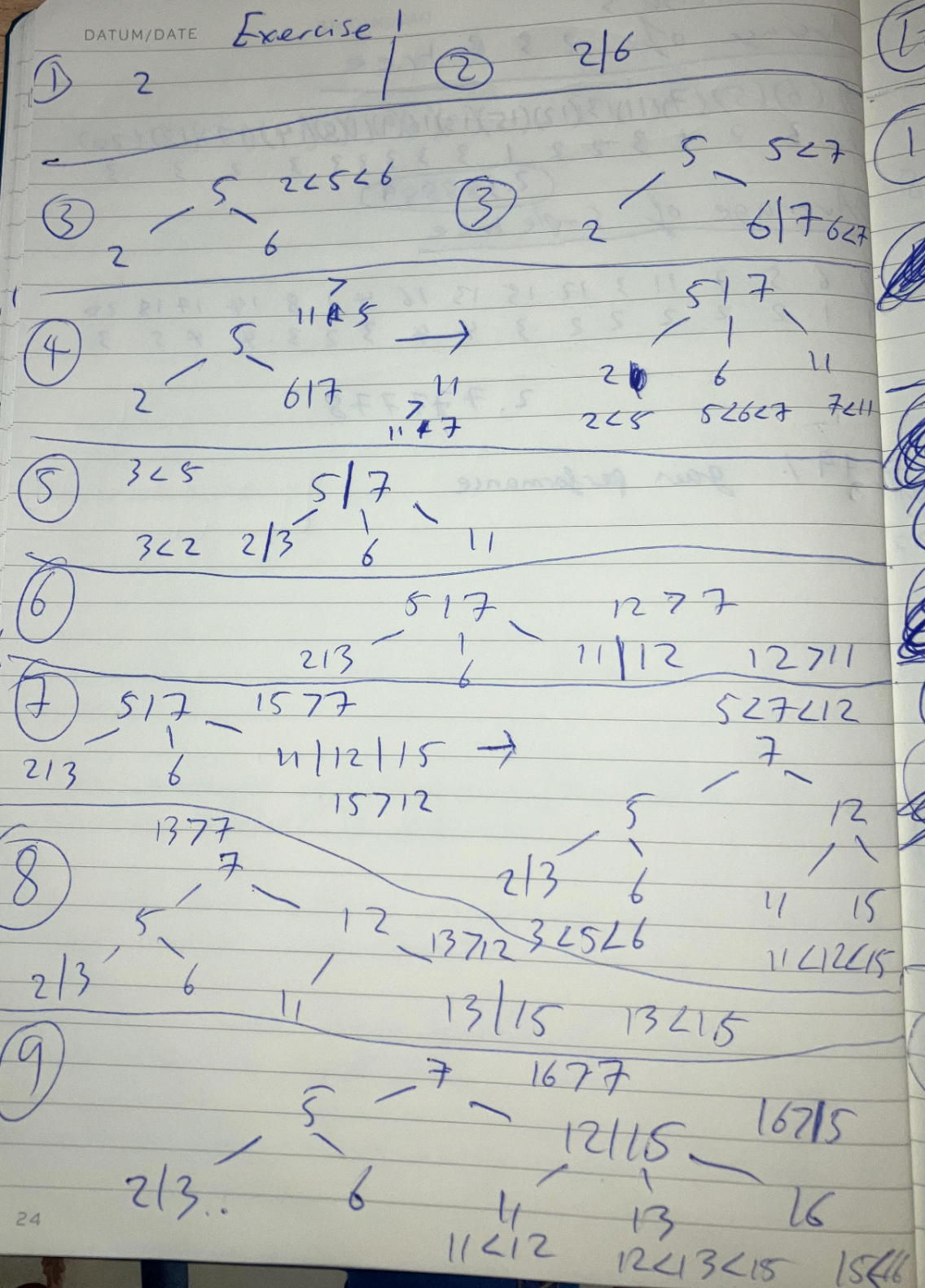
- Normalization: The schema adheres to normalization principles by eliminating redundancy. Each entity (students, addresses, applications, referees, etc.) is represented in its own table, allowing for clear relationships and data integrity.
- Referential Integrity: Foreign keys ensure that relationships between tables are maintained, preventing orphaned records and ensuring valid data connections.
- Ease of Maintenance: The separation of data into distinct tables simplifies updates and changes. For example, if a student's address changes, only the relevant entry in the StudentAddresses table needs to be updated, rather than every record in a non-normalized structure.
- Flexibility: This design allows for multiple addresses per student, multiple applications per year, and multiple references per application, supporting complex queries and reports.

LAB 5

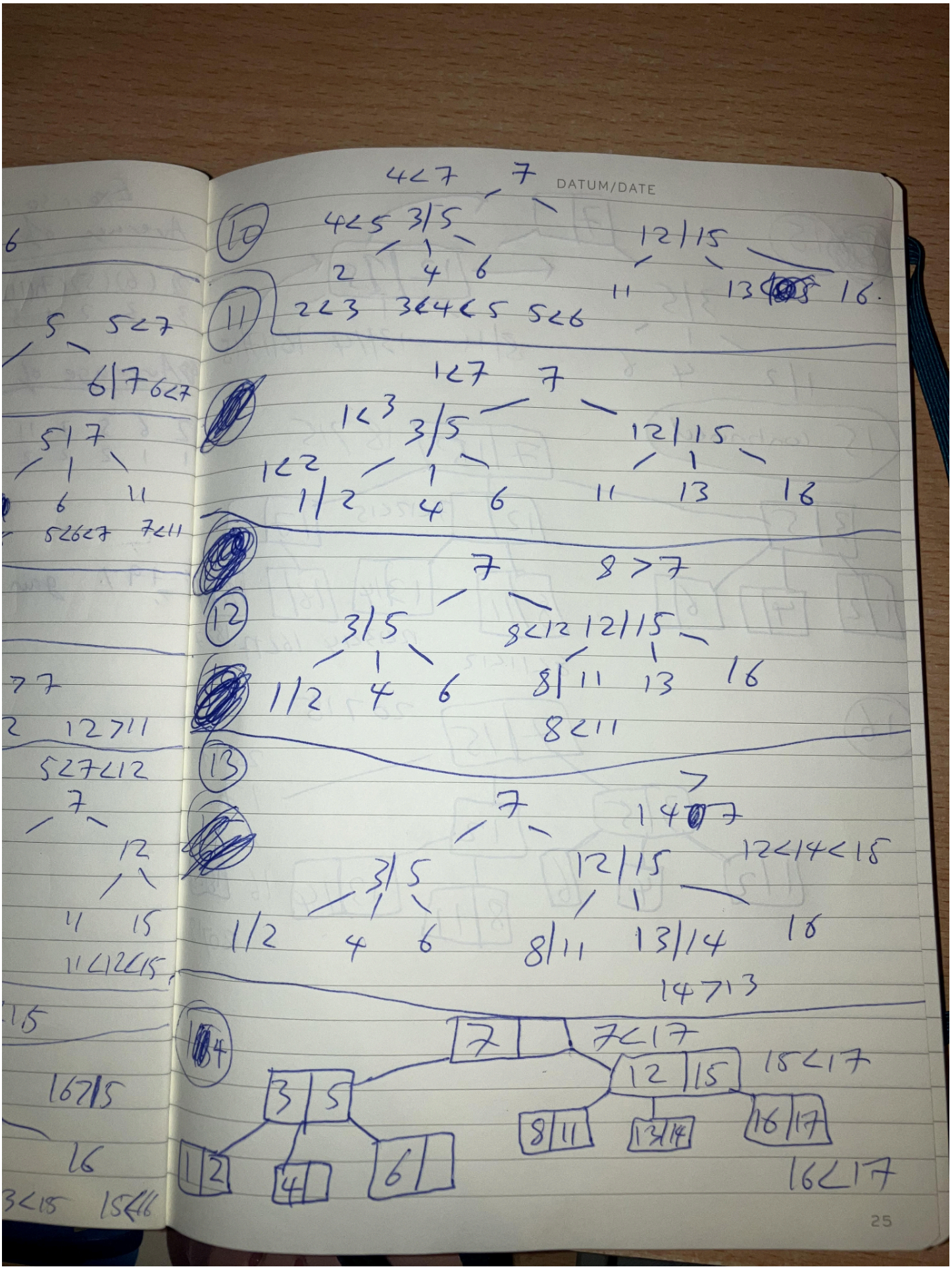
Exercise 1

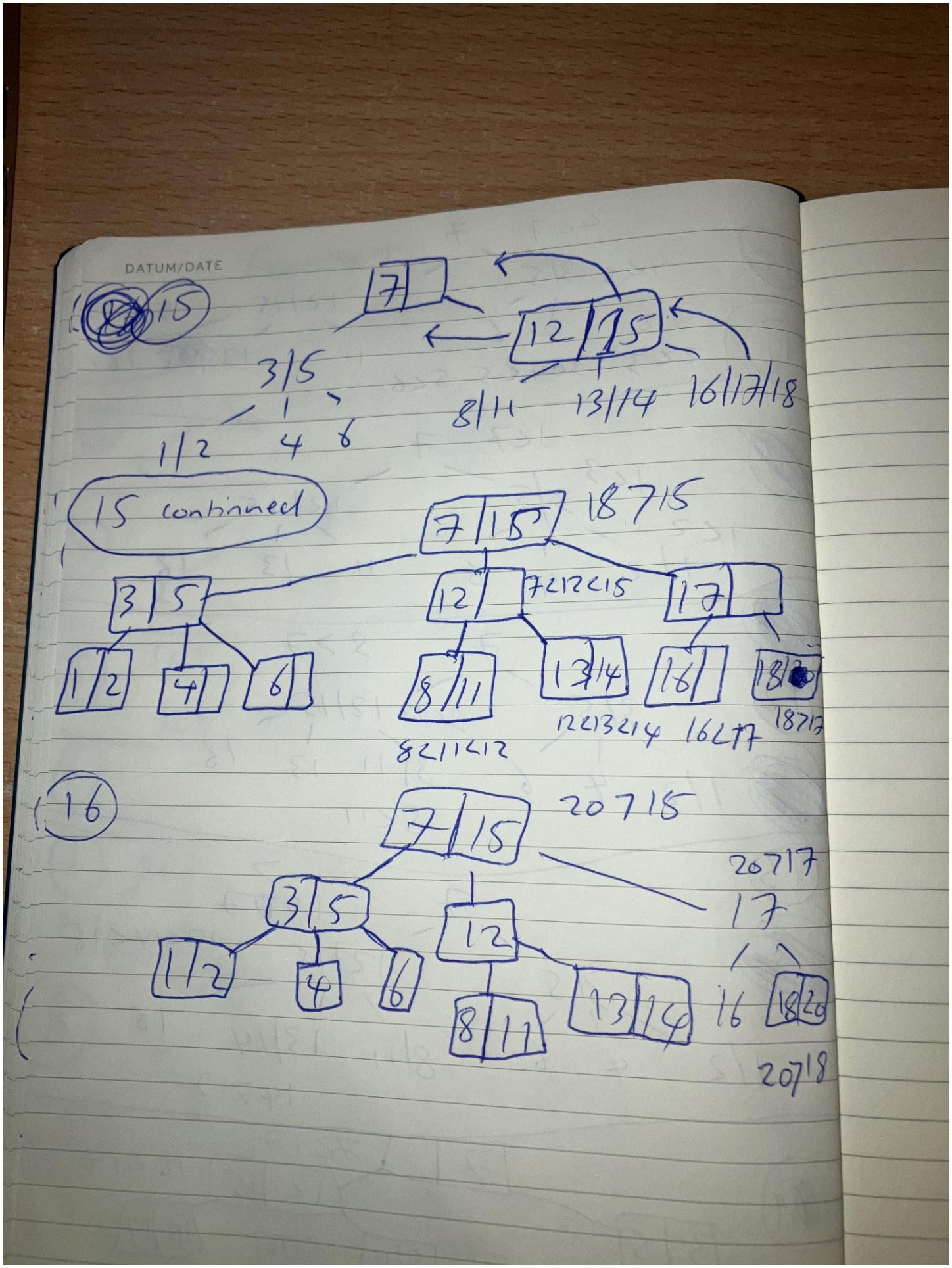
DATUM/DATE

Exercise 1



DATUM/DATE

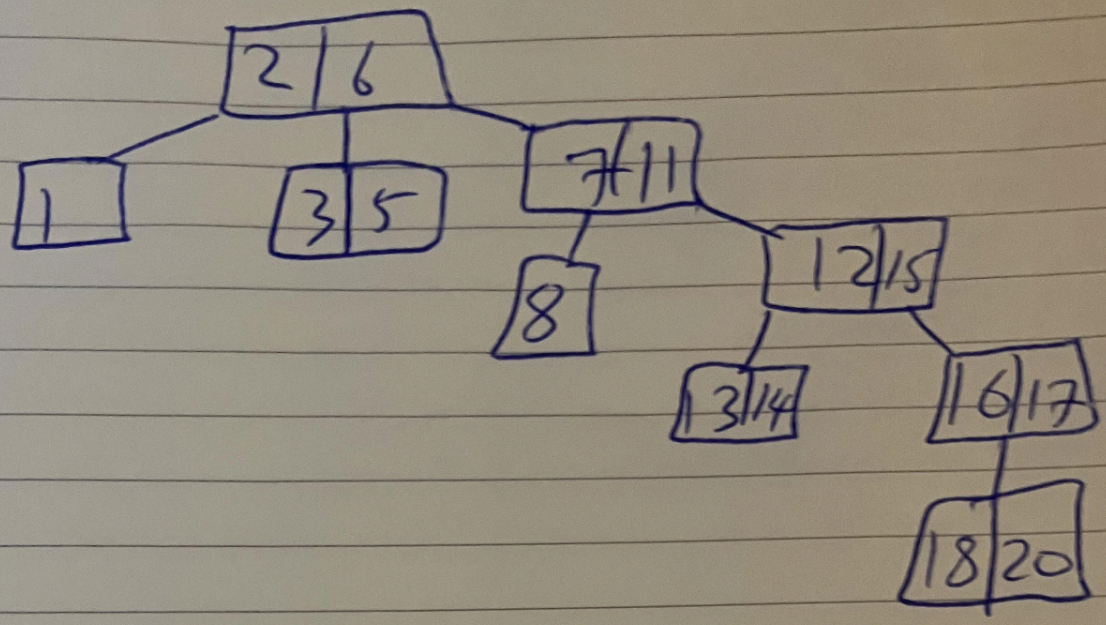

10
11
12
13
14
4<7 7
4<5 3|5
2 4 6
12|15
11 13 16
2<3 3<4<5 5<6
1<7 7
1<3 3|5
1<2 1 4 6
12|15
11 13 16
7
8>7
3|5
8<12 12|15
1|2 4 6
8|11 13 16
8<11
7
7
14 7
3|5
12|15
12<14<15
1|2 4 6
8|11 13|14 16
14>13
7
7<17
12|15
15<17
3|5
8|11
13|14
16|17
1|2
4
6
16<17


DATUM/DATE
15
7
12 15
3 5
1 2 4 6
8 11 13 14 16 17 18
15 combined
7 15
18 7 15
12
7 12 15
17
8 11
13 14
16
18
8 11 12
12 13 14
16 17
18 17
16
20 7 15
20 7 17
17
16
18 20
20 18

Exercise 2

Exercise 2

DATUM/DATE



Exercise 3

Exercise 3

DATUM/DATE

Average of 2.3 B tree

(2)(6)(5)(7)(11)(3)(12)(15)(13)(16)(4)(1)(8)(14)(17)(18)(20)

3 3 2 1 3 2 2 1 3 3 3 3 3 3 2 3 3

2.388889

Average of Simple tree

2 6 5 7 11 3 12 15 13 16 4 1 8 14 17 18 20

1 1 2 2 2 2 2 3 4 4 3 2 3 4 4 5 3

2.777778

19% gain performance

B-tree Analysis:

- All node visits: 3,3,2,1,3,2,2,1,3,3,3,3,3,3,2,3,3
- Total sum = 43
- Number of lookups = 17
- Average = 43/17 = 2.389 nodes per query
- Distribution of node visits:
    - 1 node: 2 times
    - 2 nodes: 4 times
    - 3 nodes: 11 times
    - Maximum depth: 3 nodes

Simple Tree Analysis:

- All node visits: 1,1,2,2,2,2,2,3,4,4,3,2,3,4,4,5,3
- Total sum = 47
- Number of lookups = 17
- Average = 47/17 = 2.778 nodes per query
- Distribution of node visits:
    - 1 node: 2 times
    - 2 nodes: 7 times
    - 3 nodes: 4 times
    - 4 nodes: 3 times
    - 5 nodes: 1 time
    - Maximum depth: 5 nodes

Performance Comparison:

- B-tree average: 2.389
- Simple tree average: 2.778
- Difference: 2.778 - 2.389 = 0.389
- Performance gain = (0.389/2.778) × 100 = 19%

Key Observations:

1. B-tree:
    - More consistent depth (mostly 2-3 nodes)
    - Maximum depth is 3
    - Higher concentration of 3-node visits
2. Simple tree:
    - More varied depth (ranges from 1-5 nodes)
    - Maximum depth is 5
    - More evenly distributed visits across 2-4 nodes

The B-tree's more balanced structure results in more consistent and generally lower node visits, making it 19% more efficient overall.

Exercise 4

1. Query Analysis:

Query: `EXPLAIN ANALYZE SELECT * FROM persons;`

- Total Cost: 169.86

- Type of Operation: Sequential Scan

2. Query Analysis:

Query: `EXPLAIN ANALYZE SELECT * FROM persons WHERE person_id > 1000 AND person_id < 3000;`

- Total Cost: 197.79

- Type of Operation: Sequential Scan

- Is the Cost Higher or Lower than Query 1? Higher

- Reason: The cost is higher because:

  - A large portion of the table is scanned.

  - No index is used, leading to a full table scan.

  - Additional processing is needed due to filtering.

3. Query Analysis:

Query: `EXPLAIN ANALYZE SELECT * FROM persons WHERE person_id >= 3;`

- Total Cost: 183.82

- Type of Operation: Sequential Scan

- Is the Cost Higher or Lower than Query 2? Lower

- Reason: While Query 2 limits rows to 1000-3000, PostgreSQL examines more rows to locate the subset, increasing cost. Query 3 processes a larger portion of the table, but benefits from continuous data retrieval, lowering cost due to reduced filtering.

4. Adding a Primary Key and Re-running Queries:

Command: `ALTER TABLE persons ADD PRIMARY KEY (person_id);`

Re-running Queries 2 and 3:

- Query 2 Total Cost (after indexing): 97.07

- Query 3 Total Cost (after indexing): 221.78

- Explanation: Query 2 benefits more from the index by efficiently retrieving a specific range, while Query 3's cost increases due to overhead from scanning nearly all rows with the index.

5. Query Analysis with Expression:

Query: `EXPLAIN ANALYZE SELECT * FROM persons WHERE person_id + 5 > 1000 AND person_id < 3000;`

- Total Cost: 211.75

- Is the Cost Higher or Lower than Query 2? Higher

- Reason: The expression `person_id + 5 > 1000` prevents index use, resulting in a sequential scan and increased overhead from row-by-row computation.

- Index Usage: No. The arithmetic expression alters `person_id`, making direct index use impossible, leading PostgreSQL to perform a sequential scan.

6. Complex Condition Query Analysis:

Query: `EXPLAIN ANALYZE SELECT * FROM persons WHERE person_id + 5 > 1000 AND person_id * 2 < 3000;`

- Total Cost: 225.72

- Is the Cost Higher or Lower than Query 2? Higher

- Reason: Calculations on `person_id` prevent index usage, forcing a full sequential scan and row-by-row evaluation, adding processing load.

- Index Usage: No, due to the calculations involved, PostgreSQL defaults to a sequential scan.

7. Group By Query Analysis:

Query: `EXPLAIN ANALYZE SELECT person_age, COUNT(person_id) FROM persons GROUP BY person_age;`

- Total Cost: 199.79

- Why is the Start-up Cost Similar to the Total Cost? The query scans the entire table to count `person_id` for each unique `person_age`, with most effort on scanning, causing similar start-up and total costs.

8. Group By with Index:

Query: Same as Query 7

- Total Cost: 199.79

- Comparison to Query 6: The same cost as Query 6, as they perform identical operations.

9. Optimized Group By Query with Index:

Query: `EXPLAIN ANALYZE SELECT person_age, COUNT(person_age) FROM persons GROUP BY person_age;`

- Total Cost: Lower than Query 6 and 7

- Explanation: An index on `person_age` allows faster grouping and counting, enabling PostgreSQL to organize rows by age using the index, reducing time and resource usage.

10. Join Query Optimization:

Original Query:

```
EXPLAIN ANALYZE

SELECT joblist.job_id, joblist.job_description, joblist.salary, job_person.person_id, person.person_name

FROM joblist

INNER JOIN job_person ON joblist.job_id = job_person.job_id

WHERE job_person.job_id = 34;
```

- Total Cost: 3129.15

Optimization Steps:

- Indexes Created:

```
CREATE INDEX idx_joblist_job_id ON joblist(job_id);

CREATE INDEX idx_job_person_job_id ON job_person(job_id);
```

Optimized Query:

```
EXPLAIN ANALYZE

SELECT joblist.job_id, joblist.job_description, joblist.salary, job_person.person_id

FROM joblist

INNER JOIN job_person ON joblist.job_id = job_person.job_id

WHERE job_person.job_id = 34;
```

- Final Cost of Optimized Query: 1544.93