Project Comics

Author: Radu Gabriel Toporan

Table of Contents

List of Figures

1. Introduction

The project is related to the comics business and the design of a comics database for a company like Excelsior, an online retailer of comic books. Comics have a long history as collectibles, and their value as such is influenced by factors like their rarity and condition. Going through periods of boom and bust, comics were originally designed to be cheap and disposable reading matter for children, but the disposability has added to their value as collectibles.

The Excelsior database is expected to be designed as a product inventory tracking system for a store that sells comics, just like Mile High Comics. The database should track the year of every comic as well as its issue number. The benefit of the Excelsior database for power users is that it allows them to write complex queries to immediately find highly specific answers. The price of each comic should be tracked, and it should reflect its condition. The system must track every comic's sale price and the purchase price paid by Excelsior to calculate profit and loss.

To manage complex numbering systems like those found in the Marvel universe, it may be best to track each new run of the comic (when the numbering returns to #1) by its start year. This will make it easier to keep track of different runs of the same title and ensure that issues are properly organised in the database. Additionally, the database should include a way to distinguish between different runs of the same title, such as by including the start year in the title or using a unique identifier. This will help avoid confusion and make it easier for users to find the specific issue they are looking for.

2. Database Plan: A Schematic View

As the database is for a comic book collection, it contains tables such as for publishers, countries, publisher countries (linking publishers to the countries they operate in), writers, artists, comics, and grades (used to grade the condition of the comics). The publishers table lists the names of the different comic book publishers, while the countries table lists the names and codes of different countries. The publisher_countries table links the publishers and countries tables. The writers and artists tables list the names of different comic book writers and artists, respectively. The comics table lists the details of different comic books, including the title, publisher, writer, artist, issue number, and publication date. Finally, the grades table lists different grades that can be used to assess the condition of a comic book, along with their descriptions and grading scale.

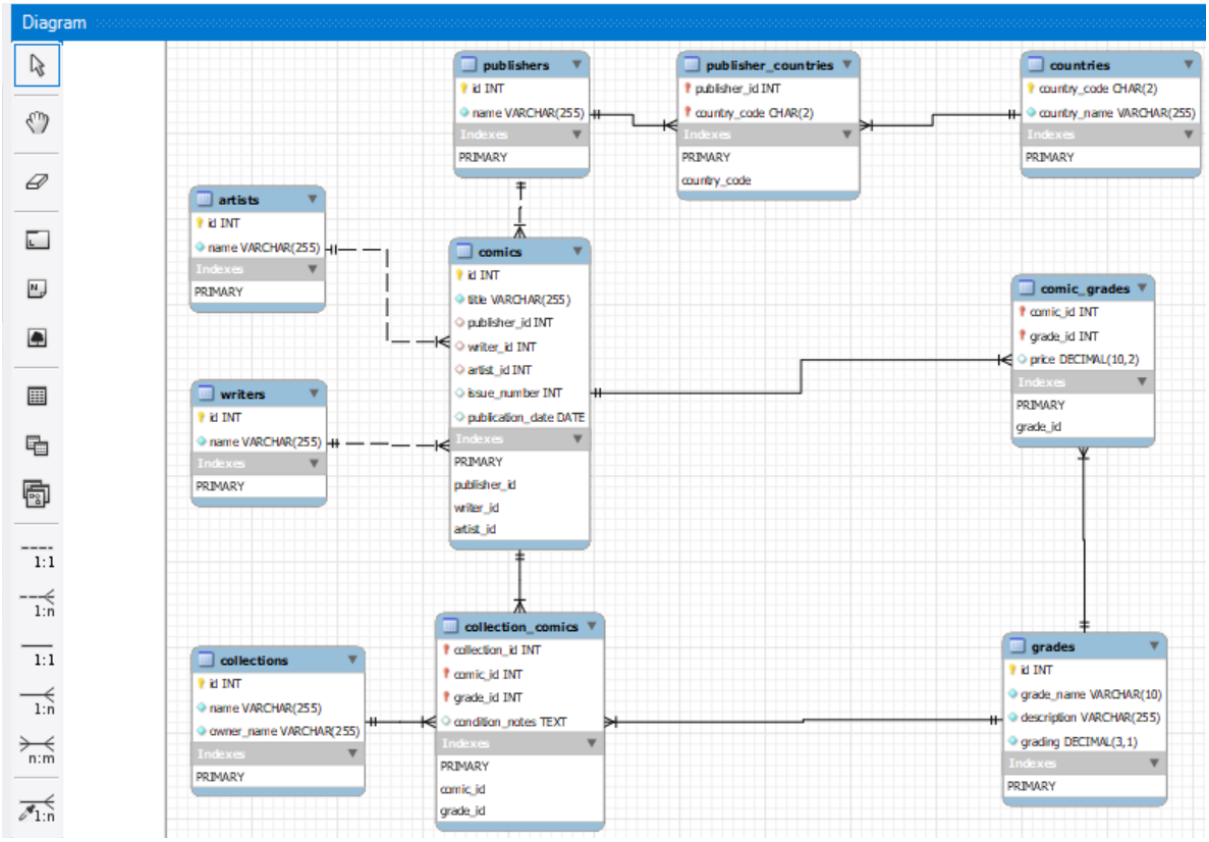

Figure 1: ER model

The design of the database tables was based on the principles of normalisation, which aims to eliminate data redundancy and inconsistencies, ensuring that each table has a clear and distinct purpose.

For the "comics" table, the attributes were chosen to represent the essential information about a comic book, such as its title, publisher, and publication date.

The "writers" and "artists" tables were created to establish a many-to-many relationship between the "comics" table and these two tables, allowing multiple writers and artists to be associated with a comic, and vice versa.

The "publishers" table serves to store information about the publishers of the comics, including their names and locations.

The "grades" table serves to define the different grades that a comic book can have, along with a description and a grading score.

The "comic_grades" table serves as an intermediary table between the "comics" and "grades" tables, allowing each comic book to have multiple grades associated with it.

The "collections" table was created to store information about the various comic book collections that exist, such as their names and owners.

The "collection_comics" table establishes a many-to-many relationship between the "collections" and "comics" tables, enabling multiple comics to be associated with each collection, and vice versa. Additionally, this table allows for the storage of condition notes specific to each comic in a collection.

Overall, this design allows for efficient storage and management of data related to comic books while also enabling a variety of relationships and associations between the different tables. It provides a flexible and expandable structure that can be adapted to meet the needs of various users and applications.

3. Database Structure: A Normalised View

1) Publishers table:

- Stores information about publishers of comic books.
- Contains two columns: "id" and "name".
- The "id" column is the primary key and is used to uniquely identify each publisher.
- The "name" column is a string that stores the name of the publisher.

2) Countries table:

- Stores information about countries.
- Contains two columns: "country_code" and "country_name".
- The "country_code" column is the primary key and stores the two-letter country code.
- The "country_name" column is a string that stores the full name of the country.

3) Publisher_countries table:

- Stores information about the countries where a publisher operates.
- Contains two columns: "publisher_id" and "country_code".
- The "publisher_id" column is a foreign key that references the "id" column of the "publishers" table.
- The "country_code" column is a foreign key that references the "country_code" column of the "countries" table.
- The primary key of the table is a combination of the "publisher_id" and "country_code" columns.

4) Writers table:

- Stores information about the writers of comic books.
- Contains two columns: "id" and "name".
- The "id" column is the primary key and is used to uniquely identify each writer.
- The "name" column is a string that stores the name of the writer.

5) Artists table:

- Stores information about comic book artists.
- Contains two columns: "id" and "name".
- The "id" column is the primary key and is used to uniquely identify each artist.
- The "name" column is a string that stores the name of the artist.

6) Comics table:

- Stores information about comic books.

- Contains seven columns: "id", "title", "publisher_id", "writer_id", "artist_id", "issue_number", and "publication_date".
- The "id" column is the primary key and is used to uniquely identify each comic.
- The "title" column is a string that stores the title of the comic book.
- The "publisher_id" column is a foreign key that references the "id" column of the "publishers" table.
- The "writer_id" column is a foreign key that references the "id" column of the "writers" table.
- The "artist_id" column is a foreign key that references the "id" column of the "artists" table.
- The "issue_number" column is an integer that stores the issue number of the comic book.
- The "publication_date" column is a date that stores the publication date of the comic book.

7) Grades table:

- Stores information about comic book grades.
- Contains four columns: "id", "grade_name", "description", and "grading".
- The "id" column is the primary key and is used to uniquely identify each grade.
- The "grade_name" column is a string that stores the name of the grade.
- The "description" column is a string that describes the condition of the comic book.
- The "grading" column is a decimal that stores the grading score of the comic book.

8) Comics_grades table:

- Stores information about the different grades of a comic book and their respective prices.
- Contains three columns: "comic_id", "grade_id", and "price".
- The "comic_id" column is a foreign key that references the "id" column in the Comics table.
- The "grade_id" column is a foreign key that references the "id" column in the Grades table.
- The "price" column is a decimal that stores the price of the comic book for the corresponding grade.

9) Collections table:

- Stores information about comic book collections.

- Contains three columns: "id", "name", and "owner_name".
- The "id" column is the primary key and is used to uniquely identify each collection.
- The "name" column is a string that stores the name of the collection.
- The "owner_name" column is a string that stores the name of the owner of the collection.

10) Collection_comics table:

- Stores information about the comics included in a collection and their respective grades and conditions.
- Contains four columns: "collection_id", "comic_id", "grade_id", and "condition_notes".
- The "collection_id" column is a foreign key that references the "id" column in the Collections table.
- The "comic_id" column is a foreign key that references the "id" column in the Comics table.
- The "grade_id" column is a foreign key that references the "id" column in the Grades table.
- The "condition_notes" column is a text field that stores any notes about the condition of the comic book in the collection.

In the database design, normalisation is the process of organising data in a way that reduces redundancy and dependency, improves data integrity, and helps ensure that data is consistent and accurate. There are four main levels of normalisation: First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF), and Boyce-Codd Normal Form (BCNF).

The normalisation levels can be analysed as follows:

- First Normal Form (1NF):

The database is already in 1NF because all the tables have a primary key and each column contains only atomic values.

- Second Normal Form (2NF):

The database is also in 2NF because all the non-key attributes of the tables are functionally dependent on the entire primary key. In other words, there are no partial dependencies in any table.

- Third Normal Form (3NF):

The database is in 3NF because all the non-key attributes of the tables are not transitively dependent on the primary key. In other words, there are no transitive dependencies in any table.

- Boyce-Codd Normal Form (BCNF):

The database is in BCNF because all tables have no non-trivial dependencies between the primary key and any other attributes. Therefore, all tables meet the BCNF definition of being in 3NF and having no non-trivial functional dependencies on any candidate key.

In summary, the comics database is well normalised and meets the criteria of 1NF, 2NF, 3NF, and BCNF.

4. Database Views

```
-- Create a view that retrieves all the countries where a specific publisher operates
CREATE VIEW publisher_country_specific_view AS
SELECT publishers.name AS publisher_name, countries.country_name
FROM publisher_countries
JOIN publishers ON publisher_countries.publisher_id = publishers.id
JOIN countries ON publisher_countries.country_code = countries.country_code
WHERE publishers.name = 'Marvel Comics';
select * from publisher_country_specific_view;
```

Figure 2: View that retrieves all the countries where a specific publisher operates

This view is created to provide a straightforward way to retrieve all the countries where a specific publisher, in this case, Marvel Comics, operates.

The view joins the publisher_countries, publishers, and countries tables to link the publisher to its corresponding countries of operation. The WHERE clause filters the results to only include data for the specified publisher name.

Creating this view allows users to quickly retrieve this information without having to write out the SQL code to join the tables each time they need this information. It also simplifies the process of updating the query to include additional publishers or other relevant data.

In this particular case, creating a view is a better choice than creating a table because the data in the view is not intended to be modified. The purpose of this view is to retrieve information on the countries where a specific publisher operates based on existing data in the publisher_countries, publishers, and countries tables.

Creating a table for this data would require duplicating the data from those tables, which could lead to data inconsistencies and redundancy. Also, the data in the publisher_countries table could be updated frequently, which would require updating the table that duplicates it. By creating a view, any changes to the publisher_countries table or any related tables will be automatically reflected in the view.

Moreover, a view provides a simplified way to retrieve information from multiple tables. This is particularly useful in this case, where data from three tables needs to be joined to provide a specific piece of information. Creating a view saves time and effort, as users can query the view directly rather than writing out the SQL code to join the tables each time they need this information.

```sql
-- A view that lists all the comics with their publisher name, writer name, and artist name
CREATE VIEW comics_details AS
SELECT comics.title, publishers.name AS publisher_name, writers.name AS writer_name, artists.name AS artist_name
FROM comics
JOIN publishers ON comics.publisher_id = publishers.id
JOIN writers ON comics.writer_id = writers.id
JOIN artists ON comics.artist_id = artists.id;
select * from comics_details;
```

Figure 3: View that lists all the comics with their publisher name, writer name, and artist name

This view, named comics_details, provides a convenient way to retrieve information about comics, including their title, publisher name, writer name, and artist name. By joining the comics, publishers, writers, and artists tables, this view aggregates data from multiple tables into a single result set. This simplifies querying the database, as it eliminates the need to write more complex queries involving joins.

This view is useful when we need to retrieve all the details of a comic book, including the name of the publisher, writer, and artist, which are stored in different tables. It is also helpful when we need to search for specific comics based on their details, such as the publisher or the writer.

In this particular case, a view is a better option than a table because it simplifies the process of retrieving information from multiple tables. The comics_details view joins the comics, publishers, writers, and artists tables to retrieve information about each comic's publisher, writer, and artist.

If we were to create a table to store this information, we would need to update the table every time a new comic, publisher, writer, or artist is added to the database. Additionally, if we were to make changes to the information in the comics, publishers, writers, or artists tables, we would need to manually update the corresponding information in the table that stores the comic details.

Using a view eliminates these potential issues. The view automatically updates whenever information is added or modified in any of the underlying tables, ensuring that the information is always up-to-date. Additionally, the view eliminates the need to write out the SQL code to join the tables each time the information is needed, providing a simpler and more efficient way to retrieve the desired data.

```sql
-- A view that lists the top 5 publishers by the number of comics they have published
CREATE VIEW top_publishers AS
SELECT publishers.name, COUNT(comics.id) AS comic_count
FROM publishers
JOIN comics ON comics.publisher_id = publishers.id
GROUP BY publishers.id
ORDER BY comic_count DESC
LIMIT 5;
select * from top_publishers;
```

Figure 4: View that lists the top 5 publishers by the number of comics they have published

The "top_publishers" view provides a list of the top five publishers based on the number of comics they have published. This view is useful for analysing the distribution of comics among publishers and identifying which publishers are the most prolific.

By joining the "publishers" and "comics" tables and grouping by publisher, we can count the number of comics published by each publisher. The results are then sorted in descending order by comic count and limited to the top five publishers.

This view can be useful for various stakeholders in the comic industry, such as publishers, collectors, and investors. Publishers can use this view to benchmark their publishing activity against their competitors and identify potential growth opportunities. Collectors and investors can use this view to identify which publishers have a significant presence in the market and which publishers are likely to have valuable comics in their collections.

Creating a table with the information of the top 5 publishers would require updating it every time a new comic is added to the database, which could be inefficient and error-prone.

By creating a view instead, we can always get the most up-to-date information by querying the view without having to update a separate table.

```sql
-- A view that lists all the comics with their publisher name, writer name, and artist name,
-- but only includes comics with a publication date after 1990:
CREATE VIEW recent_comics_details AS
SELECT comics.title, publishers.name AS publisher_name, writers.name AS writer_name, artists.name AS artist_name
FROM comics
JOIN publishers ON comics.publisher_id = publishers.id
JOIN writers ON comics.writer_id = writers.id
JOIN artists ON comics.artist_id = artists.id
WHERE comics.publication_date > '1990-01-01';
select * from recent_comics_details;
```

Figure 5: View that lists all the comics with their publisher name, writer name, and artist name but only includes comics with a publication date after 1990

The comics table contains information about various comics, such as their title, publisher, writer, artist, issue number, and publication date. The recent_comics_details view is created to provide a list of all the comics with their publisher name, writer name, and artist name, but only includes comics with a publication date after 1990.

This view could be useful for a comic bookstore or collector who wants to keep track of more recent comics and their creators. It provides a way to easily see the relevant details of recent comics without having to sift through all the older comics in the database. Additionally, it can help identify which publishers, writers, and artists are currently active in the industry and producing latest content.

The "recent_comics_details" view is better than a table because it provides a specific set of data that meets certain criteria, in this case comics published after 1990, while also including additional information about the publishers, writers, and artists involved in creating those comics.

By using a view, we can define and save this specific query and easily access it in the future without having to write the query again. This saves time and effort and also ensures consistency in the data returned by the view.

Additionally, if there are any changes to the underlying tables (comics, publishers, writers, and artists), the view will automatically reflect those changes without requiring any additional updates or modifications. This makes it easier to maintain and manage the database as a whole.

```
-- View to find the highest and the lowest graded items
CREATE VIEW high_low_grades AS
SELECT * FROM (
  SELECT *, RANK() OVER (ORDER BY grading DESC) AS rank_high,
         RANK() OVER (ORDER BY grading ASC) AS rank_low
  FROM grades
) AS ranked_grades
WHERE rank_high = 1 OR rank_low = 1;
select * from high_low_grades;
```

Figure 6: View to find the highest and lowest-graded items

The purpose of the grades table is to store information about different grades for comic books, including their names, descriptions, and numerical grading scale. However, it is

important to ensure that the grading values fall within an appropriate range, in this case between 0.1 and 10.0. To enforce this constraint, a trigger has been created that adjusts the grading value if it falls outside this range.

The high_low_grades view is designed to provide a quick reference for the highest and lowest-graded items in the grades table. This view utilises the RANK() function to assign a ranking to each grade based on its grading value, with the highest grade receiving a rank of 1 and the lowest grade receiving a rank of n, where n is the total number of grades in the table. The view then selects only the rows where the rank is equal to 1, indicating the highest and lowest-graded items.

This view can be useful for quickly identifying the most valuable or rare items in a collection or for identifying areas where a collection may be lacking in certain grades. It can also be used as a quick reference for grading standards and terminology.

The high_low_grades view is better than a table because it provides a dynamic and easily updatable way to retrieve the highest and lowest-graded items without needing to write complex queries every time. If we were to use a table instead, we would need to update it every time a new grade is added or the grading scale changes, which would be time-consuming and prone to errors.

```
-- View to show the collections with the most expensive comics
CREATE VIEW expensive_collections_view AS
SELECT
  c.id AS collection_id,
  c.name AS collection_name,
  SUM(cg.price) AS total_price
FROM collections c
JOIN collection_comics cc ON c.id = cc.collection_id
JOIN comic_grades cg ON cc.comic_id = cg.comic_id AND cc.grade_id = cg.grade_id
GROUP BY c.id
ORDER BY total_price DESC;
```

Figure 7: View to show the collections with the most expensive comics

The expensive_collections_view is a view that provides information about the collections with the most expensive comics. It does this by joining the collections, collection_comics, and comic_grades tables and grouping by collection ID to calculate the total price of all the comics in each collection. The view is created using a JOIN statement to link the collections

table with the collection_comics and comic_grades tables. The SUM function is used to calculate the total price of all the comics in each collection. The view is then sorted in descending order based on the total price of each collection.

This view can be useful for collectors who are interested in knowing which collections have the most valuable comics. It can help them make informed decisions about which collections to purchase or which collections to prioritise when organising their collection. It can also be useful for comic bookstores or dealers who are interested in knowing which collections may be worth more than others and how to price their inventory accordingly.

The "expensive_collections_view" is better than a table because it provides a way to easily retrieve and display information from multiple tables. In this case, it allows us to see the total price of all the comics in each collection by joining the "collections", "collection_comics", and "comic_grades" tables. As new comics are added to a collection or prices are updated, the information in the view will automatically update to reflect these changes.

5. Procedural Elements

Procedural elements, such as stored procedures and triggers, can be useful in certain scenarios where complex logic needs to be executed on the database server side. However, they also have some disadvantages:

- Portability: Procedural elements are often specific to a particular database management system, which can make it harder to migrate to a different system or switch to a different database technology altogether.
- Maintainability: Procedural elements can be difficult to maintain and debug, especially as the size and complexity of the database grow.
- Security: Procedural elements can introduce additional security risks, such as SQL injection attacks.

Therefore, many modern databases prefer to rely on declarative constructs, such as views, constraints, and indexes, to ensure data consistency and perform advanced querying. This approach provides a simpler and more standardised way to manage and operate the database, and it also tends to be more compatible with modern application architectures, such as microservices and serverless computing.

Triggers are an essential aspect of database management that allows developers to automate various actions within the database. In this report, we will discuss the importance of triggers in managing databases.

Importance of triggers:

Triggers are database objects that are executed automatically in response to certain events such as updates, inserts, or deletes. They allow developers to enforce business rules, perform data validation, and ensure data integrity. Triggers provide a powerful mechanism for implementing complex business rules that can be difficult or impossible to achieve using other database features.

Triggers can also help improve data quality by automatically correcting data as it is inserted or updated. For example, triggers can be used to ensure that all names and country codes are capitalised, or that prices are correctly entered. Additionally, triggers can be used to enforce security policies, such as preventing unauthorised access to sensitive data.

My implementation of triggers:

As a developer, I have implemented several triggers to ensure data accuracy and consistency. For instance, I have implemented triggers that capitalise names, country names, and country codes to ensure consistency across the database. I have also implemented triggers that check for the correct insertion of prices and ensure that grades are between 0.1 and 10.

Moreover, I have implemented triggers to ensure that the writer's and artist's names are correctly formatted and spelt and that titles are consistently formatted. These triggers have significantly improved the quality of the data in my databases and reduced the workload required to maintain the data.

They help ensure data accuracy and consistency, enforce business rules, and improve data quality. My implementation of triggers has improved the quality of data in my databases, and I believe that developers should consider triggers as an important aspect of their database management strategy.

```
CREATE TABLE publishers (
    id INT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);
DELIMITER //
CREATE TRIGGER capitalise_publisher_name
BEFORE INSERT ON publishers
FOR EACH ROW
BEGIN
    SET NEW.name = CONCAT(UPPER(SUBSTR(NEW.name, 1, 1)), SUBSTR(NEW.name, 2));
END //
DELIMITER ;
```

Figure 8: Trigger capitalise_publisher_name (publishers table)

The trigger capitalise_publisher_name is designed to automatically capitalise the first letter of a publisher's name whenever a new record is inserted into the publishers table.

This trigger is useful because it ensures consistency in the way publisher names are displayed throughout the database. Without the trigger, there could be inconsistencies in capitalisation between records, which could make it difficult to search for or group records by publisher name.

By using this trigger, it also helps to maintain data quality and accuracy, which is essential for any database that will be used in a production environment. Additionally, it saves time

and effort for developers and users who would otherwise have to manually capitalise each new record that is added to the publishers table.

Overall, the capitalise_publisher_name trigger is a simple but effective way to ensure consistency and accuracy in the database and make it easier to use and maintain over time.

```
CREATE TABLE countries (
    country_code CHAR(2) PRIMARY KEY,
    country_name VARCHAR(255) NOT NULL
);
DELIMITER //
CREATE TRIGGER format_country_info
BEFORE INSERT ON countries
FOR EACH ROW
BEGIN
    SET NEW.country_code = UPPER(NEW.country_code);
    SET NEW.country_name = CONCAT(UCASE(LEFT(NEW.country_name, 1)), SUBSTRING(NEW.country_name, 2));
END //
DELIMITER ;
```

Figure 9: Trigger format_country_info (countries table)

This trigger is designed to format the information inserted into the countries table by ensuring that the country code is in uppercase and the country name is properly capitalised. This helps to ensure consistency and accuracy in the data stored in the table.

The trigger is called format_country_info and is set to execute before each INSERT operation on the countries table. The trigger first converts the country_code value to uppercase using the UPPER function. Next, the trigger capitalises the first letter of the country_name value using the LEFT and SUBSTRING functions and the UCASE function, which returns the string in uppercase.

Overall, this trigger helps to ensure that the country codes and names stored in the countries table are consistent and correctly formatted, which can help to prevent errors and make it easier to analyse and report on the data in the table.

```
CREATE TABLE writers (
    id INT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);
DELIMITER //
CREATE TRIGGER capitalize_writer_name
BEFORE INSERT ON writers
FOR EACH ROW
BEGIN
  DECLARE first_letter CHAR(1);
  SET first_letter = LEFT(NEW.name, 1);
  SET NEW.name = CONCAT(UCASE(first_letter), SUBSTR(NEW.name, 2));
END //
DELIMITER ;
```

Figure 10: Trigger capitalise_writer_name (writers table)

This trigger is designed to capitalize the first letter of a writer's name when inserting it into the "writers" table. This helps to ensure consistency in the data and makes it easier to search and sort by writer's name. By using a trigger, this process is automated and eliminates the need for manual data entry or updates. It also helps to maintain the integrity of the data and ensures that all names are correctly formatted. Overall, this trigger improves the quality of the data and makes it more user-friendly for those accessing the database.

```
CREATE TABLE artists (
    id INT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);
DELIMITER //
CREATE TRIGGER capitalize_artist_name
BEFORE INSERT ON artists
FOR EACH ROW
BEGIN
    SET NEW.name = CONCAT(UPPER(LEFT(NEW.name, 1)), SUBSTRING(NEW.name, 2));
END //
DELIMITER ;
```

Figure 11: Trigger capitalise_artist_name (artists table)

The trigger capitalize_artist_name is used to capitalize the first letter of an artist's name before it is inserted into the artists table. This trigger ensures consistency in the formatting of the artist names and makes it easier to search and sort the data. Capitalizing the first letter also enhances the readability of the data, which can be especially helpful when presenting it to users.

For example, if an artist's name is entered as "jim lee" with all lowercase letters, the trigger will modify it to "Jim lee" with a capitalized first letter. This makes it easier to distinguish the artist's name from other text in the database and presents it in a more professional and consistent manner.

Overall, the capitalize_artist_name trigger helps ensure data quality and consistency in the artists table, making it easier for users to work with and interpret the data.

```
CREATE TABLE comics (
    id INT PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    publisher_id INT,
    writer_id INT,
    artist_id INT,
    issue_number INT,
    publication_date DATE,
    FOREIGN KEY (publisher_id) REFERENCES publishers(id),
    FOREIGN KEY (writer_id) REFERENCES writers(id),
    FOREIGN KEY (artist_id) REFERENCES artists(id)
);
-- ensures the title column is always capitalized when inserting or updating a row
DELIMITER //
CREATE TRIGGER capitalize_title
BEFORE INSERT ON comics
FOR EACH ROW
BEGIN
    SET NEW.title = CONCAT(UCASE(LEFT(NEW.title, 1)), LCASE(SUBSTRING(NEW.title, 2)));
END//
```

Figure 12: Trigger capitalise_title (comics table)

The "capitalize_title" trigger ensures that the "title" column of the "comics" table is always capitalized, regardless of whether the data is being inserted or updated. This helps to maintain consistency and readability in the data by ensuring that the first letter of each word in the title is capitalized.

The trigger achieves this by using the CONCAT function to concatenate the uppercase first letter of the title with the lowercase remainder of the title. The trigger is set to run before each insertion into the "comics" table and is triggered for each row that is inserted.

By using this trigger, we can be confident that the "title" column of the "comics" table will always be formatted in a consistent and readable way, which can be helpful for both users and developers who may be working with the data.

```
CREATE TABLE grades (
    id INT PRIMARY KEY,
    grade_name VARCHAR(10) NOT NULL,
    description VARCHAR(255) NOT NULL,
    grading DECIMAL(3,1) NOT NULL
);
-- grades should not be >10 or <0.1
delimiter //
CREATE TRIGGER insert_grade
BEFORE INSERT ON grades
FOR EACH ROW
BEGIN
  IF NEW.grading > 10 THEN
    SET NEW.grading = 10.0;
  ELSEIF NEW.grading < 0.1 THEN
    SET NEW.grading = 0.1;
  END IF;
END;//
delimiter ;
```

Figure 13: Trigger insert_grade (grades table)

This trigger is used to prevent invalid data from being inserted into the grades table by ensuring that the grading value is always within the acceptable range of 0.1 to 10.0. This is important because grades are used to evaluate the condition and value of comic books, and allowing values outside this range could lead to inaccurate or inconsistent grading, which could negatively impact the market value of comic books.

The trigger is defined as a BEFORE INSERT trigger, which means that it will execute before any data is inserted into the grades table. It uses an IF statement to check whether the value of

grading is greater than ten or less than 0.1. If either of these conditions is true, the trigger will modify the value of grading to be either 10 or 0.1, respectively.

The use of this trigger ensures that only valid grading values are allowed in the grades table, which helps to maintain the accuracy and consistency of the grading system for comic books.

```
CREATE TABLE comic_grades (
    comic_id INT,
    grade_id INT,
    price DECIMAL(10, 2),
    PRIMARY KEY (comic_id, grade_id),
    FOREIGN KEY (comic_id) REFERENCES comics(id),
    FOREIGN KEY (grade_id) REFERENCES grades(id)
);
-- This trigger will check the maximum grading for the grade_id of a newly inserted row in
-- the comic_grades table, and if the price of the row is greater than the maximum grading,
-- it will set the price to the maximum grading.
DELIMITER //
CREATE TRIGGER check_comic_price
BEFORE INSERT ON comic_grades
FOR EACH ROW
BEGIN
    DECLARE max_price DECIMAL(10, 2);
    SELECT grading INTO max_price FROM grades WHERE id = NEW.grade_id;
    IF NEW.price > max_price THEN
        SET NEW.price = max_price;
```

Figure 14: Trigger check_comic_price (comic_grades table)

The purpose of this trigger is to ensure that the price of a comic in a particular grade is not set higher than the maximum grading value for that grade, as defined in the "grades" table. This trigger is important to ensure data integrity and accuracy, as it prevents invalid data from being entered into the "comic_grades" table. By setting the price to the maximum grading value if it exceeds this value, the trigger ensures that the price is within a valid range for the specified grade.

For example, if a comic is assigned a grade of "Near Mint" with a maximum grading of 9.4, and a price of 10,000 is entered for that comic in the "comic_grades" table, the trigger will automatically set the price to the maximum grading of 9.4, which is a more accurate representation of the comic's value.

Overall, this trigger helps to maintain data accuracy and consistency, ensuring that prices for comics are assigned correctly based on their assigned grade and preventing data entry errors that could lead to inaccurate data and financial loss.

```
CREATE TABLE collections (
    id INT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    owner_name VARCHAR(255) NOT NULL
);
DELIMITER //
CREATE TRIGGER capitalize_collection_names
BEFORE INSERT ON collections
FOR EACH ROW
BEGIN
    SET NEW.name = CONCAT(UCASE(LEFT(NEW.name, 1)), SUBSTRING(NEW.name, 2));
    SET NEW.owner_name = CONCAT(UCASE(LEFT(NEW.owner_name, 1)), SUBSTRING(NEW.owner_name, 2));
END//
DELIMITER ;
```

Figure 15: Trigger capitalise_collection_names (collections table)

This trigger will capitalize the first letter of the name and owner_name fields of a newly inserted row in the collections table. This ensures consistency and uniformity in the naming convention of collections, making it easier to search and filter the data. Capitalizing the first letter also improves the readability and aesthetics of the data.

6. Example Queries: The Database In Action

```
-- Show the top 5 most expensive comics
SELECT comics.title, comic_grades.price
FROM comics
JOIN comic_grades ON comic_grades.comic_id = comics.id
ORDER BY comic_grades.price DESC
LIMIT 5;
```

Figure 16: Select (Show the top five most expensive comics)

The given SELECT query retrieves the top five most expensive comics from the database. To do this, it joins the comics and comic_grades tables on the comic_id column and retrieves the comic title and price from the corresponding rows. Then, it orders the results in descending order by price and limits the output to the top five rows.

This query is useful for various purposes, such as identifying the most valuable comics in a collection or determining which comics are in high demand and command premium prices in the market. By using a JOIN operation, it combines data from two tables and retrieves information that is not available in a single table. The ORDER BY clause sorts the results in a specific order, and the LIMIT clause restricts the number of rows returned.

However, this query does not take into account other factors that may affect the value of a comic, such as its condition, rarity, or historical significance. Moreover, the prices in the comic_grades table may not always accurately reflect the actual market value of a comic, as they are based on a specific grading system and may vary depending on the buyer and seller. Therefore, it is important to use this query as a starting point and conduct further research or analysis before making any decisions based on the results.

```sql
-- Show the comics in each collection with their grades and condition notes
SELECT
  cc.collection_id,
  c.title,
  g.grade_name,
  cc.condition_notes
FROM collection_comics cc
JOIN comics c ON cc.comic_id = c.id
JOIN grades g ON cc.grade_id = g.id;
```

Figure 17: Select (Show the comics in each collection with their grades and condition notes)

This SELECT query is used to retrieve the comics in each collection along with their grades and condition notes from the "collection_comics" table.

The query joins the "collection_comics" table with the "comics" and "grades" tables using their respective foreign keys. It selects the collection ID, comic title, grade name, and condition notes from the tables.

This query can be useful for collectors who want to keep track of their collection and the condition of each comic. It can also be used to evaluate the value of a collection based on the grades and conditions of the comics within it. Overall, this query provides valuable information about the comics in each collection and their respective conditions.

```sql
-- Show the total value of each collection based on the condition of the comics
SELECT
  cc.collection_id,
  SUM(g.grading * cg.price) AS total_value
FROM collection_comics cc
JOIN comic_grades cg ON cc.comic_id = cg.comic_id AND cc.grade_id = cg.grade_id
JOIN grades g ON cc.grade_id = g.id
GROUP BY cc.collection_id;
```

Figure 18: Select (Show the total value of each collection based on the condition of the comics)

This SQL query selects the total value of each collection based on the condition of the comics within each collection. The collection_comics table stores the relationship between a comic and a collection, including the grade of the comic and any condition notes. The comic_grades table maps comic IDs to prices based on their grade, while the grades table defines the different grades and their corresponding grading values.

The query uses JOINs to join the collection_comics, comic_grades, and grades tables based on the comic ID and grade ID. It then calculates the total value of each comic within a collection by multiplying the grading value and price of the comic based on its grade and then summing these values to calculate the total value of the collection. Finally, it groups the results by collection ID to display the total value of each collection.

This query can be useful for tracking the value of a comic book collection, particularly for collectors who are interested in the value of their collection based on the condition of the comics. By calculating the total value of each collection based on the condition of the comics within it, collectors can get a better understanding of the overall value of their collection and how it may change over time.

```
SELECT c.title, c.issue_number, c.publication_date, w.name AS writer_name, a.name AS artist_name
FROM comics c
JOIN writers w ON c.writer_id = w.id
JOIN artists a ON c.artist_id = a.id
WHERE c.publisher_id = 1;
```

Figure 19: Select (comics published by a specific publisher, identified by the ID)

This query retrieves information about comics that were published by a specific publisher, in this case, identified by the ID 1. The information includes the title of the comic, its issue number, publication date, the name of the writer, and the name of the artist.

The query achieves this by joining three tables: comics, writers, and artists. It uses the publisher_id column from the comics table to filter out only the comics published by the specified publisher. It then joins the writers and artists tables to retrieve the names of the writer and artist for each comic.

Overall, this query can be useful for someone who wants to get a list of all comics published by a particular publisher, along with the names of their respective writers and artists.

7. Conclusions

In conclusion, this database and its associated views and triggers provide a solid foundation for managing a comic book collection. The structure allows for easy tracking of individual comics, their grades, and their condition within a collection. Additionally, the views provide valuable insights into the most expensive collections and the highest and lowest-graded items.

Here are some additional thoughts on how the work presented here might be built upon in the future:

- Scalability: As the amount of data grows, it may become necessary to optimise the database schema and queries to improve performance. This could involve partitioning data across multiple servers or using distributed databases.
- Data Analytics: By analysing data on collections and comics, we could identify patterns in buying and selling behaviour, track changes in the popularity of different comics or collections over time, and predict future trends. This could be useful for collectors and investors looking to make informed decisions about buying and selling.
- User Interface: Currently, the database is accessed through SQL queries. A more user-friendly interface could be developed to allow users to search for and browse collections and comics based on various criteria such as title, grade, price, or publisher.
- Social Features: An additional feature that could be added is a social aspect to the platform where users can share their collections with others, rate comics, and participate in discussion forums. This would create a sense of community around the platform and could potentially lead to increased engagement and user retention.
- Integration with E-Commerce Platforms: By integrating with e-commerce platforms like eBay or Amazon, collectors could easily search for and purchase comics from within the platform without having to navigate to a separate website.
- Artificial Intelligence: As the database grows, there may be opportunities to use artificial intelligence and machine learning to assist with grading comics or predicting market trends. For example, an AI-powered tool could automatically grade a comic based on its condition or predict which comics are likely to appreciate in value.

In the future, this database could be expanded to include additional tables and views, such as a table for tracking comic book creators or a view for identifying the most popular comics

within a collection. Furthermore, it could be adapted to support other types of collectibles, such as sports cards or action figures, by adding additional tables and modifying the existing ones as needed.

Overall, this project demonstrates the importance of good database design and highlights the benefits of using a relational database management system to manage complex data.

References

C.J. Date. (2012). Database Design and Relational Theory: Normal Forms and All That Jazz. O'Reilly Media, Inc.

Beaulieu, A. (2020). Learning SQL: Generate, Manipulate, and Retrieve Data. O'Reilly Media, Inc.