you are an expert in SQL, building data structures for Data Ware House - data lake and all, writing optimized queries, explaining required concepts in very detail as if to a high scholler

**ChatGPT said:**

ChatGPT

Got it! I'll explain and assist with SQL, data warehousing, data lakes, and related concepts in a clear and detailed way, as if teaching someone new to these topics. Let me know where you'd like to begin or what you're curious about! 😊

4o

**You said:**

Dimensional Data Modeling Day 1 Lecture In this lecture, we talk about one big table vs relational vs dimensional data modeling and their tradeoffs! explain in detail with examples, sample data and outputs, take nba players and seasons dataset as example

2/2

**ChatGPT said:**

ChatGPT

**Dimensional Data Modeling: Day 1 Lecture**

Welcome to **Day 1** of learning Dimensional Data Modeling! Today, we'll explore three key data modeling approaches:

1. **Single Big Table Modeling**
2. **Relational Modeling**
3. **Dimensional Modeling**

We’ll also discuss **tradeoffs** of each approach, using NBA players and seasons as our dataset example. At the end, you'll understand why dimensional modeling is the preferred choice for **data warehouses**.

**Key Concepts**

1. **Single Big Table Modeling**:
   * All the data is stored in one massive table.
   * Simple to design and query.
   * Good for small datasets but becomes inefficient as data grows.
2. **Relational Modeling**:
   * Data is split into normalized tables to avoid redundancy (e.g., players, teams, seasons).
   * Excellent for transactional systems (OLTP), like apps needing frequent updates.
   * Can be slow for analytical purposes due to complex joins.
3. **Dimensional Modeling**:
   * Data is organized into **facts** and **dimensions**.
   * Best for analytical systems (OLAP) like data warehouses.
   * Balances storage efficiency and query performance.

**NBA Dataset Example**

We’ll use a fictional NBA dataset with the following details:

* **Players**: Names, teams, positions.
* **Seasons**: Year, stats like points, assists, and rebounds.
* **Teams**: Team details like name and city.

**1. Single Big Table Modeling**

**Example Structure**

A single table might look like this:

| **Player Name** | **Season** | **Team Name** | **Points** | **Assists** | **Rebounds** | **Position** | **City** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| LeBron James | 2023 | Lakers | 30.2 | 8.4 | 7.6 | Forward | Los Angeles |
| Stephen Curry | 2023 | Warriors | 29.4 | 6.3 | 5.2 | Guard | San Francisco |
| LeBron James | 2022 | Lakers | 28.9 | 7.6 | 7.8 | Forward | Los Angeles |

**Pros:**

* **Simple structure**: All data in one place.
* **Easy to query**: No joins needed; e.g., SELECT \* FROM table WHERE Player = 'LeBron James'.

**Cons:**

* **Redundancy**: "Lakers" and "Los Angeles" are repeated many times.
* **Poor scalability**: Larger datasets result in wasted storage and slower queries.
* **Not flexible**: Hard to adapt if we add new data (e.g., MVP awards).

**2. Relational Modeling**

**Example Structure**

We normalize the data into separate tables to avoid redundancy:

1. **Players Table**:

| **Player ID** | **Name** | **Position** |
| --- | --- | --- |
| 1 | LeBron James | Forward |
| 2 | Stephen Curry | Guard |

1. **Teams Table**:

| **Team ID** | **Name** | **City** |
| --- | --- | --- |
| 1 | Lakers | Los Angeles |
| 2 | Warriors | San Francisco |

1. **Seasons Table**:

| **Season ID** | **Year** |
| --- | --- |
| 1 | 2023 |
| 2 | 2022 |

1. **Player Stats Table**:

| **Player ID** | **Season ID** | **Team ID** | **Points** | **Assists** | **Rebounds** |
| --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 30.2 | 8.4 | 7.6 |
| 2 | 1 | 2 | 29.4 | 6.3 | 5.2 |

**Pros:**

* **Eliminates redundancy**: Team and player data are stored only once.
* **Scalable**: Adding new attributes or entities is easier.

**Cons:**

* **Complex queries**: Need multiple joins to combine data. Example:

sql

Copy code

SELECT p.Name, t.Name AS Team, s.Year, ps.Points

FROM Player\_Stats ps

JOIN Players p ON ps.Player\_ID = p.Player\_ID

JOIN Teams t ON ps.Team\_ID = t.Team\_ID

JOIN Seasons s ON ps.Season\_ID = s.Season\_ID

WHERE p.Name = 'LeBron James';

* **Not ideal for reporting**: Analytical queries are slower due to joins.

**3. Dimensional Modeling**

**Example Structure**

Dimensional modeling organizes data into **facts** and **dimensions**:

1. **Fact Table**: Stores measurable data (**facts**), like points, assists, and rebounds.
   * **Fact\_Player\_Stats**:

| **Player ID** | **Season** | **Team ID** | **Points** | **Assists** | **Rebounds** |
| --- | --- | --- | --- | --- | --- |
| 1 | 2023 | 1 | 30.2 | 8.4 | 7.6 |
| 2 | 2023 | 2 | 29.4 | 6.3 | 5.2 |

1. **Dimension Tables**: Provide descriptive data about players, teams, and seasons.
   * **Dim\_Player**:

| **Player ID** | **Name** | **Position** |
| --- | --- | --- |
| 1 | LeBron James | Forward |
| 2 | Stephen Curry | Guard |

* + **Dim\_Team**:

| **Team ID** | **Name** | **City** |
| --- | --- | --- |
| 1 | Lakers | Los Angeles |
| 2 | Warriors | San Francisco |

* + **Dim\_Season**:

| **Season** | **Year** |
| --- | --- |
| 2023 | 2023 |
| 2022 | 2022 |

**Pros:**

* **Optimized for analysis**: Queries are faster and simpler.

sql

Copy code

SELECT dp.Name, dt.Name AS Team, ds.Year, fps.Points

FROM Fact\_Player\_Stats fps

JOIN Dim\_Player dp ON fps.Player\_ID = dp.Player\_ID

JOIN Dim\_Team dt ON fps.Team\_ID = dt.Team\_ID

JOIN Dim\_Season ds ON fps.Season = ds.Season

WHERE dp.Name = 'LeBron James';

* **Easier to report**: Dimension tables can grow without impacting fact tables.
* **Supports star schema**: Central fact table surrounded by dimensions.

**Cons:**

* Slightly more complex to design compared to a single table.
* Fact and dimension separation may require extra initial effort.

**Tradeoffs Summary**

| **Feature** | **Single Big Table** | **Relational Model** | **Dimensional Model** |
| --- | --- | --- | --- |
| **Simplicity** | ✅ Simple | ❌ Complex Joins | ✅ Balanced |
| **Redundancy** | ❌ High | ✅ Minimal | ✅ Minimal |
| **Query Performance** | ❌ Poor for large data | ❌ Slow for analytics | ✅ Optimized |
| **Scalability** | ❌ Limited | ✅ Flexible | ✅ Flexible |
| **Use Case** | Small datasets | Transactional systems | Analytical systems |

**Conclusion**

Dimensional modeling is the **best choice for data warehousing** because it balances scalability, performance, and simplicity. It allows for **fast queries**, reduces redundancy, and is well-suited for reporting and analysis tasks.

In the next lecture, we’ll dive deeper into **star schemas** and **fact table design**! 🚀

4o

**You said:**

explain in detail, if possible go through the github repo also, consider the same dataset for explanation: Dimensional Data Modeling Day 1 Lab In this video will be building a cumulative table design (https://github.com/EcZachly/cumulative-table-design) with complex data types STRUCT and ARRAY. We will take the table player\_seasons and create a new table called players that holds onto an historical record of all NBA players and the seasons that they played!

**ChatGPT said:**

ChatGPT

Searched 3 sites

The "Dimensional Data Modeling Day 1 Lab" on cumulative table design introduces a method to maintain historical records of data efficiently using STRUCT and ARRAY data types. The GitHub repository you referenced contains scripts for creating cumulative tables, which can be adapted for datasets like NBA players and seasons.

**Key Concepts of the Lab:**

1. **Cumulative Table Design**:
   * This approach aggregates historical data into a single table for simplified access and storage efficiency.
   * The table includes arrays and nested structures to maintain all related historical data.
2. **STRUCT and ARRAY**:
   * **STRUCT**: Groups related fields into a single data type (e.g., player stats for a season).
   * **ARRAY**: Stores lists of values, such as an array of seasons for a player.
3. **NBA Players and Seasons Example**:
   * Starting with a player\_seasons table, we can build a players table that stores all historical seasons for each player.

**Example Data Transformation:**

**Initial player\_seasons Table**:

| **PlayerID** | **Season** | **Points** | **Assists** | **Rebounds** |
| --- | --- | --- | --- | --- |
| 1 | 2021 | 1000 | 300 | 500 |
| 1 | 2022 | 1200 | 350 | 550 |
| 2 | 2021 | 800 | 200 | 400 |

**Transformed players Table with ARRAY**:

| **PlayerID** | **Seasons** | **Points** | **Assists** | **Rebounds** |
| --- | --- | --- | --- | --- |
| 1 | [2021, 2022] | [1000, 1200] | [300, 350] | [500, 550] |
| 2 | [2021] | [800] | [200] | [400] |

Here, arrays consolidate all season-wise data for each player.

**SQL Example Using STRUCT and ARRAY**:

sql

Copy code

CREATE TABLE players AS

SELECT

PlayerID,

ARRAY\_AGG(Season) AS Seasons,

ARRAY\_AGG(Points) AS Points,

ARRAY\_AGG(Assists) AS Assists,

ARRAY\_AGG(Rebounds) AS Rebounds

FROM player\_seasons

GROUP BY PlayerID;

**Advanced Concepts:**

The repository also implements DAGs (Directed Acyclic Graphs) in Apache Airflow for automated processing:

* A **daily task** computes incremental updates to the cumulative table.
* A **cumulative task** aggregates historical data, ensuring consistency with dependencies.

This design improves performance for queries requiring historical insights while minimizing redundancy and storage costs. For more details and code examples, visit the [GitHub repository](https://github.com/EcZachly/cumulative-table-design).

4o

**You said:**

Dimensional Data Modeling Day 2 Lecture We cover slowly changing dimensions and idempotent pipelines in this lecture! Enjoy! Dimensional Data Modeling Day 2 Lab In this video we build an slowly changing dimension type 2 model on top of the players table we built in the first lab! EXPLAIN IN DETAIL WITH EXAMPLES

**ChatGPT said:**

ChatGPT

**Dimensional Data Modeling Day 2: Slowly Changing Dimensions (SCDs) and Idempotent Pipelines**

In Day 2, we dive deeper into **Slowly Changing Dimensions (SCDs)** and how to implement them in a **data pipeline** using the NBA dataset. Let’s break it down with clear concepts, examples, and step-by-step guidance.

**Part 1: Slowly Changing Dimensions (SCDs)**

A **dimension table** contains descriptive attributes, like player names, teams, and positions. These attributes may **change over time**, which leads to the concept of **Slowly Changing Dimensions (SCDs)**. There are different strategies to handle these changes:

**SCD Types Overview**

1. **SCD Type 1**: Overwrite the old value with the new value. No history is kept.
2. **SCD Type 2**: Maintain a full history by creating a new row for each change.
3. **SCD Type 3**: Store the current and previous values in the same row.
4. **SCD Type 4**: Use a separate historical table.
5. **SCD Type 6**: A hybrid of Types 1, 2, and 3.

For today’s lab, we focus on **SCD Type 2**.

**SCD Type 2: Example with NBA Players**

SCD Type 2 creates a new row in the dimension table whenever there’s a change in the descriptive attributes, along with metadata to track the changes (e.g., start date, end date, active flag).

**Players Table Before Change**:

| **PlayerID** | **PlayerName** | **Team** | **StartDate** | **EndDate** | **IsActive** |
| --- | --- | --- | --- | --- | --- |
| 1 | LeBron James | Cavaliers | 2010-01-01 | 2018-07-01 | No |
| 1 | LeBron James | Lakers | 2018-07-02 | NULL | Yes |

**Scenario**: LeBron moves to the Warriors in 2024.

**Players Table After Change**:

| **PlayerID** | **PlayerName** | **Team** | **StartDate** | **EndDate** | **IsActive** |
| --- | --- | --- | --- | --- | --- |
| 1 | LeBron James | Cavaliers | 2010-01-01 | 2018-07-01 | No |
| 1 | LeBron James | Lakers | 2018-07-02 | 2024-01-01 | No |
| 1 | LeBron James | Warriors | 2024-01-02 | NULL | Yes |

**Part 2: Building SCD Type 2 in a Data Pipeline**

To implement SCD Type 2, you need a pipeline that:

1. **Detects Changes**: Compares the incoming data with the current dimension table.
2. **Updates the Dimension Table**:
   * Ends existing records when changes are detected.
   * Inserts new records for changed attributes.
3. **Maintains Metadata**: Tracks effective dates and active status.

**Step-by-Step Implementation**

1. **Input Data**:

| **PlayerID** | **PlayerName** | **Team** | **EffectiveDate** |
| --- | --- | --- | --- |
| 1 | LeBron James | Warriors | 2024-01-02 |
| 2 | Stephen Curry | Warriors | 2024-01-02 |

1. **Existing Dimension Table**:

| **PlayerID** | **PlayerName** | **Team** | **StartDate** | **EndDate** | **IsActive** |
| --- | --- | --- | --- | --- | --- |
| 1 | LeBron James | Lakers | 2018-07-02 | NULL | Yes |

1. **Pipeline Logic**:
   * **Detect Changes**: Identify records where the new team differs from the existing table.
   * **End Old Record**: Update the EndDate and set IsActive = No for the previous record.
   * **Insert New Record**: Add a new row with the updated team.

**SQL Example**:

sql

Copy code

-- Step 1: Update the old record

UPDATE players

SET EndDate = '2024-01-01', IsActive = 'No'

WHERE PlayerID = 1 AND IsActive = 'Yes';

-- Step 2: Insert the new record

INSERT INTO players (PlayerID, PlayerName, Team, StartDate, EndDate, IsActive)

VALUES (1, 'LeBron James', 'Warriors', '2024-01-02', NULL, 'Yes');

**Part 3: Idempotent Pipelines**

An **idempotent pipeline** ensures that running the same process multiple times produces the same results, avoiding duplicate or incorrect updates. This is critical in data engineering to maintain consistency.

**Designing an Idempotent Pipeline for SCD Type 2:**

1. **Key Checks**:
   * Before updating, check if the EndDate has already been set.
   * Before inserting, check if the new record already exists.
2. **SQL with Idempotence**:

sql

Copy code

-- Step 1: End old records if not already ended

UPDATE players

SET EndDate = '2024-01-01', IsActive = 'No'

WHERE PlayerID = 1 AND IsActive = 'Yes' AND EndDate IS NULL;

-- Step 2: Insert only if the record doesn't already exist

INSERT INTO players (PlayerID, PlayerName, Team, StartDate, EndDate, IsActive)

SELECT 1, 'LeBron James', 'Warriors', '2024-01-02', NULL, 'Yes'

WHERE NOT EXISTS (

SELECT 1 FROM players

WHERE PlayerID = 1 AND Team = 'Warriors' AND IsActive = 'Yes'

);

**Summary**

1. **SCD Type 2** ensures historical changes are tracked by inserting new rows for changes.
2. **Idempotent pipelines** prevent duplicate or inconsistent updates in case of reprocessing.
3. **Practical Example**: Tracking NBA player team changes while maintaining accurate historical records.
4. In the lab, you'll implement this logic using SQL or a tool like Apache Airflow to automate the process.

By understanding these techniques, you'll be equipped to handle dynamic changes in data warehouses! 🚀

4o

**You said:**

CREATE A DETAIL EXAMPLE IN DETAIL which covers day 1 and day 2 topics, WITH DETAIL SOLUTION SO THAT I CAN UNDERSTAND THE CONCEPT VERY CLEARLY, use NBA dataset only,

2/2

**ChatGPT said:**

ChatGPT

Let’s create a comprehensive example that combines **Day 1 (Dimensional Modeling)** and **Day 2 (SCD Type 2 and Idempotent Pipelines)** using the NBA dataset. By the end of this example, you will have a clear understanding of these concepts.

**Scenario Overview**

We’re building a data warehouse for an NBA analytics system. The goal is to:

1. **Day 1**: Model the data using dimensional modeling.
   * Fact table: Player statistics.
   * Dimension tables: Players, Teams, Seasons.
2. **Day 2**: Implement **SCD Type 2** for tracking changes in player information (e.g., team transfers).
3. Design idempotent pipelines to ensure consistent and reliable updates.

**Data Example**

**Source Tables**

1. **Raw Player Data (player\_raw)**:

| **PlayerID** | **PlayerName** | **TeamName** | **Position** | **EffectiveDate** |
| --- | --- | --- | --- | --- |
| 1 | LeBron James | Lakers | Forward | 2018-07-02 |
| 2 | Stephen Curry | Warriors | Guard | 2009-10-27 |
| 1 | LeBron James | Warriors | Forward | 2024-01-01 |

1. **Game Statistics (player\_stats)**:

| **PlayerID** | **Season** | **Points** | **Assists** | **Rebounds** |
| --- | --- | --- | --- | --- |
| 1 | 2023 | 1600 | 450 | 500 |
| 2 | 2023 | 1800 | 700 | 400 |

**Part 1: Dimensional Modeling**

We’ll model the data into a **Star Schema** with one fact table and three dimension tables.

**Dimension Tables**

1. **Player Dimension (dim\_player)**:
   * Stores details about players.
   * Handles historical changes using SCD Type 2.

| **PlayerID** | **PlayerName** | **TeamName** | **Position** | **StartDate** | **EndDate** | **IsActive** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | LeBron James | Lakers | Forward | 2018-07-02 | 2023-12-31 | No |
| 1 | LeBron James | Warriors | Forward | 2024-01-01 | NULL | Yes |
| 2 | Stephen Curry | Warriors | Guard | 2009-10-27 | NULL | Yes |

1. **Team Dimension (dim\_team)**:
   * Details about teams.

| **TeamID** | **TeamName** | **City** |
| --- | --- | --- |
| 1 | Lakers | Los Angeles |
| 2 | Warriors | San Francisco |

1. **Season Dimension (dim\_season)**:
   * Information about seasons.

| **SeasonID** | **Year** |
| --- | --- |
| 1 | 2023 |
| 2 | 2024 |

**Fact Table**

1. **Player Statistics Fact (fact\_player\_stats)**:
   * Links dimensions and stores measurable data.

| **PlayerID** | **TeamID** | **SeasonID** | **Points** | **Assists** | **Rebounds** |
| --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1600 | 450 | 500 |
| 2 | 2 | 1 | 1800 | 700 | 400 |

**Part 2: Slowly Changing Dimensions (SCD Type 2)**

We’ll implement SCD Type 2 for the dim\_player table to track player transfers.

**Steps to Update the Dimension Table**

1. **Detect Changes**:
   * Compare new data (player\_raw) with the existing dimension table (dim\_player) to find:
     + New players.
     + Players whose attributes (e.g., team) have changed.
2. **End Existing Records**:
   * For players whose data has changed, update the EndDate and set IsActive = No.
3. **Insert New Records**:
   * Add a new row for the updated player data with a new StartDate.

**SQL Implementation**

**Step 1: Detect Changes**

sql

Copy code

-- Identify players with changes

SELECT

r.PlayerID,

r.PlayerName,

r.TeamName,

r.Position,

r.EffectiveDate

FROM player\_raw r

LEFT JOIN dim\_player p

ON r.PlayerID = p.PlayerID

AND p.IsActive = 'Yes'

WHERE (

r.TeamName != p.TeamName

OR r.Position != p.Position

OR p.PlayerID IS NULL

);

**Step 2: End Existing Records**

sql

Copy code

-- Mark previous records as inactive

UPDATE dim\_player

SET EndDate = '2023-12-31', IsActive = 'No'

WHERE PlayerID = 1 AND IsActive = 'Yes';

**Step 3: Insert New Records**

sql

Copy code

-- Insert new records with updated data

INSERT INTO dim\_player (PlayerID, PlayerName, TeamName, Position, StartDate, EndDate, IsActive)

VALUES (1, 'LeBron James', 'Warriors', 'Forward', '2024-01-01', NULL, 'Yes');

**Part 3: Idempotent Pipelines**

To make the pipeline idempotent:

1. **Check Before Updating**:
   * Ensure you don’t update records that are already ended.
2. **Check Before Inserting**:
   * Ensure new records don’t already exist.

**Idempotent SQL Example**:

sql

Copy code

-- End existing records only if they are active and not already ended

UPDATE dim\_player

SET EndDate = '2023-12-31', IsActive = 'No'

WHERE PlayerID = 1 AND IsActive = 'Yes' AND EndDate IS NULL;

-- Insert new records only if they don’t already exist

INSERT INTO dim\_player (PlayerID, PlayerName, TeamName, Position, StartDate, EndDate, IsActive)

SELECT r.PlayerID, r.PlayerName, r.TeamName, r.Position, r.EffectiveDate, NULL, 'Yes'

FROM player\_raw r

LEFT JOIN dim\_player p

ON r.PlayerID = p.PlayerID AND r.TeamName = p.TeamName AND p.IsActive = 'Yes'

WHERE p.PlayerID IS NULL;

**Final Dimension and Fact Tables**

**Final dim\_player:**

| **PlayerID** | **PlayerName** | **TeamName** | **Position** | **StartDate** | **EndDate** | **IsActive** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | LeBron James | Lakers | Forward | 2018-07-02 | 2023-12-31 | No |
| 1 | LeBron James | Warriors | Forward | 2024-01-01 | NULL | Yes |
| 2 | Stephen Curry | Warriors | Guard | 2009-10-27 | NULL | Yes |

**Final fact\_player\_stats:**

| **PlayerID** | **TeamID** | **SeasonID** | **Points** | **Assists** | **Rebounds** |
| --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1600 | 450 | 500 |
| 2 | 2 | 1 | 1800 | 700 | 400 |

**Conclusion**

This example covers:

1. **Dimensional Modeling**: Fact and dimension tables designed for analytics.
2. **SCD Type 2**: Tracking changes over time with historical data.
3. **Idempotent Pipelines**: Ensuring reliable updates.

By following these steps, you can build a robust NBA data warehouse! 🚀

1/2

4o

**You said:**

automate this above task using Apache Spark in python, consider more data for better understanding, with this example i have to clearly understand the topics and also become expert in implementing in Spark

**ChatGPT said:**

ChatGPT

To implement the above dimensional modeling, Slowly Changing Dimensions (SCD Type 2), and idempotent pipelines in **Apache Spark with Python**, let’s follow a practical and scalable approach. This example includes data ingestion, transformation, and SCD Type 2 implementation, ensuring a clear understanding.

**Scenario Overview**

* **Source Data**:
  + Raw player data containing historical and updated records (player\_raw).
  + Player statistics data (player\_stats).
* **Output Tables**:
  + **dim\_player**: SCD Type 2 dimension table for player details.
  + **fact\_player\_stats**: Fact table for player statistics.
* **Technology**:
  + **Apache Spark** (PySpark): Scalable for handling large datasets.
  + **Spark SQL and DataFrame API**: For transformations and querying.

**Step 1: Environment Setup**

Before you start, ensure you have a Spark environment configured. Use pyspark in a local or distributed cluster.

python

Copy code

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, lit, when, array, current\_date

from pyspark.sql.window import Window

# Initialize Spark session

spark = SparkSession.builder \

.appName("NBA Dimensional Modeling with SCD") \

.getOrCreate()

**Step 2: Sample Input Data**

**Raw Player Data (player\_raw)**

python

Copy code

# Sample raw player data

player\_raw\_data = [

(1, "LeBron James", "Lakers", "Forward", "2018-07-02"),

(2, "Stephen Curry", "Warriors", "Guard", "2009-10-27"),

(1, "LeBron James", "Warriors", "Forward", "2024-01-01")

]

columns\_raw = ["PlayerID", "PlayerName", "TeamName", "Position", "EffectiveDate"]

player\_raw\_df = spark.createDataFrame(player\_raw\_data, columns\_raw)

player\_raw\_df.show()

**Player Statistics Data (player\_stats)**

python

Copy code

# Sample player statistics data

player\_stats\_data = [

(1, "2023", 1600, 450, 500),

(2, "2023", 1800, 700, 400)

]

columns\_stats = ["PlayerID", "Season", "Points", "Assists", "Rebounds"]

player\_stats\_df = spark.createDataFrame(player\_stats\_data, columns\_stats)

player\_stats\_df.show()

**Step 3: Existing Dimension Table (dim\_player)**

The existing dimension table tracks player records with SCD Type 2 fields.

python

Copy code

# Sample existing dimension table

dim\_player\_data = [

(1, "LeBron James", "Lakers", "Forward", "2018-07-02", "2023-12-31", "No"),

(1, "LeBron James", "Warriors", "Forward", "2024-01-01", None, "Yes"),

(2, "Stephen Curry", "Warriors", "Guard", "2009-10-27", None, "Yes")

]

columns\_dim = ["PlayerID", "PlayerName", "TeamName", "Position", "StartDate", "EndDate", "IsActive"]

dim\_player\_df = spark.createDataFrame(dim\_player\_data, columns\_dim)

dim\_player\_df.show()

**Step 4: Detect Changes in player\_raw**

Compare the incoming raw data (player\_raw) with the current dimension table (dim\_player) to identify changes.

python

Copy code

# Join raw data with the current dimension table

joined\_df = player\_raw\_df.alias("raw").join(

dim\_player\_df.alias("dim"),

(col("raw.PlayerID") == col("dim.PlayerID")) & (col("dim.IsActive") == "Yes"),

"left"

)

# Detect changes

changes\_df = joined\_df.filter(

(col("raw.TeamName") != col("dim.TeamName")) |

(col("raw.Position") != col("dim.Position")) |

col("dim.PlayerID").isNull()

).select("raw.\*")

changes\_df.show()

**Step 5: End Existing Records**

Mark existing active records as inactive if changes are detected.

python

Copy code

# End old records for changed players

updated\_dim\_df = dim\_player\_df.withColumn(

"EndDate",

when(col("IsActive") == "Yes", current\_date()).otherwise(col("EndDate"))

).withColumn(

"IsActive",

when(col("IsActive") == "Yes", "No").otherwise(col("IsActive"))

)

updated\_dim\_df.show()

**Step 6: Insert New Records**

Insert new records into the dim\_player table for detected changes.

python

Copy code

# Prepare new records with SCD Type 2 fields

new\_records\_df = changes\_df.withColumn("StartDate", col("EffectiveDate")) \

.withColumn("EndDate", lit(None).cast("date")) \

.withColumn("IsActive", lit("Yes"))

# Combine with the updated dimension table

final\_dim\_player\_df = updated\_dim\_df.union(new\_records\_df)

final\_dim\_player\_df.show()

**Step 7: Create Fact Table (fact\_player\_stats)**

Join player statistics with the updated dimension table to build the fact table.

python

Copy code

# Join with dimension table to get TeamID

team\_mapping = {

"Lakers": 1,

"Warriors": 2

}

team\_mapping\_df = spark.createDataFrame(list(team\_mapping.items()), ["TeamName", "TeamID"])

fact\_player\_stats\_df = player\_stats\_df.join(

final\_dim\_player\_df.filter(col("IsActive") == "Yes").select("PlayerID", "TeamName"),

"PlayerID"

).join(

team\_mapping\_df, "TeamName"

).select("PlayerID", "TeamID", "Season", "Points", "Assists", "Rebounds")

fact\_player\_stats\_df.show()

**Step 8: Automate with a Function**

Combine all the steps into a reusable pipeline function.

python

Copy code

def process\_scd2(player\_raw\_df, player\_stats\_df, dim\_player\_df):

# Step 4: Detect Changes

changes\_df = player\_raw\_df.alias("raw").join(

dim\_player\_df.alias("dim"),

(col("raw.PlayerID") == col("dim.PlayerID")) & (col("dim.IsActive") == "Yes"),

"left"

).filter(

(col("raw.TeamName") != col("dim.TeamName")) |

(col("raw.Position") != col("dim.Position")) |

col("dim.PlayerID").isNull()

).select("raw.\*")

# Step 5: End Existing Records

updated\_dim\_df = dim\_player\_df.withColumn(

"EndDate",

when(col("IsActive") == "Yes", current\_date()).otherwise(col("EndDate"))

).withColumn(

"IsActive",

when(col("IsActive") == "Yes", "No").otherwise(col("IsActive"))

)

# Step 6: Insert New Records

new\_records\_df = changes\_df.withColumn("StartDate", col("EffectiveDate")) \

.withColumn("EndDate", lit(None).cast("date")) \

.withColumn("IsActive", lit("Yes"))

final\_dim\_player\_df = updated\_dim\_df.union(new\_records\_df)

# Step 7: Create Fact Table

team\_mapping\_df = spark.createDataFrame(list(team\_mapping.items()), ["TeamName", "TeamID"])

fact\_player\_stats\_df = player\_stats\_df.join(

final\_dim\_player\_df.filter(col("IsActive") == "Yes").select("PlayerID", "TeamName"),

"PlayerID"

).join(

team\_mapping\_df, "TeamName"

).select("PlayerID", "TeamID", "Season", "Points", "Assists", "Rebounds")

return final\_dim\_player\_df, fact\_player\_stats\_df

# Run the pipeline

final\_dim\_player\_df, fact\_player\_stats\_df = process\_scd2(player\_raw\_df, player\_stats\_df, dim\_player\_df)

**Summary**

* **Dimensional Modeling**: Designed dim\_player and fact\_player\_stats tables.
* **SCD Type 2**: Implemented history tracking with StartDate, EndDate, and IsActive.
* **Spark Pipeline**: Automated detection, updates, and inserts in a scalable manner.

By running this script, you now have a full implementation of dimensional modeling and SCD Type 2 using Apache Spark!

4o

**You said:**

create a task to practice the above, create a mid size sample data with atleast 20 rows, create tasks to implement all the 2 modules, indivudually and combined, create tasks in detail. only tasks nothing else.

**ChatGPT said:**

ChatGPT

**Practice Tasks: Dimensional Modeling and SCD Type 2 Implementation**

**Dataset Preparation**

Create two datasets with **at least 20 rows each**:

1. **Player Data (player\_raw)**:
   * Columns: PlayerID, PlayerName, TeamName, Position, EffectiveDate.
   * Contains historical and current records for players, including changes to TeamName or Position.
2. **Player Statistics (player\_stats)**:
   * Columns: PlayerID, Season, Points, Assists, Rebounds.
   * Contains statistics for players across different seasons.

**Module 1: Dimensional Modeling Tasks**

1. **Design the Dimensional Model**
   * Identify the fact table and dimension tables.
   * Define relationships between dimensions and the fact table.
   * Map all columns to the respective tables.
   * Create sketches or diagrams of the schema.
2. **Build Dimension Tables**
   * Implement the following dimension tables:
     + dim\_player: Tracks player details (PlayerID, PlayerName, TeamName, Position, StartDate, EndDate, IsActive).
     + dim\_team: Tracks team details (TeamID, TeamName, City).
     + dim\_season: Tracks season details (SeasonID, Year).
   * Populate the dimension tables with sample data.
3. **Build the Fact Table**
   * Implement the fact\_player\_stats table with columns: PlayerID, TeamID, SeasonID, Points, Assists, Rebounds.
   * Populate the fact table with sample data, ensuring it connects properly with the dimensions.
4. **Run Sample Queries**
   * Write and execute queries to analyze:
     + Top-performing players in specific seasons.
     + Total points scored by players on a particular team.

**Module 2: Slowly Changing Dimensions (SCD Type 2) Tasks**

1. **SCD Type 2 Design**
   * Define the fields for the dim\_player table to track history:
     + StartDate, EndDate, and IsActive.
2. **Detect Changes in Data**
   * Compare incoming player\_raw data with the existing dim\_player table.
   * Identify records where:
     + TeamName has changed.
     + Position has changed.
     + New players have been added.
3. **Update Existing Records**
   * Write a transformation to mark existing active records as inactive when changes are detected.
   * Update the EndDate field for these records.
4. **Insert New Records**
   * Add new records for players whose data has changed.
   * Ensure new records have the correct StartDate, EndDate, and IsActive fields.
5. **Validate Updates**
   * Write queries to verify that:
     + Historical changes are accurately recorded.
     + Only one active record exists per player at any given time.

**Module 3: Combine Dimensional Modeling and SCD Type 2**

1. **Automate the Entire Workflow**
   * Combine tasks from both modules into a single automated process using Apache Spark or SQL scripts:
     + Ingest raw data.
     + Detect changes.
     + Update dim\_player using SCD Type 2 logic.
     + Populate the fact table (fact\_player\_stats).
2. **Create an Idempotent Pipeline**
   * Ensure that running the pipeline multiple times does not produce duplicate records or corrupt data.
   * Implement safeguards, such as:
     + Checking before updating records.
     + Validating new inserts.
3. **Run End-to-End Testing**
   * Execute the pipeline on the sample dataset and verify the following:
     + All dimensional tables and the fact table are updated correctly.
     + Queries against the tables return expected results.
4. **Write Analytics Queries**
   * Query the final tables to analyze:
     + Player performance trends over time.
     + Players with the most points in a specific year.
     + Impact of team transfers on player performance.

**Dataset Requirements**

* Include multiple rows per player in player\_raw to simulate historical changes.
* Include at least 5 seasons and ensure statistics cover multiple seasons per player in player\_stats.

By completing these tasks, you will gain hands-on experience in **dimensional modeling**, **SCD Type 2 implementation**, and building robust pipelines using tools like Apache Spark.

4o

**You said:**

create a different type od dataset : actor\_films with data

**ChatGPT said:**

ChatGPT

**Dataset: actor\_films**

The actor\_films dataset tracks actors, films they've starred in, their roles in those films, and the years the films were released. This can be used for understanding actor-film relationships, tracking the career trajectories of actors, or for analyzing trends in the film industry.

**Dataset Description:**

* **Table Name**: actor\_films
* **Columns**:
  1. ActorID: Unique identifier for each actor (integer).
  2. ActorName: Name of the actor (string).
  3. FilmID: Unique identifier for each film (integer).
  4. FilmTitle: Title of the film (string).
  5. Role: The character or role the actor portrayed (string).
  6. ReleaseYear: Year the film was released (integer).
  7. Rating: Film rating (numeric, float between 1 and 10).
  8. BoxOfficeRevenue: Revenue generated by the film (numeric, in millions).
  9. Genre: Genre of the film (string).

**Sample Data:**

Here’s a sample of data in the actor\_films dataset:

| **ActorID** | **ActorName** | **FilmID** | **FilmTitle** | **Role** | **ReleaseYear** | **Rating** | **BoxOfficeRevenue** | **Genre** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | Robert Downey Jr. | 101 | Iron Man | Tony Stark | 2008 | 8.5 | 585.2 | Action |
| 1 | Robert Downey Jr. | 102 | The Avengers | Tony Stark | 2012 | 8.0 | 1500.0 | Action |
| 2 | Scarlett Johansson | 101 | Iron Man | Natasha Romanoff | 2008 | 8.5 | 585.2 | Action |
| 2 | Scarlett Johansson | 103 | Lost in Translation | Charlotte | 2003 | 7.5 | 44.0 | Drama |
| 3 | Chris Hemsworth | 104 | Thor | Thor | 2011 | 7.0 | 450.0 | Action |
| 3 | Chris Hemsworth | 105 | Avengers: Endgame | Thor | 2019 | 8.5 | 279780.0 | Action |
| 4 | Tom Hanks | 106 | Forrest Gump | Forrest Gump | 1994 | 8.8 | 678.0 | Drama |
| 4 | Tom Hanks | 107 | Cast Away | Chuck Noland | 2000 | 7.8 | 429.0 | Drama |
| 5 | Meryl Streep | 108 | The Devil Wears Prada | Miranda Priestly | 2006 | 6.9 | 326.0 | Comedy |
| 5 | Meryl Streep | 109 | Doubt | Sister Aloysius | 2008 | 7.5 | 8.0 | Drama |
| 6 | Denzel Washington | 110 | Training Day | Alonzo Harris | 2001 | 8.3 | 104.0 | Crime/Thriller |
| 6 | Denzel Washington | 111 | Fences | Troy Maxson | 2016 | 7.2 | 64.0 | Drama |
| 7 | Leonardo DiCaprio | 112 | Titanic | Jack Dawson | 1997 | 7.8 | 2187.0 | Drama/Romance |
| 7 | Leonardo DiCaprio | 113 | The Revenant | Hugh Glass | 2015 | 8.0 | 532.0 | Drama/Thriller |
| 8 | Jennifer Lawrence | 114 | The Hunger Games | Katniss Everdeen | 2012 | 7.2 | 694.0 | Action/Sci-Fi |
| 8 | Jennifer Lawrence | 115 | Silver Linings Playbook | Tiffany Maxwell | 2012 | 7.7 | 236.0 | Romance/Drama |
| 9 | Will Smith | 116 | Men in Black | Agent J | 1997 | 7.3 | 589.0 | Sci-Fi/Action |
| 9 | Will Smith | 117 | Pursuit of Happyness | Chris Gardner | 2006 | 8.0 | 307.0 | Drama |
| 10 | Emma Stone | 118 | La La Land | Mia Dolan | 2016 | 8.0 | 446.0 | Romance/Drama |

**Explanation of Columns:**

* **ActorID**: A unique identifier for each actor.
* **ActorName**: Name of the actor.
* **FilmID**: A unique identifier for each film.
* **FilmTitle**: The title of the film.
* **Role**: The character the actor played in the film.
* **ReleaseYear**: Year the film was released.
* **Rating**: IMDB or other rating platforms' rating of the film (could be float).
* **BoxOfficeRevenue**: The film’s box office earnings in millions.
* **Genre**: Genre(s) of the film (action, drama, comedy, etc.).

**Key Use Cases for This Dataset:**

1. **Track Actors’ Film Careers**:
   * Identify all films an actor has worked on over the years.
   * Calculate the actor’s total box office earnings across all films.
2. **Analyze Film Popularity**:
   * Identify which genres generate the most revenue or have the highest ratings.
   * Compare the success of films with the same actor or director.
3. **Career Trajectory of Actors**:
   * Analyze how an actor’s rating and choice of roles have evolved over time.
   * Determine if there’s a correlation between film genre and success.

**Sample Tasks to Practice:**

1. **Task 1: Create the Dataset**
   * Manually or programmatically generate 20 rows of data for the actor\_films dataset as shown above. Use random values for ReleaseYear, Rating, BoxOfficeRevenue, etc., for practice.
2. **Task 2: Query All Films of an Actor**
   * Write a query that lists all the films an actor has worked on, ordered by the release year.
3. **Task 3: Find the Highest Box Office Films**
   * Write a query to find the top 5 films based on box office revenue.
4. **Task 4: Find Most Frequent Collaborations**
   * Write a query to find actors who have starred together in more than one film.
5. **Task 5: Average Rating per Genre**
   * Calculate the average film rating for each genre.
6. **Task 6: Track Actor’s Career Rating Trend**
   * Write a query that tracks the average rating of films a specific actor has been in, year by year.
7. **Task 7: Identify Genre with Best Performance**
   * Write a query to find which film genre has the highest average box office revenue.
8. **Task 8: Top Grossing Actor**
   * Write a query to find the actor with the highest total box office revenue.

This dataset and the related tasks will help you practice SQL queries, analyze trends in data, and get a better understanding of actor-film relationships in the entertainment industry.

4o mini

**You said:**

tasks related to the 2 modules we discussed, tasks in very detail

**ChatGPT said:**

ChatGPT

**Detailed Tasks Related to Dimensional Modeling and Slowly Changing Dimensions (SCD Type 2)**

These tasks are designed to give you a hands-on understanding of **Dimensional Modeling** (Module 1) and **Slowly Changing Dimensions (SCD Type 2)** (Module 2), using an **actor-films** dataset as a practice case.

**Module 1: Dimensional Modeling Tasks**

1. **Task 1: Design a Star Schema for the Actor-Films Dataset**
   * **Objective**: Understand the structure of a star schema.
   * **Instructions**:
     + Identify the **fact table** and **dimension tables**.
     + **Fact Table**: fact\_actor\_performance
       - Columns: ActorID, FilmID, BoxOfficeRevenue, Rating.
     + **Dimension Tables**:
       - dim\_actor: Contains actor details (ActorID, ActorName, DateOfBirth).
       - dim\_film: Contains film details (FilmID, FilmTitle, ReleaseYear, Genre).
       - dim\_role: Contains role details (RoleID, RoleName).
     + Sketch the **star schema** and show the relationships between the fact and dimension tables.
     + Include the necessary primary and foreign key relationships.
2. **Task 2: Populate the Fact and Dimension Tables**
   * **Objective**: Load the dataset into the appropriate tables and structure them for query optimization.
   * **Instructions**:
     + Insert sample data into the **dimension tables** (dim\_actor, dim\_film, dim\_role).
     + Insert the **fact table** (fact\_actor\_performance) by linking the ActorID, FilmID, and RoleID from the dimension tables.
     + Example data:
       - dim\_actor: ActorID, ActorName, DateOfBirth.
       - dim\_film: FilmID, FilmTitle, ReleaseYear, Genre.
       - dim\_role: RoleID, RoleName.
3. **Task 3: Query the Fact Table to Analyze Actor Performance**
   * **Objective**: Use basic queries to understand how actors have performed in films.
   * **Instructions**:
     + Write a SQL query to find all **actors** who have starred in films with a **rating greater than 7**.
     + Write a SQL query to get the **average box office revenue** for all films by a specific actor.
     + Write a SQL query to find **actors** who have acted in **more than 5 films**.
4. **Task 4: Design Aggregation Queries**
   * **Objective**: Perform aggregations to analyze actor performance by genre, year, and rating.
   * **Instructions**:
     + Write a SQL query to find the **top 3 genres** based on the **average rating** of films.
     + Write a SQL query to calculate the **total box office revenue** by **actor**.
     + Write a SQL query to find the **most successful genre** based on **average box office revenue**.
5. **Task 5: Create Views and Optimized Queries**
   * **Objective**: Create views to simplify recurring queries and optimize for performance.
   * **Instructions**:
     + Create a **view** for querying the performance of each actor across multiple films.
     + Create a **view** to summarize each actor's earnings per year.
     + Optimize the views by adding appropriate **indexes** on the fact and dimension tables.

**Module 2: Slowly Changing Dimensions (SCD Type 2) Tasks**

1. **Task 1: Design SCD Type 2 for Actor Roles**
   * **Objective**: Implement Slowly Changing Dimensions (SCD) Type 2 for tracking changes in actor roles over time.
   * **Instructions**:
     + Extend the dim\_actor table to handle role changes over time:
       - Add columns: StartDate, EndDate, IsActive.
       - Example: If an actor’s role or film title changes, we store the old record with IsActive = 0 and the new record with IsActive = 1.
     + Create a script to track changes in the ActorName or RoleName for actors across different films.
2. **Task 2: Detect Changes and Update Actor Information**
   * **Objective**: Implement logic to detect and update changes in actor’s role or film titles.
   * **Instructions**:
     + Write a Spark or SQL script to detect **changes** in actor roles or film titles:
       - Compare incoming data with existing data in the dimension tables.
       - When a change is detected, **close out the previous record** (set IsActive = 0, update EndDate).
       - **Insert a new record** for the updated role/film (set IsActive = 1, StartDate = current\_date).
3. **Task 3: Handling Multiple Role Changes for the Same Actor**
   * **Objective**: Test SCD Type 2 behavior when an actor plays multiple roles over time.
   * **Instructions**:
     + Ensure that the dim\_actor table can handle **multiple changes** for the same actor.
     + For example, an actor might have played Role A in the first movie, and Role B in the next film.
     + Write queries to validate that multiple active records exist for each actor if they change roles over time.
4. **Task 4: Perform SCD Type 2 Logic for Historical Data**
   * **Objective**: Implement historical tracking for actors' roles in films.
   * **Instructions**:
     + Load a **new set of incoming data** (e.g., new movies or roles) and apply the SCD Type 2 logic.
     + Implement the following steps:
       - **Check if the actor exists** in dim\_actor.
       - **If yes**, check if the role has changed.
       - **If yes**, close the old record, insert a new one.
       - **If no**, retain the record as is.
     + Validate with a query to check if actors have historical records.
5. **Task 5: Implement Idempotent Pipeline for SCD Type 2**
   * **Objective**: Ensure that running the pipeline multiple times doesn’t result in duplicates.
   * **Instructions**:
     + Design a script or pipeline using **Apache Spark** or **SQL** to apply SCD Type 2 logic.
     + Implement **checks for idempotency**:
       - Ensure that the data load does not insert duplicate records.
       - Test by running the pipeline multiple times with the same input data and verify that the data remains consistent.
6. **Task 6: Query and Validate SCD Type 2 Updates**
   * **Objective**: Write SQL queries to validate that the changes are correctly tracked.
   * **Instructions**:
     + Write a query to get the **most recent record** of an actor from dim\_actor (i.e., IsActive = 1).
     + Write a query to get the **historical records** of an actor (i.e., all records where IsActive = 0 and check StartDate and EndDate).
     + Validate that the **EndDate** of the old record is set correctly and that the new record has the correct StartDate.

**Combined Tasks (Both Modules)**

1. **Task 1: Combine Dimensional Modeling and SCD Type 2 Logic**
   * **Objective**: Combine both dimensional modeling and SCD Type 2 logic into a complete solution.
   * **Instructions**:
     + Integrate the dimension tables (dim\_actor, dim\_film, dim\_role) with SCD Type 2 logic to track changes in **actor roles** and **film titles** over time.
     + Ensure that the fact table (fact\_actor\_performance) is also updated accordingly with the changes.
     + Automate the data transformation process and test for correct historical tracking and aggregation.
2. **Task 2: Full End-to-End Data Pipeline**
   * **Objective**: Build a complete data pipeline that performs extraction, transformation, and loading (ETL) with both dimensional modeling and SCD Type 2.
   * **Instructions**:
     + Use Apache Spark or SQL to perform the ETL process:
       - **Extract** data from the source dataset (actor\_films).
       - **Transform** the data to fit the dimensional model, implementing SCD Type 2 for actors.
       - **Load** the transformed data into the fact and dimension tables.
     + Test the pipeline on multiple runs and verify that no duplicate data is inserted.

These tasks will help you understand the implementation of **Dimensional Modeling** (Star Schema) and **Slowly Changing Dimensions (SCD Type 2)** with both theoretical and practical approaches. You'll work with an actor-films dataset, implementing these concepts using SQL, Apache Spark, or any other relevant ETL tool to solidify your understanding and ability to execute these techniques.

4o mini

**You said:**

you did not include STRUC and ARRAY data types and tasks and use cases including them...task like unmask, explode etc

**ChatGPT said:**

ChatGPT

Certainly! Let's build upon the **actor\_films dataset** with **STRUCT** and **ARRAY** data types, focusing on the **tasks, use cases, and transformations** such as **unmasking, exploding**, and handling complex data types.

**Dataset with STRUCT and ARRAY Data Types**

In the **actor\_films** dataset, let's assume that some attributes could have more complex data. For example, an actor could have multiple **awards**, and a film could have **multiple genres**. We'll store these attributes using **STRUCT** (for nested records) and **ARRAY** (for lists of values).

**Modified Dataset Structure:**

1. **Actor** (actor):
   * ActorID: Unique identifier for the actor.
   * ActorName: Actor's full name.
   * Awards: An **ARRAY** containing the awards the actor has won (e.g., ["Oscar", "Golden Globe"]).
   * BirthDetails: A **STRUCT** that holds actor's birth info (Country, BirthYear).
2. **Film** (film):
   * FilmID: Unique identifier for the film.
   * FilmTitle: Title of the film.
   * Genres: An **ARRAY** containing the genres of the film (e.g., ["Drama", "Action"]).
   * Director: Director’s name (e.g., "Steven Spielberg").
   * Actors: An **ARRAY** of ActorID values, representing the actors in the film (e.g., [1, 2, 3]).

**Sample Data:**

| **ActorID** | **ActorName** | **Awards** | **BirthDetails** |
| --- | --- | --- | --- |
| 1 | Robert Downey Jr. | ["Oscar", "Golden Globe"] | {"Country": "USA", "BirthYear": 1965} |
| 2 | Scarlett Johansson | ["BAFTA"] | {"Country": "USA", "BirthYear": 1984} |
| 3 | Tom Hanks | ["Oscar", "Golden Globe", "BAFTA"] | {"Country": "USA", "BirthYear": 1956} |

| **FilmID** | **FilmTitle** | **Genres** | **Director** | **Actors** |
| --- | --- | --- | --- | --- |
| 101 | Iron Man | ["Action", "Sci-Fi"] | "Jon Favreau" | [1, 2] |
| 102 | The Avengers | ["Action", "Sci-Fi"] | "Joss Whedon" | [1, 2, 3] |
| 103 | Lost in Translation | ["Drama", "Romance"] | "Sofia Coppola" | [2] |
| 104 | Forrest Gump | ["Drama"] | "Robert Zemeckis" | [3] |

**Tasks with STRUC and ARRAY Data Types:**

**Task 1: Query Nested STRUCT and ARRAY Columns**

* **Objective**: Retrieve information stored in STRUCT and ARRAY columns.
* **Instructions**:
  + **Access Nested Fields in STRUCT**: Write a query to retrieve the **country** and **birth year** of an actor.
  + Example SQL (in Spark SQL or BigQuery):

sql

Copy code

SELECT ActorName, BirthDetails.Country, BirthDetails.BirthYear

FROM actor\_films;

* + **Access Array Elements**: Write a query to get the **first award** of an actor from the Awards array.

sql

Copy code

SELECT ActorName, Awards[SAFE\_OFFSET(0)] AS FirstAward

FROM actor\_films;

**Task 2: Use the UNNEST or EXPLODE Function to Flatten Arrays**

* **Objective**: "Unnest" or "explode" complex arrays into multiple rows.
* **Instructions**:
  + **Explode the Array of Awards**: Use the UNNEST function in BigQuery or EXPLODE in Spark SQL to "flatten" the array of awards so that each award appears in a separate row.
    - Example SQL (BigQuery):

sql

Copy code

SELECT ActorName, Award

FROM actor\_films,

UNNEST(Awards) AS Award;

* + - Example SQL (Spark SQL):

sql

Copy code

SELECT ActorName, explode(Awards) AS Award

FROM actor\_films;

* + **Explode the Genres of Films**: Write a query to get the **film title** and **individual genres** by "exploding" the genres array.

sql

Copy code

SELECT FilmTitle, Genre

FROM film\_films,

UNNEST(Genres) AS Genre;

**Task 3: Use ARRAY\_AGG to Aggregate Array Data**

* **Objective**: Aggregate data into arrays to provide insights into data collected across rows.
* **Instructions**:
  + **Aggregate All Awards by Actor**: Write a query to get a **list of all actors** who won the "Oscar" and show the **awards list** for each actor.

sql

Copy code

SELECT ActorName, ARRAY\_AGG(Award) AS AllAwards

FROM actor\_films,

UNNEST(Awards) AS Award

WHERE Award = 'Oscar'

GROUP BY ActorName;

* + **Find All Actors in a Film**: Use ARRAY\_AGG to list all **actors** who appeared in a specific film (e.g., "The Avengers").

sql

Copy code

SELECT FilmTitle, ARRAY\_AGG(ActorName) AS Actors

FROM film\_films

JOIN actor\_films ON ARRAY\_CONTAINS(Actors, actor\_films.ActorID)

WHERE FilmTitle = 'The Avengers'

GROUP BY FilmTitle;

**Task 4: Query Arrays of Actors and Films**

* **Objective**: Use the ARRAY data type to manage lists of actors in films and films they have worked on.
* **Instructions**:
  + **Actors and Their Films**: Create a query to list all **films** that an **actor** has appeared in.
    - Example SQL:

sql

Copy code

SELECT ActorName, ARRAY\_AGG(FilmTitle) AS Films

FROM actor\_films

JOIN film\_films ON ARRAY\_CONTAINS(film\_films.Actors, actor\_films.ActorID)

GROUP BY ActorName;

**Task 5: Handling Nested Data Types with Complex Logic**

* **Objective**: Work with nested data types and apply logic for transformations.
* **Instructions**:
  + **Query Actor Details**: Retrieve all actors who were born in the **USA** and sort them by their **birth year**.
    - Example SQL:

sql

Copy code

SELECT ActorName, BirthDetails.Country, BirthDetails.BirthYear

FROM actor\_films

WHERE BirthDetails.Country = 'USA'

ORDER BY BirthDetails.BirthYear;

* + **Add Role Information to Film**: Add role details for each film by creating an array of roles for each film, assuming we have a column in the film table that can track actor roles.
    - Example (if "Roles" exists):

sql

Copy code

SELECT FilmTitle, ARRAY\_AGG(Role) AS Roles

FROM film\_films

JOIN actor\_films ON ARRAY\_CONTAINS(Actors, actor\_films.ActorID)

GROUP BY FilmTitle;

**Task 6: Mask Sensitive Data**

* **Objective**: Mask sensitive data using array structures and ensure data privacy.
* **Instructions**:
  + **Unmask Actors' Information**: If some of the actor data is masked (e.g., ActorName), create an **unmasking** operation to retrieve the original actor names.
    - **Example**: Suppose we have a masked ActorName column (e.g., replaced with Xxxxxx Xxxx), and a corresponding UnmaskedActorName column with the real name. Unmask and retrieve the real names for analysis:

sql

Copy code

SELECT ActorName, UnmaskedActorName

FROM actor\_films

WHERE ActorName = 'Xxxxxx Xxxx';

**Task 7: Use of STRUCT for Detailed Actor Info**

* **Objective**: Use STRUCT to store multi-level nested information about actors or films.
* **Instructions**:
  + **Actor Birth Details**: If BirthDetails is a STRUCT column (containing fields like Country, BirthYear, City), extract each of these fields separately in your query.
    - Example:

sql

Copy code

SELECT ActorName, BirthDetails.Country, BirthDetails.BirthYear, BirthDetails.City

FROM actor\_films;

**Summary of Key Concepts for STRUCT and ARRAY:**

* **STRUCT**: Used to group related data together in a single column (e.g., storing actor birth details as Country and BirthYear).
* **ARRAY**: Used to store a collection of elements (e.g., an actor's awards or a film's genres).
* Functions like **UNNEST()**, **EXPLODE()**, and **ARRAY\_AGG()** allow you to manipulate and query these complex data types, making it easier to handle multi-value and nested data.

These tasks help demonstrate practical implementations of **STRUCT** and **ARRAY** data types inHere are detailed tasks using **STRUCT** and **ARRAY** data types, focusing on transformations like **unmasking**, **exploding**, and handling complex data types within the **actor-films** dataset:

**Dataset with STRUCT and ARRAY Data Types**

In this dataset, the actor\_films table includes **ARRAY** and **STRUCT** columns to manage nested data and complex relationships.

**Example Structure:**

1. **Actor** (actor\_films):
   * ActorID: Unique identifier for the actor.
   * ActorName: Actor's full name.
   * Awards: An **ARRAY** of awards the actor has won (e.g., ["Oscar", "Golden Globe"]).
   * BirthDetails: A **STRUCT** containing Country and BirthYear (e.g., {"Country": "USA", "BirthYear": 1965}).
2. **Film** (film\_films):
   * FilmID: Unique identifier for the film.
   * FilmTitle: Title of the film.
   * Genres: An **ARRAY** of genres (e.g., ["Drama", "Action"]).
   * Director: Director’s name (e.g., "Steven Spielberg").
   * Actors: An **ARRAY** of ActorID values, representing the actors in the film (e.g., [1, 2, 3]).

**Sample Data:**

| **ActorID** | **ActorName** | **Awards** | **BirthDetails** |
| --- | --- | --- | --- |
| 1 | Robert Downey Jr. | ["Oscar", "Golden Globe"] | {"Country": "USA", "BirthYear": 1965} |
| 2 | Scarlett Johansson | ["BAFTA"] | {"Country": "USA", "BirthYear": 1984} |
| 3 | Tom Hanks | ["Oscar", "Golden Globe", "BAFTA"] | {"Country": "USA", "BirthYear": 1956} |

| **FilmID** | **FilmTitle** | **Genres** | **Director** | **Actors** |
| --- | --- | --- | --- | --- |
| 101 | Iron Man | ["Action", "Sci-Fi"] | "Jon Favreau" | [1, 2] |
| 102 | The Avengers | ["Action", "Sci-Fi"] | "Joss Whedon" | [1, 2, 3] |
| 103 | Lost in Translation | ["Drama", "Romance"] | "Sofia Coppola" | [2] |
| 104 | Forrest Gump | ["Drama"] | "Robert Zemeckis" | [3] |

**Task 1: Query Nested STRUCT and ARRAY Columns**

* **Objective**: Retrieve information stored in STRUCT and ARRAY columns.
* **Instructions**:
  + **Access Nested Fields in STRUCT**: Write a query to retrieve the **country** and **birth year** of an actor.

sql

Copy code

SELECT ActorName, BirthDetails.Country, BirthDetails.BirthYear

FROM actor\_films;

* + **Access Array Elements**: Write a query to get the **first award** of an actor from the Awards array.

sql

Copy code

SELECT ActorName, Awards[SAFE\_OFFSET(0)] AS FirstAward

FROM actor\_films;

**Task 2: Use the UNNEST or EXPLODE Function to Flatten Arrays**

* **Objective**: Flatten the **ARRAY** into individual rows.
* **Instructions**:
  + **Explode the Array of Awards**: Use UNNEST (BigQuery) or EXPLODE (Spark SQL) to "flatten" the array of awards.

sql

Copy code

SELECT ActorName, Award

FROM actor\_films,

UNNEST(Awards) AS Award;

* + **Explode the Genres of Films**: Write a query to get the **film title** and **individual genres** by "exploding" the genres array.

sql

Copy code

SELECT FilmTitle, Genre

FROM film\_films,

UNNEST(Genres) AS Genre;

**Task 3: Use ARRAY\_AGG to Aggregate Array Data**

* **Objective**: Aggregate data into arrays to provide insights into data collected across rows.
* **Instructions**:
  + **Aggregate All Awards by Actor**: Write a query to get a **list of all awards** won by an actor.

sql

Copy code

SELECT ActorName, ARRAY\_AGG(Award) AS AllAwards

FROM actor\_films,

UNNEST(Awards) AS Award

GROUP BY ActorName;

* + **Find All Actors in a Film**: Use ARRAY\_AGG to list all **actors** who appeared in a specific film (e.g., "The Avengers").

sql

Copy code

SELECT FilmTitle, ARRAY\_AGG(ActorName) AS Actors

FROM film\_films

JOIN actor\_films ON ARRAY\_CONTAINS(Actors, actor\_films.ActorID)

WHERE FilmTitle = 'The Avengers'

GROUP BY FilmTitle;

**Task 4: Handling Nested Data Types with Complex Logic**

* **Objective**: Work with nested data types and apply logic for transformations.
* **Instructions**:
  + **Query Actor Details**: Retrieve all actors who were born in the **USA** and sort them by their **birth year**.

sql

Copy code

SELECT ActorName, BirthDetails.Country, BirthDetails.BirthYear

FROM actor\_films

WHERE BirthDetails.Country = 'USA'

ORDER BY BirthDetails.BirthYear;

* + **Add Role Information to Film**: Assume the existence of a Roles column (a STRUCT) for storing actor roles in each film. Write a query to list actors and their roles.

sql

Copy code

SELECT FilmTitle, ActorName, Roles.RoleName

FROM film\_films

JOIN actor\_films ON ARRAY\_CONTAINS(Actors, actor\_films.ActorID)

JOIN UNNEST(Roles) AS Role;

**Task 5: Mask Sensitive Data**

* **Objective**: Mask sensitive data and ensure proper access controls for privacy.
* **Instructions**:
  + **Unmask Actor Names**: If some of the ActorName data is masked (e.g., Xxxxxx Xxxx), create a **unmasking** operation to retrieve the real names for analysis.

sql

Copy code

SELECT ActorName, UnmaskedActorName

FROM actor\_films

WHERE ActorName = 'Xxxxxx Xxxx';

**Task 6: Query Arrays of Actors and Films**

* **Objective**: Use the ARRAY data type to manage lists of actors in films and films they have worked on.
* **Instructions**:
  + **Actors and Their Films**: Create a query to list all **films** that an **actor** has appeared in.

sql

Copy code

SELECT ActorName, ARRAY\_AGG(FilmTitle) AS Films

FROM actor\_films

JOIN film\_films ON ARRAY\_CONTAINS(film\_films.Actors, actor\_films.ActorID)

GROUP BY ActorName;

**Task 7: Use of STRUCT for Detailed Actor Info**

* **Objective**: Use STRUCT to store and query detailed nested information about actors or films.
* **Instructions**:
  + **Actor Birth Details**: If BirthDetails is a STRUCT column (containing fields like Country, BirthYear, City), extract each of these fields separately in your query.

sql

Copy code

SELECT ActorName, BirthDetails.Country, BirthDetails.BirthYear, BirthDetails.City

FROM actor\_films;

**Summary of Key Concepts:**

* **STRUCT**: A complex data type used to group related attributes together (e.g., BirthDetails for storing Country and BirthYear).
* **ARRAY**: A collection of elements stored in a single column (e.g., an actor's list of awards or a film's list of genres).
* **UNNEST()** or **EXPLODE()**: These functions help flatten the **ARRAY** into individual rows, making it easier to perform further operations.
* **ARRAY\_AGG()**: Aggregates values back into an array from individual rows.
* **Masking and Unmasking**: Used for data privacy, where certain fields may be masked, and unmasking is performed for authorized users.

By practicing these tasks, you'll become more proficient in working with **complex data types** (like **STRUCT** and **ARRAY**) and understand how to implement these transformations using **SQL** or **Apache Spark** for large-scale data processing.

4o mini

**You said:**

ok we will start from first, create SQL queries for creating raw tables and queries for inserting 30 rows, we will do it step by step, STEP 1 : we will set the data

**ChatGPT said:**

ChatGPT

To start with, we'll create two raw tables based on our **actor\_films dataset**, which involves using **STRUCT** and **ARRAY** data types. Let's break this process into the following steps:

**Step 1: Creating the Raw Tables**

1. **actor\_films table**:
   * This table will store information about actors, including their name, awards, and birth details using **STRUCT** and **ARRAY** types.
2. **film\_films table**:
   * This table will store information about films, including the genres and actors involved, again using **ARRAY**.

**SQL Queries to Create Tables**

**1. Create actor\_films Table (Using STRUCT and ARRAY)**

The **actor\_films** table contains the following columns:

* ActorID: A unique identifier for the actor.
* ActorName: The actor's full name.
* Awards: An **ARRAY** of awards the actor has won (e.g., ["Oscar", "Golden Globe"]).
* BirthDetails: A **STRUCT** containing Country and BirthYear.

sql

Copy code

CREATE TABLE actor\_films (

ActorID INT PRIMARY KEY,

ActorName STRING,

Awards ARRAY<STRING>,

BirthDetails STRUCT<Country STRING, BirthYear INT>

);

* **ActorID** is an integer, uniquely identifying the actor.
* **ActorName** is a string representing the actor's name.
* **Awards** is an **ARRAY** of strings, storing multiple awards the actor has received.
* **BirthDetails** is a **STRUCT** that stores two fields: Country (STRING) and BirthYear (INT).

**2. Create film\_films Table (Using ARRAY)**

The **film\_films** table will have the following columns:

* FilmID: A unique identifier for the film.
* FilmTitle: The title of the film.
* Genres: An **ARRAY** containing the genres associated with the film.
* Director: The director's name.
* Actors: An **ARRAY** of ActorID values, representing the actors in the film.

sql

Copy code

CREATE TABLE film\_films (

FilmID INT PRIMARY KEY,

FilmTitle STRING,

Genres ARRAY<STRING>,

Director STRING,

Actors ARRAY<INT>

);

* **FilmID** is an integer, uniquely identifying the film.
* **FilmTitle** is a string representing the title of the film.
* **Genres** is an **ARRAY** of strings representing the genres of the film.
* **Director** is a string representing the director’s name.
* **Actors** is an **ARRAY** of integers, where each integer represents an ActorID who participated in the film.

**Step 2: Inserting Data into the Raw Tables**

**1. Insert Data into actor\_films Table**

Now that the table structure is set, we'll insert 30 rows of data for actors into the actor\_films table.

sql

Copy code

INSERT INTO actor\_films (ActorID, ActorName, Awards, BirthDetails)

VALUES

(1, 'Robert Downey Jr.', ARRAY['Oscar', 'Golden Globe'], STRUCT('USA', 1965)),

(2, 'Scarlett Johansson', ARRAY['BAFTA'], STRUCT('USA', 1984)),

(3, 'Tom Hanks', ARRAY['Oscar', 'Golden Globe', 'BAFTA'], STRUCT('USA', 1956)),

(4, 'Johnny Depp', ARRAY['Golden Globe', 'César'], STRUCT('USA', 1963)),

(5, 'Leonardo DiCaprio', ARRAY['Oscar', 'Golden Globe'], STRUCT('USA', 1974)),

(6, 'Meryl Streep', ARRAY['Oscar', 'Golden Globe', 'BAFTA'], STRUCT('USA', 1949)),

(7, 'Will Smith', ARRAY['Golden Globe'], STRUCT('USA', 1968)),

(8, 'Natalie Portman', ARRAY['Oscar'], STRUCT('Israel', 1981)),

(9, 'Matt Damon', ARRAY['Golden Globe'], STRUCT('USA', 1970)),

(10, 'Brad Pitt', ARRAY['Golden Globe'], STRUCT('USA', 1963)),

(11, 'Julia Roberts', ARRAY['Oscar', 'Golden Globe'], STRUCT('USA', 1967)),

(12, 'Denzel Washington', ARRAY['Oscar', 'Golden Globe'], STRUCT('USA', 1954)),

(13, 'Morgan Freeman', ARRAY['Oscar', 'Golden Globe'], STRUCT('USA', 1937)),

(14, 'Sandra Bullock', ARRAY['Oscar', 'Golden Globe'], STRUCT('USA', 1964)),

(15, 'Emma Stone', ARRAY['Oscar', 'Golden Globe'], STRUCT('USA', 1988)),

(16, 'Harrison Ford', ARRAY['Golden Globe'], STRUCT('USA', 1942)),

(17, 'Cate Blanchett', ARRAY['Oscar', 'Golden Globe'], STRUCT('Australia', 1969)),

(18, 'Christian Bale', ARRAY['Oscar', 'Golden Globe'], STRUCT('Wales', 1974)),

(19, 'Keanu Reeves', ARRAY['Golden Globe'], STRUCT('Canada', 1964)),

(20, 'Tom Cruise', ARRAY['Golden Globe'], STRUCT('USA', 1962)),

(21, 'Nicole Kidman', ARRAY['Oscar', 'Golden Globe'], STRUCT('Australia', 1967)),

(22, 'Johnny Depp', ARRAY['Golden Globe'], STRUCT('USA', 1963)),

(23, 'Hugh Jackman', ARRAY['Golden Globe'], STRUCT('Australia', 1968)),

(24, 'Reese Witherspoon', ARRAY['Oscar', 'Golden Globe'], STRUCT('USA', 1976)),

(25, 'Bradley Cooper', ARRAY['Oscar', 'Golden Globe'], STRUCT('USA', 1975)),

(26, 'Ryan Gosling', ARRAY['Golden Globe'], STRUCT('Canada', 1980)),

(27, 'Zoe Saldana', ARRAY['Golden Globe'], STRUCT('USA', 1978)),

(28, 'Chadwick Boseman', ARRAY['Golden Globe'], STRUCT('USA', 1976)),

(29, 'Idris Elba', ARRAY['Golden Globe'], STRUCT('UK', 1972)),

(30, 'Chris Hemsworth', ARRAY['Golden Globe'], STRUCT('Australia', 1983));

**2. Insert Data into film\_films Table**

Next, we insert data into the film\_films table, which stores information about films.

sql

Copy code

INSERT INTO film\_films (FilmID, FilmTitle, Genres, Director, Actors)

VALUES

(101, 'Iron Man', ARRAY['Action', 'Sci-Fi'], 'Jon Favreau', ARRAY[1, 2]),

(102, 'The Avengers', ARRAY['Action', 'Sci-Fi'], 'Joss Whedon', ARRAY[1, 2, 3]),

(103, 'Lost in Translation', ARRAY['Drama', 'Romance'], 'Sofia Coppola', ARRAY[2]),

(104, 'Forrest Gump', ARRAY['Drama'], 'Robert Zemeckis', ARRAY[3]),

(105, 'The Social Network', ARRAY['Drama', 'Biography'], 'David Fincher', ARRAY[9]),

(106, 'Pulp Fiction', ARRAY['Crime', 'Drama'], 'Quentin Tarantino', ARRAY[4, 10]),

(107, 'Inception', ARRAY['Action', 'Sci-Fi'], 'Christopher Nolan', ARRAY[1, 9]),

(108, 'Titanic', ARRAY['Drama', 'Romance'], 'James Cameron', ARRAY[5]),

(109, 'The Martian', ARRAY['Sci-Fi', 'Drama'], 'Ridley Scott', ARRAY[9]),

(110, 'The Revenant', ARRAY['Drama', 'Adventure'], 'Alejandro González Iñárritu', ARRAY[5, 17]),

(111, 'The Wolf of Wall Street', ARRAY['Drama', 'Biography'], 'Martin Scorsese', ARRAY[5, 12]),

(112, 'Gravity', ARRAY['Drama', 'Sci-Fi'], 'Alfonso Cuarón', ARRAY[8]),

(113, 'Avatar', ARRAY['Action', 'Sci-Fi'], 'James Cameron', ARRAY[27]),

(114, 'The Hunger Games', ARRAY['Action', 'Sci-Fi'], 'Gary Ross', ARRAY[26]),

(115, 'The Dark Knight', ARRAY['Action', 'Crime'], 'Christopher Nolan', ARRAY[1, 12]),

(116, 'Les Misérables', ARRAY['Drama', 'Musical'], 'Tom Hooper', ARRAY[20]),

(117, 'Gladiator', ARRAY['Action', 'Drama'], 'Ridley Scott', ARRAY[16]),

(118, 'Mission: Impossible', ARRAY['Action', 'Thriller'], 'Brian De Palma', ARRAY[20]),

(119, 'Mad Max: Fury Road', ARRAY['Action', 'Adventure'], 'George Miller', ARRAY[23]),

(120, 'The Theory of Everything', ARRAY['Drama', 'Biography'], 'James Marsh', ARRAY[21]),

(121, 'Guardians of the Galaxy', ARRAY['Action', 'Sci-Fi'], 'James Gunn', ARRAY[27]),

(122, 'Wonder Woman', ARRAY['Action', 'Adventure'], 'Patty Jenkins', ARRAY[26]),

(123, 'The Matrix', ARRAY['Action', 'Sci-Fi'], 'Lana Wachowski', ARRAY[19]),

(124, 'Deadpool', ARRAY['Action', 'Comedy'], 'Tim Miller', ARRAY[25]),

(125, 'A Star is Born', ARRAY['Drama', 'Music'], 'Bradley Cooper', ARRAY[25]),

(126,

4o mini

**You said:**

not in array or struct, raw data should be free flow data

**ChatGPT said:**

ChatGPT

To create raw tables with free-flow data (not using **ARRAY** or **STRUCT**) and insert values into them, we will proceed step-by-step. Here's how we can structure our tables and data for the **actor\_films** and **film\_films** datasets without relying on complex data types like **ARRAY** or **STRUCT**.

**Step 1: Create Raw Tables for Actor and Film Data**

We'll create two raw tables:

1. **actor\_films table**: This will store actor information, including their name, awards, and birth details.
2. **film\_films table**: This will store film information, including film title, genres, director, and the actors involved in each film.

**1. Create actor\_films Table**

Here, we will create a simple table with free-flow data.

sql

Copy code

CREATE TABLE actor\_films (

ActorID INT PRIMARY KEY,

ActorName STRING,

Awards STRING, -- Stores a comma-separated list of awards

BirthCountry STRING, -- Birth country

BirthYear INT -- Birth year

);

* **ActorID** is the unique identifier for each actor.
* **ActorName** is the name of the actor.
* **Awards** is a string field that contains a list of awards, separated by commas (instead of using an **ARRAY**).
* **BirthCountry** and **BirthYear** are simple fields representing the country and the year the actor was born.

**2. Create film\_films Table**

The **film\_films** table will store details about films in a free-flow format.

sql

Copy code

CREATE TABLE film\_films (

FilmID INT PRIMARY KEY,

FilmTitle STRING, -- Title of the film

Genres STRING, -- Stores a comma-separated list of genres

Director STRING, -- Director of the film

ActorIDs STRING -- Comma-separated list of ActorIDs who appeared in the film

);

* **FilmID** is the unique identifier for each film.
* **FilmTitle** is the title of the film.
* **Genres** stores the genres in a comma-separated string.
* **Director** is the name of the director of the film.
* **ActorIDs** is a comma-separated list of **ActorID** values representing the actors who were involved in the film.

**Step 2: Insert Data into the Raw Tables**

Next, we'll insert 30 rows of data into each table. For simplicity, the **Awards** and **Genres** will be stored as comma-separated strings, and the **ActorIDs** for the films will also be stored in a similar format.

**1. Insert Data into actor\_films Table**

sql

Copy code

INSERT INTO actor\_films (ActorID, ActorName, Awards, BirthCountry, BirthYear)

VALUES

(1, 'Robert Downey Jr.', 'Oscar,Golden Globe', 'USA', 1965),

(2, 'Scarlett Johansson', 'BAFTA', 'USA', 1984),

(3, 'Tom Hanks', 'Oscar,Golden Globe,BAFTA', 'USA', 1956),

(4, 'Johnny Depp', 'Golden Globe,César', 'USA', 1963),

(5, 'Leonardo DiCaprio', 'Oscar,Golden Globe', 'USA', 1974),

(6, 'Meryl Streep', 'Oscar,Golden Globe,BAFTA', 'USA', 1949),

(7, 'Will Smith', 'Golden Globe', 'USA', 1968),

(8, 'Natalie Portman', 'Oscar', 'Israel', 1981),

(9, 'Matt Damon', 'Golden Globe', 'USA', 1970),

(10, 'Brad Pitt', 'Golden Globe', 'USA', 1963),

(11, 'Julia Roberts', 'Oscar,Golden Globe', 'USA', 1967),

(12, 'Denzel Washington', 'Oscar,Golden Globe', 'USA', 1954),

(13, 'Morgan Freeman', 'Oscar,Golden Globe', 'USA', 1937),

(14, 'Sandra Bullock', 'Oscar,Golden Globe', 'USA', 1964),

(15, 'Emma Stone', 'Oscar,Golden Globe', 'USA', 1988),

(16, 'Harrison Ford', 'Golden Globe', 'USA', 1942),

(17, 'Cate Blanchett', 'Oscar,Golden Globe', 'Australia', 1969),

(18, 'Christian Bale', 'Oscar,Golden Globe', 'Wales', 1974),

(19, 'Keanu Reeves', 'Golden Globe', 'Canada', 1964),

(20, 'Tom Cruise', 'Golden Globe', 'USA', 1962),

(21, 'Nicole Kidman', 'Oscar,Golden Globe', 'Australia', 1967),

(22, 'Johnny Depp', 'Golden Globe', 'USA', 1963),

(23, 'Hugh Jackman', 'Golden Globe', 'Australia', 1968),

(24, 'Reese Witherspoon', 'Oscar,Golden Globe', 'USA', 1976),

(25, 'Bradley Cooper', 'Oscar,Golden Globe', 'USA', 1975),

(26, 'Ryan Gosling', 'Golden Globe', 'Canada', 1980),

(27, 'Zoe Saldana', 'Golden Globe', 'USA', 1978),

(28, 'Chadwick Boseman', 'Golden Globe', 'USA', 1976),

(29, 'Idris Elba', 'Golden Globe', 'UK', 1972),

(30, 'Chris Hemsworth', 'Golden Globe', 'Australia', 1983);

**2. Insert Data into film\_films Table**

sql

Copy code

INSERT INTO film\_films (FilmID, FilmTitle, Genres, Director, ActorIDs)

VALUES

(101, 'Iron Man', 'Action,Sci-Fi', 'Jon Favreau', '1,2'),

(102, 'The Avengers', 'Action,Sci-Fi', 'Joss Whedon', '1,2,3'),

(103, 'Lost in Translation', 'Drama,Romance', 'Sofia Coppola', '2'),

(104, 'Forrest Gump', 'Drama', 'Robert Zemeckis', '3'),

(105, 'The Social Network', 'Drama,Biography', 'David Fincher', '9'),

(106, 'Pulp Fiction', 'Crime,Drama', 'Quentin Tarantino', '4,10'),

(107, 'Inception', 'Action,Sci-Fi', 'Christopher Nolan', '1,9'),

(108, 'Titanic', 'Drama,Romance', 'James Cameron', '5'),

(109, 'The Martian', 'Sci-Fi,Drama', 'Ridley Scott', '9'),

(110, 'The Revenant', 'Drama,Adventure', 'Alejandro González Iñárritu', '5,17'),

(111, 'The Wolf of Wall Street', 'Drama,Biography', 'Martin Scorsese', '5,12'),

(112, 'Gravity', 'Drama,Sci-Fi', 'Alfonso Cuarón', '8'),

(113, 'Avatar', 'Action,Sci-Fi', 'James Cameron', '27'),

(114, 'The Hunger Games', 'Action,Sci-Fi', 'Gary Ross', '26'),

(115, 'The Dark Knight', 'Action,Crime', 'Christopher Nolan', '1,12'),

(116, 'Les Misérables', 'Drama,Musical', 'Tom Hooper', '20'),

(117, 'Gladiator', 'Action,Drama', 'Ridley Scott', '16'),

(118, 'Mission: Impossible', 'Action,Thriller', 'Brian De Palma', '20'),

(119, 'Mad Max: Fury Road', 'Action,Adventure', 'George Miller', '23'),

(120, 'The Theory of Everything', 'Drama,Biography', 'James Marsh', '21'),

(121, 'Guardians of the Galaxy', 'Action,Sci-Fi', 'James Gunn', '27'),

(122, 'Wonder Woman', 'Action,Adventure', 'Patty Jenkins', '26'),

(123, 'The Matrix', 'Action,Sci-Fi', 'Lana Wachowski', '19'),

(124, 'Deadpool', 'Action,Comedy', 'Tim Miller', '25'),

(125, 'A Star is Born', 'Drama,Music', 'Bradley Cooper', '25');

**Conclusion**

* We have created two raw tables: actor\_films and film\_films, where actor data and film data are stored in a free-flow format (using comma-separated strings for awards, genres, and actors).
* We've inserted 30 rows of data into both tables.

**Next Steps**

* In the next steps, we can focus on writing SQL queries to manipulate and extract useful insights from this raw data.
* You can practice querying this data, focusing on topics like joining tables, filtering, and aggregating data.

4o mini

**You said:**

i dont see rating reviews ect columns

**ChatGPT said:**

ChatGPT

Got it! If you'd like to include additional columns like **rating**, **reviews**, etc., we can modify the dataset to reflect more detailed information. Below are the updated SQL table creation scripts and corresponding insert statements, which now include these additional columns.

**Updated Table Design for actor\_films and film\_films**

**1. Updated actor\_films Table**

In this table, we will include additional columns such as **Awards**, **BirthCountry**, **BirthYear**, and now **Rating** (average actor rating across films) and **Reviews** (number of reviews the actor has received).

sql

Copy code

CREATE TABLE actor\_films (

ActorID INT PRIMARY KEY,

ActorName STRING,

Awards STRING, -- Stores a comma-separated list of awards

BirthCountry STRING, -- Birth country

BirthYear INT, -- Birth year

Rating DECIMAL(3, 2), -- Average rating of actor's films

Reviews INT -- Number of reviews received for the actor's films

);

* **Rating**: A **DECIMAL** type field that represents the average rating across all the films the actor has appeared in. It ranges from 0.00 to 10.00 (as an example).
* **Reviews**: The **INT** field will store the number of reviews the actor’s films have accumulated (or reviews that specifically mention the actor).

**2. Updated film\_films Table**

We will also update the film\_films table to include **FilmRating**, **ReviewCount**, and **BoxOffice** columns.

sql

Copy code

CREATE TABLE film\_films (

FilmID INT PRIMARY KEY,

FilmTitle STRING, -- Title of the film

Genres STRING, -- Stores a comma-separated list of genres

Director STRING, -- Director of the film

ActorIDs STRING, -- Comma-separated list of ActorIDs who appeared in the film

FilmRating DECIMAL(3, 2), -- Average rating for the film

ReviewCount INT, -- Number of reviews the film has received

BoxOffice BIGINT -- Total box office revenue for the film

);

* **FilmRating**: A **DECIMAL** field that represents the average rating of the film.
* **ReviewCount**: An **INT** field that stores the number of reviews the film has received.
* **BoxOffice**: A **BIGINT** field for storing the total box office earnings for the film in USD.

**Step 2: Insert Data into the Updated Tables**

Now, we'll insert 30 rows of sample data into both tables, including the new columns for ratings, reviews, and box office earnings.

**Insert Data into actor\_films Table**

sql

Copy code

INSERT INTO actor\_films (ActorID, ActorName, Awards, BirthCountry, BirthYear, Rating, Reviews)

VALUES

(1, 'Robert Downey Jr.', 'Oscar,Golden Globe', 'USA', 1965, 8.7, 1200),

(2, 'Scarlett Johansson', 'BAFTA', 'USA', 1984, 8.3, 1100),

(3, 'Tom Hanks', 'Oscar,Golden Globe,BAFTA', 'USA', 1956, 9.2, 1500),

(4, 'Johnny Depp', 'Golden Globe,César', 'USA', 1963, 7.8, 900),

(5, 'Leonardo DiCaprio', 'Oscar,Golden Globe', 'USA', 1974, 9.0, 1800),

(6, 'Meryl Streep', 'Oscar,Golden Globe,BAFTA', 'USA', 1949, 8.9, 1400),

(7, 'Will Smith', 'Golden Globe', 'USA', 1968, 8.0, 950),

(8, 'Natalie Portman', 'Oscar', 'Israel', 1981, 8.4, 800),

(9, 'Matt Damon', 'Golden Globe', 'USA', 1970, 8.2, 1000),

(10, 'Brad Pitt', 'Golden Globe', 'USA', 1963, 8.6, 1100),

(11, 'Julia Roberts', 'Oscar,Golden Globe', 'USA', 1967, 8.5, 1300),

(12, 'Denzel Washington', 'Oscar,Golden Globe', 'USA', 1954, 8.7, 1050),

(13, 'Morgan Freeman', 'Oscar,Golden Globe', 'USA', 1937, 8.3, 850),

(14, 'Sandra Bullock', 'Oscar,Golden Globe', 'USA', 1964, 8.1, 1100),

(15, 'Emma Stone', 'Oscar,Golden Globe', 'USA', 1988, 8.6, 1200),

(16, 'Harrison Ford', 'Golden Globe', 'USA', 1942, 7.9, 1150),

(17, 'Cate Blanchett', 'Oscar,Golden Globe', 'Australia', 1969, 8.4, 900),

(18, 'Christian Bale', 'Oscar,Golden Globe', 'Wales', 1974, 8.8, 1300),

(19, 'Keanu Reeves', 'Golden Globe', 'Canada', 1964, 7.7, 750),

(20, 'Tom Cruise', 'Golden Globe', 'USA', 1962, 8.3, 1400),

(21, 'Nicole Kidman', 'Oscar,Golden Globe', 'Australia', 1967, 8.5, 950),

(22, 'Johnny Depp', 'Golden Globe', 'USA', 1963, 7.8, 900),

(23, 'Hugh Jackman', 'Golden Globe', 'Australia', 1968, 8.2, 850),

(24, 'Reese Witherspoon', 'Oscar,Golden Globe', 'USA', 1976, 8.0, 1000),

(25, 'Bradley Cooper', 'Oscar,Golden Globe', 'USA', 1975, 8.4, 950),

(26, 'Ryan Gosling', 'Golden Globe', 'Canada', 1980, 7.9, 1150),

(27, 'Zoe Saldana', 'Golden Globe', 'USA', 1978, 8.1, 800),

(28, 'Chadwick Boseman', 'Golden Globe', 'USA', 1976, 8.5, 650),

(29, 'Idris Elba', 'Golden Globe', 'UK', 1972, 8.3, 700),

(30, 'Chris Hemsworth', 'Golden Globe', 'Australia', 1983, 8.0, 950);

**Insert Data into film\_films Table**

sql

Copy code

INSERT INTO film\_films (FilmID, FilmTitle, Genres, Director, ActorIDs, FilmRating, ReviewCount, BoxOffice)

VALUES

(101, 'Iron Man', 'Action,Sci-Fi', 'Jon Favreau', '1,2', 8.5, 2500, 585000000),

(102, 'The Avengers', 'Action,Sci-Fi', 'Joss Whedon', '1,2,3', 8.8, 5000, 1500000000),

(103, 'Lost in Translation', 'Drama,Romance', 'Sofia Coppola', '2', 8.1, 1200, 100000000),

(104, 'Forrest Gump', 'Drama', 'Robert Zemeckis', '3', 9.2, 1500, 678000000),

(105, 'The Social Network', 'Drama,Biography', 'David Fincher', '9', 8.0, 1000, 225000000),

(106, 'Pulp Fiction', 'Crime,Drama', 'Quentin Tarantino', '4,10', 8.9, 3000, 213000000),

(107, 'Inception', 'Action,Sci-Fi', 'Christopher Nolan', '1,9', 8.8, 4500, 829000000),

(108, 'Titanic', 'Drama,Romance', 'James Cameron', '5', 8.7, 4300, 2200000000),

(109, 'The Martian', 'Sci-Fi,Drama', 'Ridley Scott', '9', 8.4, 3500, 630000000),

(110, 'The Revenant', 'Drama,Adventure', 'Alejandro González Iñárritu', '5,17', 8.0, 2000, 533000000),

(111, 'The Wolf of Wall Street', 'Drama,Biography', 'Martin Scorsese', '5,12', 8.2, 2200, 392000000),

(112, 'Gravity', 'Drama,Sci-Fi', 'Alfonso Cuarón', '8', 7.9, 1500, 723000000),

(113, 'Avatar', 'Action,Sci-Fi', 'James Cameron', '27', 8.0, 3700, 2780000000),

(114, 'The Hunger Games', 'Action,Sci-Fi', 'Gary Ross', '26', 8.0, 2800, 694000000),

(

4o mini

**You said:**

does actor film rating and film rating different? what value does it add?

**ChatGPT said:**

ChatGPT

Yes, **actor film rating** and **film rating** are different, and they serve different purposes. Here's how they differ and what value each one provides:

**1. Film Rating:**

* **Definition**: The **film rating** is an average score that reflects how viewers rate the film as a whole, based on their opinions about the story, direction, acting, cinematography, etc.
* **Value Added**:
  + The **film rating** gives an overall view of how well the film was received by the audience.
  + This rating typically comes from movie review platforms like IMDb, Rotten Tomatoes, or Metacritic, based on a broad range of reviewers and viewers.
  + It helps a potential viewer assess whether the movie is worth watching based on collective feedback.

**Example**: A movie with a rating of **8.5** on IMDb indicates that viewers generally think highly of the film.

**2. Actor Film Rating:**

* **Definition**: The **actor film rating** refers to an average score or rating based on the performance of a specific actor in the films they have appeared in.
* **Value Added**:
  + The **actor film rating** provides insight into how well an actor is perceived for their roles, based on their performances across multiple films.
  + This is especially useful when analyzing an actor's career and impact on the film industry. A high actor film rating suggests that the actor's performances are consistently praised by critics and audiences.
  + It can also help in understanding an actor's versatility or range, as the rating might differ based on the genres or types of characters they play.

**Example**: If **Robert Downey Jr.** has an **actor rating** of **8.7**, it means that, on average, his performances in his movies are highly rated by critics and viewers, which may be due to his portrayal of iconic characters like Iron Man in the **Marvel Cinematic Universe**.

**How They Differ:**

* **Focus**:
  + **Film rating** measures how the overall film (including its direction, script, cinematography, and performances) is perceived.
  + **Actor film rating** measures how well the actor's performance is perceived, based on their portrayal of a character in a film.
* **Purpose**:
  + The **film rating** is used by viewers to decide if they want to watch a film.
  + The **actor film rating** is useful for analyzing an actor's career, understanding their strengths in specific roles, or assessing their impact on a film's success.

**Example:**

* If you look at a film like *The Avengers*:
  + The **film rating** would represent the general reception of the movie's plot, visual effects, direction, pacing, etc.
  + The **actor film rating** for an actor like **Robert Downey Jr.** would focus only on his portrayal of **Iron Man/Tony Stark** in *The Avengers* and how that performance contributed to the film's overall success.

**When to Use Actor vs Film Rating:**

* **Film rating** is more general and reflects the success or failure of the entire movie.
* **Actor film rating** is useful when you want to understand how individual actors contribute to a film’s success or how they are perceived across multiple films.

**Why Both Are Important:**

* **Film ratings** help potential viewers gauge the quality of a film as a whole, while **actor ratings** offer insight into the actor's contribution to a film's appeal.
* When analyzing the effectiveness of a film or the popularity of an actor, both ratings can offer valuable perspectives.

**In Summary:**

* **Film rating** is an evaluation of the entire movie.
* **Actor film rating** is an evaluation of an actor's performance in a given movie or across their career.

These ratings are complementary and, when combined, offer a more detailed picture of a film’s success and an actor's career impact.

4o mini

**You said:**

all the tables should be postgres tables

**ChatGPT said:**

ChatGPT

Sure! Let's create **PostgreSQL** tables for the **actor\_films** and **film\_films** datasets, along with the relevant data types for the columns and some **SQL queries** for inserting data. We will focus on **film ratings**, **actor ratings**, and additional fields like **Reviews**, **Awards**, etc., as mentioned earlier.

Below are the SQL commands to create the tables in PostgreSQL and insert sample data into them:

**1. Creating actor\_films Table**

In this table, we will store details about the actors, including their **awards**, **birth country**, **birth year**, **actor ratings**, and **review count**.

sql

Copy code

CREATE TABLE actor\_films (

ActorID SERIAL PRIMARY KEY, -- Auto-incrementing unique ID for each actor

ActorName VARCHAR(100), -- Name of the actor

Awards VARCHAR(255), -- Comma-separated list of awards the actor has won

BirthCountry VARCHAR(100), -- Country where the actor was born

BirthYear INT, -- Year the actor was born

Rating DECIMAL(3,2), -- Actor's average rating based on their performances in various films

Reviews INT -- Number of reviews for the actor's films

);

* **ActorID**: SERIAL will automatically generate a unique integer for each actor.
* **ActorName**: VARCHAR(100) stores the actor's name.
* **Awards**: VARCHAR(255) stores a comma-separated list of the actor's awards.
* **BirthCountry**: VARCHAR(100) stores the country of birth.
* **BirthYear**: INT stores the year of birth.
* **Rating**: DECIMAL(3,2) represents the average rating for the actor's movies (out of 10, with two decimal places).
* **Reviews**: INT stores the number of reviews the actor has received.

**2. Creating film\_films Table**

In this table, we will store details about the films, including **genres**, **directors**, **actor IDs** (to link to the actor\_films table), **film ratings**, **review counts**, and **box office earnings**.

sql

Copy code

CREATE TABLE film\_films (

FilmID SERIAL PRIMARY KEY, -- Auto-incrementing unique ID for each film

FilmTitle VARCHAR(255), -- Title of the film

Genres VARCHAR(255), -- Comma-separated list of genres (e.g., Drama, Sci-Fi)

Director VARCHAR(100), -- Director of the film

ActorIDs VARCHAR(255), -- Comma-separated list of ActorIDs who appeared in the film

FilmRating DECIMAL(3,2), -- Average rating of the film (out of 10)

ReviewCount INT, -- Total number of reviews for the film

BoxOffice BIGINT -- Total box office earnings (in USD)

);

* **FilmID**: SERIAL auto-increments to provide a unique ID for each film.
* **FilmTitle**: VARCHAR(255) stores the title of the film.
* **Genres**: VARCHAR(255) stores a comma-separated list of genres.
* **Director**: VARCHAR(100) stores the director's name.
* **ActorIDs**: VARCHAR(255) stores a comma-separated list of Actor IDs (to associate actors with the film).
* **FilmRating**: DECIMAL(3,2) stores the average rating of the film (out of 10).
* **ReviewCount**: INT stores the total number of reviews for the film.
* **BoxOffice**: BIGINT stores the total box office earnings (in USD).

**3. Inserting Sample Data into actor\_films Table**

Here is some sample data for the **actor\_films** table, including actors' names, awards, birth country, birth year, ratings, and reviews:

sql

Copy code

INSERT INTO actor\_films (ActorName, Awards, BirthCountry, BirthYear, Rating, Reviews)

VALUES

('Robert Downey Jr.', 'Oscar,Golden Globe', 'USA', 1965, 8.7, 1200),

('Scarlett Johansson', 'BAFTA', 'USA', 1984, 8.3, 1100),

('Tom Hanks', 'Oscar,Golden Globe,BAFTA', 'USA', 1956, 9.2, 1500),

('Johnny Depp', 'Golden Globe,César', 'USA', 1963, 7.8, 900),

('Leonardo DiCaprio', 'Oscar,Golden Globe', 'USA', 1974, 9.0, 1800),

('Meryl Streep', 'Oscar,Golden Globe,BAFTA', 'USA', 1949, 8.9, 1400),

('Will Smith', 'Golden Globe', 'USA', 1968, 8.0, 950),

('Natalie Portman', 'Oscar', 'Israel', 1981, 8.4, 800),

('Matt Damon', 'Golden Globe', 'USA', 1970, 8.2, 1000),

('Brad Pitt', 'Golden Globe', 'USA', 1963, 8.6, 1100);

**4. Inserting Sample Data into film\_films Table**

Here is some sample data for the **film\_films** table, including film titles, genres, directors, actor IDs (referencing the actor\_films table), film ratings, review counts, and box office earnings:

sql

Copy code

INSERT INTO film\_films (FilmTitle, Genres, Director, ActorIDs, FilmRating, ReviewCount, BoxOffice)

VALUES

('Iron Man', 'Action,Sci-Fi', 'Jon Favreau', '1,2', 8.5, 2500, 585000000),

('The Avengers', 'Action,Sci-Fi', 'Joss Whedon', '1,2,3', 8.8, 5000, 1500000000),

('Lost in Translation', 'Drama,Romance', 'Sofia Coppola', '2', 8.1, 1200, 100000000),

('Forrest Gump', 'Drama', 'Robert Zemeckis', '3', 9.2, 1500, 678000000),

('The Social Network', 'Drama,Biography', 'David Fincher', '9', 8.0, 1000, 225000000),

('Pulp Fiction', 'Crime,Drama', 'Quentin Tarantino', '4,10', 8.9, 3000, 213000000),

('Inception', 'Action,Sci-Fi', 'Christopher Nolan', '1,9', 8.8, 4500, 829000000),

('Titanic', 'Drama,Romance', 'James Cameron', '5', 8.7, 4300, 2200000000),

('The Martian', 'Sci-Fi,Drama', 'Ridley Scott', '9', 8.4, 3500, 630000000),

('The Revenant', 'Drama,Adventure', 'Alejandro González Iñárritu', '5,17', 8.0, 2000, 533000000);

**5. Explanation of How to Link the Two Tables**

* The **actor\_films** table stores individual actors' information, including their **Rating** (average rating across films) and **Reviews** (total number of reviews their films received).
* The **film\_films** table stores information about films, including the **ActorIDs** column, which links to the **ActorID** in the **actor\_films** table. This is how you can associate an actor with a specific film.

To summarize:

* **Film ratings** reflect how the film was received overall.
* **Actor film ratings** reflect how the actor's performance is perceived across multiple films.

For example, if you want to calculate the average **Actor Rating** for an actor like **Robert Downey Jr.**, you would query the actor\_films table for that actor’s rating. To find the **Film Rating** for a movie like *Iron Man*, you would query the film\_films table.

**Next Steps:**

Once you have set up the tables and inserted data, you can perform various SQL queries like:

1. **Join** the actor\_films and film\_films tables based on the ActorIDs to get actor details and their associated films.
2. **Aggregate** data, such as calculating the average rating for an actor across all their films.
3. **Filter** films based on their rating or box office earnings.
4. Use **Window Functions** to rank actors based on their average ratings or films based on box office revenue.

Would you like to proceed with any of these queries? Let me know if you'd like to continue or modify the structure!

4o mini

**You said:**

now lets focus on our major tasks, provide detail tasks for task 1 in very detail

**ChatGPT said:**

ChatGPT

Certainly! Let's break down **Task 1** in very detail, focused on the **actor\_films** and **film\_films** tables that we created earlier, and build a comprehensive set of tasks to help you fully implement the concepts of **Dimensional Data Modeling** and **Slowly Changing Dimensions** (SCD).

**Task 1: Create and Populate Tables, Perform Initial Queries**

This task will guide you step-by-step on how to create the tables, populate them, and write queries to manipulate the data based on **actor ratings**, **film ratings**, and other aspects like **review counts** and **box office earnings**.

**Step 1: Table Creation**

You have already defined the schema for both **actor\_films** and **film\_films** tables, but let’s revisit them to make sure we understand their relationship and purpose.

**Task 1.1: Create actor\_films and film\_films Tables**

1. **Create the actor\_films Table**:
   * This table stores data about each actor including personal information, awards, ratings, and reviews.

sql

Copy code

CREATE TABLE actor\_films (

ActorID SERIAL PRIMARY KEY,

ActorName VARCHAR(100),

Awards VARCHAR(255),

BirthCountry VARCHAR(100),

BirthYear INT,

Rating DECIMAL(3,2),

Reviews INT

);

1. **Create the film\_films Table**:
   * This table stores data about films including film title, genres, director, ratings, review counts, and box office earnings.

sql

Copy code

CREATE TABLE film\_films (

FilmID SERIAL PRIMARY KEY,

FilmTitle VARCHAR(255),

Genres VARCHAR(255),

Director VARCHAR(100),

ActorIDs VARCHAR(255), -- Comma-separated list of ActorIDs

FilmRating DECIMAL(3,2),

ReviewCount INT,

BoxOffice BIGINT

);

**Step 2: Insert Sample Data**

Now that the tables are created, it’s time to insert the data. Here, you'll simulate adding sample records to these tables. We’ve already created a set of sample data, but you can adjust and expand this as needed.

**Task 1.2: Insert Data into actor\_films Table**

sql

Copy code

INSERT INTO actor\_films (ActorName, Awards, BirthCountry, BirthYear, Rating, Reviews)

VALUES

('Robert Downey Jr.', 'Oscar,Golden Globe', 'USA', 1965, 8.7, 1200),

('Scarlett Johansson', 'BAFTA', 'USA', 1984, 8.3, 1100),

('Tom Hanks', 'Oscar,Golden Globe,BAFTA', 'USA', 1956, 9.2, 1500),

('Johnny Depp', 'Golden Globe,César', 'USA', 1963, 7.8, 900),

('Leonardo DiCaprio', 'Oscar,Golden Globe', 'USA', 1974, 9.0, 1800),

('Meryl Streep', 'Oscar,Golden Globe,BAFTA', 'USA', 1949, 8.9, 1400),

('Will Smith', 'Golden Globe', 'USA', 1968, 8.0, 950),

('Natalie Portman', 'Oscar', 'Israel', 1981, 8.4, 800),

('Matt Damon', 'Golden Globe', 'USA', 1970, 8.2, 1000),

('Brad Pitt', 'Golden Globe', 'USA', 1963, 8.6, 1100);

**Task 1.3: Insert Data into film\_films Table**

sql

Copy code

INSERT INTO film\_films (FilmTitle, Genres, Director, ActorIDs, FilmRating, ReviewCount, BoxOffice)

VALUES

('Iron Man', 'Action,Sci-Fi', 'Jon Favreau', '1,2', 8.5, 2500, 585000000),

('The Avengers', 'Action,Sci-Fi', 'Joss Whedon', '1,2,3', 8.8, 5000, 1500000000),

('Lost in Translation', 'Drama,Romance', 'Sofia Coppola', '2', 8.1, 1200, 100000000),

('Forrest Gump', 'Drama', 'Robert Zemeckis', '3', 9.2, 1500, 678000000),

('The Social Network', 'Drama,Biography', 'David Fincher', '9', 8.0, 1000, 225000000),

('Pulp Fiction', 'Crime,Drama', 'Quentin Tarantino', '4,10', 8.9, 3000, 213000000),

('Inception', 'Action,Sci-Fi', 'Christopher Nolan', '1,9', 8.8, 4500, 829000000),

('Titanic', 'Drama,Romance', 'James Cameron', '5', 8.7, 4300, 2200000000),

('The Martian', 'Sci-Fi,Drama', 'Ridley Scott', '9', 8.4, 3500, 630000000),

('The Revenant', 'Drama,Adventure', 'Alejandro González Iñárritu', '5,17', 8.0, 2000, 533000000);

**Step 3: Data Manipulation Queries**

Once the tables are created and populated, the next step is to practice querying and manipulating the data to analyze it. The following queries will help you apply the concepts of **Dimensional Data Modeling** and **Slowly Changing Dimensions (SCD)**.

**Task 1.4: Query to Find Actor Ratings for Films**

This query helps us calculate the average **actor rating** for films they have appeared in.

sql

Copy code

SELECT a.ActorName,

AVG(f.FilmRating) AS AvgActorFilmRating

FROM actor\_films a

JOIN film\_films f ON POSITION(CAST(a.ActorID AS TEXT) IN f.ActorIDs) > 0

GROUP BY a.ActorName;

* **Explanation**: We use a **JOIN** to link actors to films based on the **ActorIDs** field in the film\_films table. Then, we calculate the **average film rating** for each actor by aggregating the ratings of the films they’ve appeared in.

**Task 1.5: Query to Find Top Films Based on Box Office Earnings**

This query helps you find the top 5 films based on **box office earnings**.

sql

Copy code

SELECT FilmTitle,

BoxOffice

FROM film\_films

ORDER BY BoxOffice DESC

LIMIT 5;

* **Explanation**: This query selects the **FilmTitle** and **BoxOffice** earnings, ordering the films in descending order of earnings and limiting the output to the top 5 films.

**Task 1.6: Query to Find Films with High Ratings and Low Review Count**

This query can be used to find films that have a high **FilmRating** but relatively **low review counts**.

sql

Copy code

SELECT FilmTitle, FilmRating, ReviewCount

FROM film\_films

WHERE FilmRating > 8.0

AND ReviewCount < 1000;

* **Explanation**: The query filters films based on a **FilmRating** above **8.0** but with **less than 1000 reviews**. This might indicate films that are critically well-received but have not been widely reviewed.

**Step 4: Link Actors to Their Films**

Here, we will query both tables to find out which **actors** appeared in specific **films** by leveraging the **ActorIDs** in the film\_films table.

**Task 1.7: Query to Find All Films an Actor Appeared In**

For example, to find all films **Robert Downey Jr.** (ActorID = 1) appeared in:

sql

Copy code

SELECT f.FilmTitle, f.FilmRating

FROM film\_films f

JOIN actor\_films a ON POSITION(CAST(a.ActorID AS TEXT) IN f.ActorIDs) > 0

WHERE a.ActorName = 'Robert Downey Jr.';

* **Explanation**: The **POSITION()** function checks if the **ActorID** exists in the **ActorIDs** field of the film\_films table. It returns the films that Robert Downey Jr. appeared in along with the ratings.

**Step 5: Use Case Scenarios with Slowly Changing Dimensions (SCD)**

In this step, we introduce the concept of **Slowly Changing Dimensions (SCD)**, where we manage changes in actor details over time (e.g., if an actor's rating changes).

**Task 1.8: Simulate SCD for Actors**

In this task, we'll assume that actor ratings may change over time (e.g., based on newer films or critical reevaluations). The following example query shows how to **insert or update** actor ratings in the **actor\_films** table.

sql

Copy code

-- Insert a new record for an updated actor rating

INSERT INTO actor\_films (ActorName, Awards, BirthCountry, BirthYear, Rating, Reviews)

VALUES ('Robert Downey Jr.', 'Oscar,Golden Globe', 'USA', 1965, 8.9, 1300)

ON CONFLICT (ActorID) DO UPDATE

SET Rating = 8.9, Reviews = 1300;

* **Explanation**: This SQLHere’s **Task 1** broken down step by step, focusing on creating and populating the **actor\_films** and **film\_films** tables in PostgreSQL, and performing basic data manipulations. These tasks will help you understand the core concepts of Dimensional Data Modeling and how to manage the relationship between the data effectively.

**Task 1: Create and Populate Tables, Perform Initial Queries**

**Step 1: Table Creation**

**Task 1.1: Create actor\_films Table**

* This table stores detailed information about each actor, including awards, birth country, ratings, and reviews.

sql

Copy code

CREATE TABLE actor\_films (

ActorID SERIAL PRIMARY KEY, -- Auto-incrementing unique ID for each actor

ActorName VARCHAR(100), -- Actor's name

Awards VARCHAR(255), -- Awards won by the actor (e.g., Oscar, BAFTA)

BirthCountry VARCHAR(100), -- Country where the actor was born

BirthYear INT, -- Actor's birth year

Rating DECIMAL(3,2), -- Average rating based on actor's film performances

Reviews INT -- Number of reviews the actor has received

);

* **ActorID**: A unique identifier for each actor. SERIAL ensures it auto-increments.
* **ActorName**: Name of the actor.
* **Awards**: A list of awards the actor has won (can be stored as a comma-separated string).
* **BirthCountry**: The country of birth for the actor.
* **BirthYear**: Year when the actor was born.
* **Rating**: Average rating for the actor (based on all films they have acted in).
* **Reviews**: Number of reviews the actor's films have received.

**Task 1.2: Create film\_films Table**

* This table stores details about the films, including the title, genres, ratings, and box office earnings.

sql

Copy code

CREATE TABLE film\_films (

FilmID SERIAL PRIMARY KEY, -- Auto-incrementing unique ID for each film

FilmTitle VARCHAR(255), -- Film title

Genres VARCHAR(255), -- Genres of the film (e.g., Drama, Action)

Director VARCHAR(100), -- Film director

ActorIDs VARCHAR(255), -- Comma-separated list of ActorIDs

FilmRating DECIMAL(3,2), -- Average rating of the film

ReviewCount INT, -- Total number of reviews for the film

BoxOffice BIGINT -- Total box office earnings (in USD)

);

* **FilmID**: A unique identifier for each film. SERIAL ensures it auto-increments.
* **FilmTitle**: The title of the film.
* **Genres**: A comma-separated list of genres (e.g., Action, Drama).
* **Director**: The name of the film’s director.
* **ActorIDs**: A comma-separated list of ActorIDs, linking the actors to the film.
* **FilmRating**: The average rating for the film.
* **ReviewCount**: The number of reviews the film has received.
* **BoxOffice**: Total box office earnings in USD.

**Step 2: Insert Sample Data**

Now that the tables are created, let’s populate them with sample data.

**Task 1.3: Insert Data into actor\_films Table**

sql

Copy code

INSERT INTO actor\_films (ActorName, Awards, BirthCountry, BirthYear, Rating, Reviews)

VALUES

('Robert Downey Jr.', 'Oscar,Golden Globe', 'USA', 1965, 8.7, 1200),

('Scarlett Johansson', 'BAFTA', 'USA', 1984, 8.3, 1100),

('Tom Hanks', 'Oscar,Golden Globe,BAFTA', 'USA', 1956, 9.2, 1500),

('Johnny Depp', 'Golden Globe,César', 'USA', 1963, 7.8, 900),

('Leonardo DiCaprio', 'Oscar,Golden Globe', 'USA', 1974, 9.0, 1800),

('Meryl Streep', 'Oscar,Golden Globe,BAFTA', 'USA', 1949, 8.9, 1400),

('Will Smith', 'Golden Globe', 'USA', 1968, 8.0, 950),

('Natalie Portman', 'Oscar', 'Israel', 1981, 8.4, 800),

('Matt Damon', 'Golden Globe', 'USA', 1970, 8.2, 1000),

('Brad Pitt', 'Golden Globe', 'USA', 1963, 8.6, 1100);

* This query inserts data for 10 actors into the **actor\_films** table.

**Task 1.4: Insert Data into film\_films Table**

sql

Copy code

INSERT INTO film\_films (FilmTitle, Genres, Director, ActorIDs, FilmRating, ReviewCount, BoxOffice)

VALUES

('Iron Man', 'Action,Sci-Fi', 'Jon Favreau', '1,2', 8.5, 2500, 585000000),

('The Avengers', 'Action,Sci-Fi', 'Joss Whedon', '1,2,3', 8.8, 5000, 1500000000),

('Lost in Translation', 'Drama,Romance', 'Sofia Coppola', '2', 8.1, 1200, 100000000),

('Forrest Gump', 'Drama', 'Robert Zemeckis', '3', 9.2, 1500, 678000000),

('The Social Network', 'Drama,Biography', 'David Fincher', '9', 8.0, 1000, 225000000),

('Pulp Fiction', 'Crime,Drama', 'Quentin Tarantino', '4,10', 8.9, 3000, 213000000),

('Inception', 'Action,Sci-Fi', 'Christopher Nolan', '1,9', 8.8, 4500, 829000000),

('Titanic', 'Drama,Romance', 'James Cameron', '5', 8.7, 4300, 2200000000),

('The Martian', 'Sci-Fi,Drama', 'Ridley Scott', '9', 8.4, 3500, 630000000),

('The Revenant', 'Drama,Adventure', 'Alejandro González Iñárritu', '5,17', 8.0, 2000, 533000000);

* This query inserts data for 10 films into the **film\_films** table. Notice that actor IDs are provided as comma-separated lists in the **ActorIDs** column, linking actors to specific films.

**Step 3: Querying Data**

**Task 1.5: Query to Find Actor Ratings for Films**

This query calculates the **average film rating** for each actor based on their films.

sql

Copy code

SELECT a.ActorName,

AVG(f.FilmRating) AS AvgActorFilmRating

FROM actor\_films a

JOIN film\_films f ON POSITION(CAST(a.ActorID AS TEXT) IN f.ActorIDs) > 0

GROUP BY a.ActorName;

* **Explanation**:
  + **JOIN**: Links the actor\_films and film\_films tables using the **ActorIDs** field.
  + **POSITION**: Checks if the actor’s **ActorID** is present in the **ActorIDs** list in the film\_films table.
  + **AVG**: Calculates the average film rating for each actor based on their films.

**Task 1.6: Query to Find Top Films Based on Box Office Earnings**

This query finds the **top 5 films** based on **box office earnings**.

sql

Copy code

SELECT FilmTitle,

BoxOffice

FROM film\_films

ORDER BY BoxOffice DESC

LIMIT 5;

* **Explanation**:
  + **ORDER BY**: Sorts the films in descending order of box office earnings.
  + **LIMIT 5**: Limits the result to the top 5 films.

**Task 1.7: Query to Find Films with High Ratings and Low Review Counts**

This query finds films with a **high rating** but **low review counts**.

sql

Copy code

SELECT FilmTitle, FilmRating, ReviewCount

FROM film\_films

WHERE FilmRating > 8.0

AND ReviewCount < 1000;

* **Explanation**: Filters films with **ratings above 8.0** and **review counts under 1000**.

**Step 4: Linking Actors to Their Films**

**Task 1.8: Query to Find All Films an Actor Appeared In**

This query lists all the films an actor (e.g., **Robert Downey Jr.**) appeared in, based on their **ActorID**.

sql

Copy code

SELECT f.FilmTitle, f.FilmRating

FROM film\_films f

JOIN actor\_films a ON POSITION(CAST(a.ActorID AS TEXT) IN f.ActorIDs) > 0

WHERE a.ActorName = 'Robert Downey Jr.';

* **Explanation**:
  + Uses **JOIN** to link actors to films based on the **ActorIDs**.
  + Filters for films that **Robert Downey Jr.** appeared in.

**Step 5: Handling Slowly Changing Dimensions (SCD)**

####### Task 1: Detailed Tasks for Step 1 (Create Tables and Insert Data)

Here are detailed steps to carry out Task 1 for **creating raw tables and inserting data**, considering your **actor\_films** and **film\_films** tables in PostgreSQL.

**Step 1: Table Creation**

In this step, we will create two tables: **actor\_films** and **film\_films**. These tables will hold key information about the actors and the films they have acted in.

**Task 1.1: Create the actor\_films Table**

This table stores information about actors, including their ratings, reviews, awards, and birth details. Here is the SQL query to create the table:

sql

Copy code

CREATE TABLE actor\_films (

ActorID SERIAL PRIMARY KEY, -- Auto-incrementing unique ID for each actor

ActorName VARCHAR(100), -- Actor's name

Awards VARCHAR(255), -- Awards won by the actor (e.g., Oscar, BAFTA)

BirthCountry VARCHAR(100), -- Country where the actor was born

BirthYear INT, -- Actor's birth year

Rating DECIMAL(3,2), -- Average rating based on actor's film performances

Reviews INT -- Number of reviews the actor has received

);

**Task 1.2: Create the film\_films Table**

This table holds information about the films, including the title, genres, ratings, and the actors who appeared in them. The **ActorIDs** are stored as a comma-separated list to link the actors with films.

sql

Copy code

CREATE TABLE film\_films (

FilmID SERIAL PRIMARY KEY, -- Auto-incrementing unique ID for each film

FilmTitle VARCHAR(255), -- Film title

Genres VARCHAR(255), -- Genres of the film (e.g., Drama, Action)

Director VARCHAR(100), -- Film director

ActorIDs VARCHAR(255), -- Comma-separated list of ActorIDs

FilmRating DECIMAL(3,2), -- Average rating of the film

ReviewCount INT, -- Total number of reviews for the film

BoxOffice BIGINT -- Total box office earnings (in USD)

);

**Step 2: Insert Sample Data**

Once the tables are created, you need to insert data into them. We will insert 10 sample records for each table, as outlined earlier.

**Task 1.3: Insert Data into actor\_films Table**

The following SQL query inserts 10 records into the **actor\_films** table:

sql

Copy code

INSERT INTO actor\_films (ActorName, Awards, BirthCountry, BirthYear, Rating, Reviews)

VALUES

('Robert Downey Jr.', 'Oscar,Golden Globe', 'USA', 1965, 8.7, 1200),

('Scarlett Johansson', 'BAFTA', 'USA', 1984, 8.3, 1100),

('Tom Hanks', 'Oscar,Golden Globe,BAFTA', 'USA', 1956, 9.2, 1500),

('Johnny Depp', 'Golden Globe,César', 'USA', 1963, 7.8, 900),

('Leonardo DiCaprio', 'Oscar,Golden Globe', 'USA', 1974, 9.0, 1800),

('Meryl Streep', 'Oscar,Golden Globe,BAFTA', 'USA', 1949, 8.9, 1400),

('Will Smith', 'Golden Globe', 'USA', 1968, 8.0, 950),

('Natalie Portman', 'Oscar', 'Israel', 1981, 8.4, 800),

('Matt Damon', 'Golden Globe', 'USA', 1970, 8.2, 1000),

('Brad Pitt', 'Golden Globe', 'USA', 1963, 8.6, 1100);

* This query inserts information about the **actor**'s name, **awards**, **birth country**, **birth year**, **rating**, and **number of reviews**.

**Task 1.4: Insert Data into film\_films Table**

This query inserts 10 sample records into the **film\_films** table:

sql

Copy code

INSERT INTO film\_films (FilmTitle, Genres, Director, ActorIDs, FilmRating, ReviewCount, BoxOffice)

VALUES

('Iron Man', 'Action,Sci-Fi', 'Jon Favreau', '1,2', 8.5, 2500, 585000000),

('The Avengers', 'Action,Sci-Fi', 'Joss Whedon', '1,2,3', 8.8, 5000, 1500000000),

('Lost in Translation', 'Drama,Romance', 'Sofia Coppola', '2', 8.1, 1200, 100000000),

('Forrest Gump', 'Drama', 'Robert Zemeckis', '3', 9.2, 1500, 678000000),

('The Social Network', 'Drama,Biography', 'David Fincher', '9', 8.0, 1000, 225000000),

('Pulp Fiction', 'Crime,Drama', 'Quentin Tarantino', '4,10', 8.9, 3000, 213000000),

('Inception', 'Action,Sci-Fi', 'Christopher Nolan', '1,9', 8.8, 4500, 829000000),

('Titanic', 'Drama,Romance', 'James Cameron', '5', 8.7, 4300, 2200000000),

('The Martian', 'Sci-Fi,Drama', 'Ridley Scott', '9', 8.4, 3500, 630000000),

('The Revenant', 'Drama,Adventure', 'Alejandro González Iñárritu', '5,17', 8.0, 2000, 533000000);

* The **ActorIDs** are stored as a **comma-separated list** for each film, linking it to the actors who appeared in it. The columns include **FilmTitle**, **Genres**, **Director**, **FilmRating**, **ReviewCount**, and **BoxOffice**.

**Step 3: Data Manipulation Queries**

After inserting the data, you’ll write queries to manipulate and analyze the data.

**Task 1.5: Query to Find Actor Ratings for Films**

This query calculates the **average film rating** for each actor based on their films.

sql

Copy code

SELECT a.ActorName,

AVG(f.FilmRating) AS AvgActorFilmRating

FROM actor\_films a

JOIN film\_films f ON POSITION(CAST(a.ActorID AS TEXT) IN f.ActorIDs) > 0

GROUP BY a.ActorName;

* **Explanation**:
  + This query uses **JOIN** to link actors to films via the **ActorIDs** in the film\_films table.
  + The query calculates the average rating for each actor by taking the **FilmRating** of the films they have acted in.

**Task 1.6: Query to Find Top Films Based on Box Office Earnings**

This query finds the top 5 films based on **box office earnings**.

sql

Copy code

SELECT FilmTitle,

BoxOffice

FROM film\_films

ORDER BY BoxOffice DESC

LIMIT 5;

* **Explanation**:
  + This query orders the films by **BoxOffice** earnings in descending order and limits the result to the top 5 films.

**Task 1.7: Query to Find Films with High Ratings and Low Review Counts**

This query finds films that have a **high rating** but **relatively low review counts**.

sql

Copy code

SELECT FilmTitle, FilmRating, ReviewCount

FROM film\_films

WHERE FilmRating > 8.0

AND ReviewCount < 1000;

* **Explanation**: Filters films with **ratings above 8.0** and **review counts below 1000**. This can help identify films that may have good ratings but are under-reviewed.

**Step 4: Link Actors to Their Films**

In this step, we will query both tables to find the films that specific actors have appeared in.

**Task 1.8: Query to Find All Films an Actor Appeared In**

This query lists all the films an actor (e.g., **Robert Downey Jr.**) appeared in.

sql

Copy code

SELECT f.FilmTitle, f.FilmRating

FROM film\_films f

JOIN actor\_films a ON POSITION(CAST(a.ActorID AS TEXT) IN f.ActorIDs) > 0

WHERE a.ActorName = 'Robert Downey Jr.';

* **Explanation**:
  + The **JOIN** operation links actors to films based on the **ActorIDs** field.
  + This query filters for all films that **Robert Downey Jr.** (ActorID = 1) appeared in, and it shows the **FilmTitle** and **FilmRating**.

**Step 5: Handling Slowly Changing Dimensions (SCD)**

**Task 1.9: Simulate Slowly Changing Dimension (SCD) for Actor Ratings**

To handle situations where an actor’s rating might change over time (e.g., with newer films), we’ll use an **INSERT** statement with the ON CONFLICT clause.

sql

Copy code

-- Insert a new record for an updated actor rating

INSERT INTO actor\_films (ActorName, Awards, BirthCountry, BirthYear, Rating, Reviews)

VALUES ('Robert DowHere is a detailed breakdown of \*\*Task 1\*\* for creating and populating the `actor\_films` and `film\_films` tables in PostgreSQL, along with data manipulations and sample queries:

### \*\*Step 1: Table Creation\*\*

This step focuses on creating two primary tables, \*\*actor\_films\*\* and \*\*film\_films\*\*, which store essential information about actors and films.

#### \*\*Task 1.1: Create the `actor\_films` Table\*\*

The `actor\_films` table stores details about actors, including their names, awards, birth country, ratings, and reviews.

```sql

CREATE TABLE actor\_films (

ActorID SERIAL PRIMARY KEY, -- Auto-incrementing unique ID for each actor

ActorName VARCHAR(100), -- Actor's name

Awards VARCHAR(255), -- Awards won by the actor (e.g., Oscar, BAFTA)

BirthCountry VARCHAR(100), -- Country where the actor was born

BirthYear INT, -- Actor's birth year

Rating DECIMAL(3,2), -- Average rating based on actor's film performances

Reviews INT -- Number of reviews the actor has received

);

* **ActorID**: Auto-incremented unique identifier for each actor.
* **ActorName**: Name of the actor.
* **Awards**: Awards won by the actor (e.g., Oscar, Golden Globe).
* **BirthCountry**: Country of birth.
* **BirthYear**: Birth year of the actor.
* **Rating**: Average rating of the actor's performances in films.
* **Reviews**: Total number of reviews the actor's films have received.

**Task 1.2: Create the film\_films Table**

The film\_films table holds information about films, including their title, genres, director, linked actors (via ActorIDs), film ratings, and box office earnings.

sql

Copy code

CREATE TABLE film\_films (

FilmID SERIAL PRIMARY KEY, -- Auto-incrementing unique ID for each film

FilmTitle VARCHAR(255), -- Film title

Genres VARCHAR(255), -- Genres of the film (e.g., Drama, Action)

Director VARCHAR(100), -- Film director

ActorIDs VARCHAR(255), -- Comma-separated list of ActorIDs

FilmRating DECIMAL(3,2), -- Average rating of the film

ReviewCount INT, -- Total number of reviews for the film

BoxOffice BIGINT -- Total box office earnings (in USD)

);

* **FilmID**: Auto-incremented unique identifier for each film.
* **FilmTitle**: Title of the film.
* **Genres**: Genres of the film (e.g., Action, Drama).
* **Director**: The director of the film.
* **ActorIDs**: Comma-separated list of ActorIDs who appeared in the film.
* **FilmRating**: Average film rating.
* **ReviewCount**: Number of reviews the film has received.
* **BoxOffice**: Total box office earnings in USD.

**Step 2: Insert Sample Data**

**Task 1.3: Insert Data into actor\_films Table**

Here’s how to insert data for 10 actors into the actor\_films table:

sql

Copy code

INSERT INTO actor\_films (ActorName, Awards, BirthCountry, BirthYear, Rating, Reviews)

VALUES

('Robert Downey Jr.', 'Oscar,Golden Globe', 'USA', 1965, 8.7, 1200),

('Scarlett Johansson', 'BAFTA', 'USA', 1984, 8.3, 1100),

('Tom Hanks', 'Oscar,Golden Globe,BAFTA', 'USA', 1956, 9.2, 1500),

('Johnny Depp', 'Golden Globe,César', 'USA', 1963, 7.8, 900),

('Leonardo DiCaprio', 'Oscar,Golden Globe', 'USA', 1974, 9.0, 1800),

('Meryl Streep', 'Oscar,Golden Globe,BAFTA', 'USA', 1949, 8.9, 1400),

('Will Smith', 'Golden Globe', 'USA', 1968, 8.0, 950),

('Natalie Portman', 'Oscar', 'Israel', 1981, 8.4, 800),

('Matt Damon', 'Golden Globe', 'USA', 1970, 8.2, 1000),

('Brad Pitt', 'Golden Globe', 'USA', 1963, 8.6, 1100);

**Task 1.4: Insert Data into film\_films Table**

Insert data for 10 films into the film\_films table. ActorIDs are stored as comma-separated lists:

sql

Copy code

INSERT INTO film\_films (FilmTitle, Genres, Director, ActorIDs, FilmRating, ReviewCount, BoxOffice)

VALUES

('Iron Man', 'Action,Sci-Fi', 'Jon Favreau', '1,2', 8.5, 2500, 585000000),

('The Avengers', 'Action,Sci-Fi', 'Joss Whedon', '1,2,3', 8.8, 5000, 1500000000),

('Lost in Translation', 'Drama,Romance', 'Sofia Coppola', '2', 8.1, 1200, 100000000),

('Forrest Gump', 'Drama', 'Robert Zemeckis', '3', 9.2, 1500, 678000000),

('The Social Network', 'Drama,Biography', 'David Fincher', '9', 8.0, 1000, 225000000),

('Pulp Fiction', 'Crime,Drama', 'Quentin Tarantino', '4,10', 8.9, 3000, 213000000),

('Inception', 'Action,Sci-Fi', 'Christopher Nolan', '1,9', 8.8, 4500, 829000000),

('Titanic', 'Drama,Romance', 'James Cameron', '5', 8.7, 4300, 2200000000),

('The Martian', 'Sci-Fi,Drama', 'Ridley Scott', '9', 8.4, 3500, 630000000),

('The Revenant', 'Drama,Adventure', 'Alejandro González Iñárritu', '5,17', 8.0, 2000, 533000000);

**Step 3: Querying Data**

**Task 1.5: Query to Find Actor Ratings for Films**

To find the average film ratings for actors based on the films they have acted in:

sql

Copy code

SELECT a.ActorName,

AVG(f.FilmRating) AS AvgActorFilmRating

FROM actor\_films a

JOIN film\_films f ON POSITION(CAST(a.ActorID AS TEXT) IN f.ActorIDs) > 0

GROUP BY a.ActorName;

* **Explanation**: This query joins actor\_films and film\_films by matching the **ActorID** to **ActorIDs** in film\_films. It calculates the average film rating for each actor based on the films they have appeared in.

**Task 1.6: Query to Find Top Films Based on Box Office Earnings**

To find the top 5 films based on box office earnings:

sql

Copy code

SELECT FilmTitle,

BoxOffice

FROM film\_films

ORDER BY BoxOffice DESC

LIMIT 5;

* **Explanation**: Orders films by **BoxOffice** in descending order and limits the output to the top 5 films.

**Task 1.7: Query to Find Films with High Ratings and Low Review Counts**

To find films with ratings above 8.0 but with fewer than 1000 reviews:

sql

Copy code

SELECT FilmTitle, FilmRating, ReviewCount

FROM film\_films

WHERE FilmRating > 8.0

AND ReviewCount < 1000;

* **Explanation**: Filters films based on their **rating** (above 8.0) and **review count** (below 1000).

**Step 4: Link Actors to Their Films**

**Task 1.8: Query to Find All Films an Actor Appeared In**

For example, to find all films **Robert Downey Jr.** appeared in:

sql

Copy code

SELECT f.FilmTitle, f.FilmRating

FROM film\_films f

JOIN actor\_films a ON POSITION(CAST(a.ActorID AS TEXT) IN f.ActorIDs) > 0

WHERE a.ActorName = 'Robert Downey Jr.';

* **Explanation**: This query lists all films that **Robert Downey Jr.** has appeared in by matching the **ActorID** from the actor\_films table with the **ActorIDs** in the film\_films table.

**Step 5: Handling Slowly Changing Dimensions (SCD)**

When an actor’s rating or other details change over time, you can handle this using the ON CONFLICT clause for **upsert** operations, ensuring that historical data remains intact and changes are tracked.

Example:

sql

Copy code

-- Update or insert a new actor record if the rating changes

INSERT INTO actor\_films (ActorName, Awards, BirthCountry, BirthYear, Rating, Reviews)

VALUES ('Robert Downey Jr.', 'Oscar,Golden Globe', 'USA', 1965, 8.8, 1250)

ON CONFLICT (### Detailed Task Breakdown for Task 1 (Creating Tables and Inserting Data)

---

### \*\*Step 1: Table Creation\*\*

We start by creating the two tables: \*\*actor\_films\*\* and \*\*film\_films\*\*. These tables will store information about actors and films, with a focus on key attributes like ratings, reviews, awards, and box office earnings.

#### \*\*Task 1.1: Create the `actor\_films` Table\*\*

This table will hold actor information such as their name, awards, country of birth, ratings, and the number of reviews they've received. Below is the SQL query to create the `actor\_films` table:

```sql

CREATE TABLE actor\_films (

ActorID SERIAL PRIMARY KEY, -- Auto-incrementing unique ID for each actor

ActorName VARCHAR(100), -- Actor's name

Awards VARCHAR(255), -- Awards won by the actor (e.g., Oscar, BAFTA)

BirthCountry VARCHAR(100), -- Country where the actor was born

BirthYear INT, -- Actor's birth year

Rating DECIMAL(3,2), -- Average rating based on actor's film performances

Reviews INT -- Number of reviews the actor has received

);

* **ActorID** is an auto-incrementing ID, ensuring each actor gets a unique identifier.
* **ActorName**, **Awards**, **BirthCountry**, **BirthYear**, **Rating**, and **Reviews** are key data columns for actor-specific information.

**Task 1.2: Create the film\_films Table**

This table contains information about the films, including film title, genre, director, actors, ratings, and box office earnings. Below is the SQL query to create the film\_films table:

sql

Copy code

CREATE TABLE film\_films (

FilmID SERIAL PRIMARY KEY, -- Auto-incrementing unique ID for each film

FilmTitle VARCHAR(255), -- Film title

Genres VARCHAR(255), -- Genres of the film (e.g., Drama, Action)

Director VARCHAR(100), -- Film director

ActorIDs VARCHAR(255), -- Comma-separated list of ActorIDs

FilmRating DECIMAL(3,2), -- Average rating of the film

ReviewCount INT, -- Total number of reviews for the film

BoxOffice BIGINT -- Total box office earnings (in USD)

);

* **FilmID** is a unique identifier for each film, auto-incremented.
* **ActorIDs** stores a list of ActorIDs (comma-separated) to link actors to films.
* Other columns store key data about the film, such as **Genres**, **Director**, **FilmRating**, **ReviewCount**, and **BoxOffice**.

**Step 2: Insert Sample Data**

**Task 1.3: Insert Data into the actor\_films Table**

We will now insert sample data for 10 actors into the actor\_films table. This includes their name, awards, country of birth, birth year, ratings, and reviews.

sql

Copy code

INSERT INTO actor\_films (ActorName, Awards, BirthCountry, BirthYear, Rating, Reviews)

VALUES

('Robert Downey Jr.', 'Oscar,Golden Globe', 'USA', 1965, 8.7, 1200),

('Scarlett Johansson', 'BAFTA', 'USA', 1984, 8.3, 1100),

('Tom Hanks', 'Oscar,Golden Globe,BAFTA', 'USA', 1956, 9.2, 1500),

('Johnny Depp', 'Golden Globe,César', 'USA', 1963, 7.8, 900),

('Leonardo DiCaprio', 'Oscar,Golden Globe', 'USA', 1974, 9.0, 1800),

('Meryl Streep', 'Oscar,Golden Globe,BAFTA', 'USA', 1949, 8.9, 1400),

('Will Smith', 'Golden Globe', 'USA', 1968, 8.0, 950),

('Natalie Portman', 'Oscar', 'Israel', 1981, 8.4, 800),

('Matt Damon', 'Golden Globe', 'USA', 1970, 8.2, 1000),

('Brad Pitt', 'Golden Globe', 'USA', 1963, 8.6, 1100);

This query populates the **actor\_films** table with 10 actors and their respective data. The **Rating** and **Reviews** fields are crucial for later analysis.

**Task 1.4: Insert Data into the film\_films Table**

Next, we insert sample data for 10 films into the film\_films table. Each film has associated **ActorIDs**, which are stored as a comma-separated list to connect the films to the actors.

sql

Copy code

INSERT INTO film\_films (FilmTitle, Genres, Director, ActorIDs, FilmRating, ReviewCount, BoxOffice)

VALUES

('Iron Man', 'Action,Sci-Fi', 'Jon Favreau', '1,2', 8.5, 2500, 585000000),

('The Avengers', 'Action,Sci-Fi', 'Joss Whedon', '1,2,3', 8.8, 5000, 1500000000),

('Lost in Translation', 'Drama,Romance', 'Sofia Coppola', '2', 8.1, 1200, 100000000),

('Forrest Gump', 'Drama', 'Robert Zemeckis', '3', 9.2, 1500, 678000000),

('The Social Network', 'Drama,Biography', 'David Fincher', '9', 8.0, 1000, 225000000),

('Pulp Fiction', 'Crime,Drama', 'Quentin Tarantino', '4,10', 8.9, 3000, 213000000),

('Inception', 'Action,Sci-Fi', 'Christopher Nolan', '1,9', 8.8, 4500, 829000000),

('Titanic', 'Drama,Romance', 'James Cameron', '5', 8.7, 4300, 2200000000),

('The Martian', 'Sci-Fi,Drama', 'Ridley Scott', '9', 8.4, 3500, 630000000),

('The Revenant', 'Drama,Adventure', 'Alejandro González Iñárritu', '5,17', 8.0, 2000, 533000000);

* **ActorIDs** link actors to the films. For instance, "Iron Man" includes actors **1** and **2**, which correspond to **Robert Downey Jr.** and **Scarlett Johansson**.

**Step 3: Querying Data**

After inserting data into the tables, we can run some queries to extract useful insights.

**Task 1.5: Query to Find Actor Ratings for Films**

This query calculates the average film rating for each actor based on the films they have appeared in:

sql

Copy code

SELECT a.ActorName,

AVG(f.FilmRating) AS AvgActorFilmRating

FROM actor\_films a

JOIN film\_films f ON POSITION(CAST(a.ActorID AS TEXT) IN f.ActorIDs) > 0

GROUP BY a.ActorName;

* **Explanation**: The query joins **actor\_films** and **film\_films** on **ActorIDs** and computes the average film rating for each actor.

**Task 1.6: Query to Find Top Films Based on Box Office Earnings**

To get the top 5 films based on box office earnings:

sql

Copy code

SELECT FilmTitle,

BoxOffice

FROM film\_films

ORDER BY BoxOffice DESC

LIMIT 5;

* **Explanation**: The query retrieves the top 5 films ordered by **BoxOffice** earnings in descending order.

**Task 1.7: Query to Find Films with High Ratings and Low Review Counts**

To find films with a rating above 8.0 and review counts below 1000:

sql

Copy code

SELECT FilmTitle, FilmRating, ReviewCount

FROM film\_films

WHERE FilmRating > 8.0

AND ReviewCount < 1000;

* **Explanation**: This query helps identify films with high ratings but fewer reviews.

**Step 4: Link Actors to Their Films**

**Task 1.8: Query to Find All Films an Actor Appeared In**

For example, to find all films **Robert Downey Jr.** appeared in:

sql

Copy code

SELECT f.FilmTitle, f.FilmRating

FROM film\_films f

JOIN actor\_films a ON POSITION(CAST(a.ActorID AS TEXT) IN f.ActorIDs) > 0

WHERE a.ActorName = 'Robert Downey Jr.';

* **Explanation**: This query lists all films in which **Robert Downey Jr.** has appeared, along with their ratings.

**Step 5: Handling Slowly Changing Dimensions (SCD)**

To handle cases where actor details (like ratings) change over time, we can use **upserts** with the ON CONFLICT clause. This ensures that data is inserted if it doesn't exist or updated if it does.

Example of handling changes in actor ratings:

sql

Copy code

-- Update or insert a new actor record if the rating changes

INSERT INTO actor\_films (ActorName, Awards, BirthCountry, BirthYear, Rating, Reviews)

VALUES ('Robert Downey Jr.', 'Oscar,Golden Globe', 'USAHere's a detailed breakdown of \*\*Task 1\*\* for creating and populating the `actor\_films` and `film\_films` tables in PostgreSQL, including the associated queries and data manipulations:

---

### \*\*Step 1: Table Creation\*\*

This step is about creating the structure for the two primary tables: \*\*actor\_films\*\* and \*\*film\_films\*\*.

#### \*\*Task 1.1: Create the `actor\_films` Table\*\*

The `actor\_films` table stores information about actors, such as their names, awards, country of birth, ratings, and the number of reviews they've received.

```sql

CREATE TABLE actor\_films (

ActorID SERIAL PRIMARY KEY, -- Auto-incrementing unique ID for each actor

ActorName VARCHAR(100), -- Actor's name

Awards VARCHAR(255), -- Awards won by the actor (e.g., Oscar, BAFTA)

BirthCountry VARCHAR(100), -- Country where the actor was born

BirthYear INT, -- Actor's birth year

Rating DECIMAL(3,2), -- Average rating based on actor's film performances

Reviews INT -- Number of reviews the actor has received

);

* **ActorID**: Auto-incremented unique identifier for each actor.
* **ActorName**: Name of the actor.
* **Awards**: Awards won by the actor.
* **BirthCountry**: Actor's country of birth.
* **BirthYear**: Actor's birth year.
* **Rating**: Average rating of the actor's performances.
* **Reviews**: Total reviews for the actor's films.

**Task 1.2: Create the film\_films Table**

The film\_films table stores details about films, such as their title, genres, director, linked actors (via ActorIDs), ratings, and box office earnings.

sql

Copy code

CREATE TABLE film\_films (

FilmID SERIAL PRIMARY KEY, -- Auto-incrementing unique ID for each film

FilmTitle VARCHAR(255), -- Film title

Genres VARCHAR(255), -- Genres of the film (e.g., Drama, Action)

Director VARCHAR(100), -- Director of the film

ActorIDs VARCHAR(255), -- Comma-separated list of ActorIDs

FilmRating DECIMAL(3,2), -- Average rating of the film

ReviewCount INT, -- Number of reviews for the film

BoxOffice BIGINT -- Total box office earnings in USD

);

* **FilmID**: Auto-incremented unique identifier for each film.
* **ActorIDs**: Comma-separated list of ActorIDs who appeared in the film.
* **FilmRating**: Average rating of the film.
* **ReviewCount**: Total number of reviews for the film.
* **BoxOffice**: Total box office earnings in USD.

**Step 2: Insert Sample Data**

**Task 1.3: Insert Data into the actor\_films Table**

We populate the actor\_films table with data about 10 actors:

sql

Copy code

INSERT INTO actor\_films (ActorName, Awards, BirthCountry, BirthYear, Rating, Reviews)

VALUES

('Robert Downey Jr.', 'Oscar,Golden Globe', 'USA', 1965, 8.7, 1200),

('Scarlett Johansson', 'BAFTA', 'USA', 1984, 8.3, 1100),

('Tom Hanks', 'Oscar,Golden Globe,BAFTA', 'USA', 1956, 9.2, 1500),

('Johnny Depp', 'Golden Globe,César', 'USA', 1963, 7.8, 900),

('Leonardo DiCaprio', 'Oscar,Golden Globe', 'USA', 1974, 9.0, 1800),

('Meryl Streep', 'Oscar,Golden Globe,BAFTA', 'USA', 1949, 8.9, 1400),

('Will Smith', 'Golden Globe', 'USA', 1968, 8.0, 950),

('Natalie Portman', 'Oscar', 'Israel', 1981, 8.4, 800),

('Matt Damon', 'Golden Globe', 'USA', 1970, 8.2, 1000),

('Brad Pitt', 'Golden Globe', 'USA', 1963, 8.6, 1100);

**Task 1.4: Insert Data into the film\_films Table**

Now, insert data for 10 films, including a comma-separated list of **ActorIDs** for each film:

sql

Copy code

INSERT INTO film\_films (FilmTitle, Genres, Director, ActorIDs, FilmRating, ReviewCount, BoxOffice)

VALUES

('Iron Man', 'Action,Sci-Fi', 'Jon Favreau', '1,2', 8.5, 2500, 585000000),

('The Avengers', 'Action,Sci-Fi', 'Joss Whedon', '1,2,3', 8.8, 5000, 1500000000),

('Lost in Translation', 'Drama,Romance', 'Sofia Coppola', '2', 8.1, 1200, 100000000),

('Forrest Gump', 'Drama', 'Robert Zemeckis', '3', 9.2, 1500, 678000000),

('The Social Network', 'Drama,Biography', 'David Fincher', '9', 8.0, 1000, 225000000),

('Pulp Fiction', 'Crime,Drama', 'Quentin Tarantino', '4,10', 8.9, 3000, 213000000),

('Inception', 'Action,Sci-Fi', 'Christopher Nolan', '1,9', 8.8, 4500, 829000000),

('Titanic', 'Drama,Romance', 'James Cameron', '5', 8.7, 4300, 2200000000),

('The Martian', 'Sci-Fi,Drama', 'Ridley Scott', '9', 8.4, 3500, 630000000),

('The Revenant', 'Drama,Adventure', 'Alejandro González Iñárritu', '5,17', 8.0, 2000, 533000000);

**Step 3: Querying Data**

**Task 1.5: Query to Find Actor Ratings for Films**

To find the average rating for each actor's films:

sql

Copy code

SELECT a.ActorName,

AVG(f.FilmRating) AS AvgActorFilmRating

FROM actor\_films a

JOIN film\_films f ON POSITION(CAST(a.ActorID AS TEXT) IN f.ActorIDs) > 0

GROUP BY a.ActorName;

This query calculates the average film rating for each actor based on the films they have appeared in.

**Task 1.6: Query to Find Top Films Based on Box Office Earnings**

To retrieve the top 5 films based on box office earnings:

sql

Copy code

SELECT FilmTitle,

BoxOffice

FROM film\_films

ORDER BY BoxOffice DESC

LIMIT 5;

This query retrieves the films with the highest box office earnings.

**Task 1.7: Query to Find Films with High Ratings and Low Review Counts**

To find films with ratings greater than 8.0 but fewer than 1000 reviews:

sql

Copy code

SELECT FilmTitle, FilmRating, ReviewCount

FROM film\_films

WHERE FilmRating > 8.0

AND ReviewCount < 1000;

This query identifies films that have high ratings but are less reviewed.

**Step 4: Link Actors to Their Films**

**Task 1.8: Query to Find All Films an Actor Appeared In**

For example, to find all films **Robert Downey Jr.** has appeared in:

sql

Copy code

SELECT f.FilmTitle, f.FilmRating

FROM film\_films f

JOIN actor\_films a ON POSITION(CAST(a.ActorID AS TEXT) IN f.ActorIDs) > 0

WHERE a.ActorName = 'Robert Downey Jr.';

This query lists all films in which **Robert Downey Jr.** has appeared.

**Step 5: Handling Slowly Changing Dimensions (SCD)**

To handle updates to actor information, such as rating changes, you can use **upserts**:

sql

Copy code

-- Update or insert an actor's record

INSERT INTO actor\_films (ActorName, Awards, BirthCountry, BirthYear, Rating, Reviews)

VALUES ('Robert Downey Jr.', 'Oscar,Golden Globe', 'USA', 1965, 8.8, 1250)

ON CONFLICT (ActorID) DO UPDATE SET

Rating = EXCLUDED.Rating,

Reviews = EXCLUDED.Reviews;

This ensures that if an actor's record already exists, it is updated with the new rating and review count.

This detailed breakdown provides the necessary SQL queries to create the tables, populate them with sample data, and perform queries to extract useful information from the dataset.