

PROJECT EXCELSIOR



ADITYA TALALUR PRAKASH

22201360

aditya.talalurprakash@ucdconnect.ie

Table of Contents

Section 1: Introduction	4
Section 2: Database Plan: A Schematic View	5
Section 3: Database Structure: A Normalised View	7
Section 4: Database Views	14
Section 5: Procedural Elements	20
Section 6: Example Queries : Database in Action	29
Section 7: Conclusion	36
Acknowledgements	36
References	37

List of Figures

Figure 1: ER Diagram	5
Figure 2: All Comics	15
Figure 3: Marvel Comics	16
Figure 4: DC Comics	15
Figure 5: Order by user	18
Figure 6: Top 5 Best Selling Comics	19
Figure 7: Popular Comic Book Writers	20
Figure 8: Set Selling Price	21
Figure 9: Update Selling Price – Before	22
Figure 10: Update Selling Price – After	22
Figure 11: Result after placing order	25
Figure 12: Data inserted into orders table	25
Figure 13: Data inserted into orderItems table	25
Figure 14: Data inserted into salesInfo table	26
Figure 15: After status updated to processed	27
Figure 16: Status updated in orders table	27
Figure 17: After status updated to shipped	27
Figure 18: After status updated to delivered	27
Figure 19: Cancel failed	28
Figure 20: Orders table after cancelling order 2	28
Figure 21: Total Value of Inventory	29
Figure 22: Comics out of stock	30
Figure 23: Comics sold in last 30 days	31
Figure 24: Spider Man comics with price less than \$200.	32
Figure 25: Books that can be bought below \$100	33
Figure 26: Revenue by customer	34
Figure 27: Comics bought together	34
Figure 28: List of all special comics	35

1. Introduction

The vision for designing a database for Excelsior is to create a comprehensive system that can manage and organize the complex and diverse information related to the comic book industry. The database needs to distinguish between comic books and graphic novels, and also consider the different factors that influence the value of these products, such as issue numbers, comic series, characters, creators, physical condition, and edition printing. The database must also provide an efficient search and query functionality for users to easily navigate through the vast collection of comics and graphic novels offered by Excelsior.

Overall, the database should be able to support the business operations of Excelsior and enhance the customer experience by providing relevant and accurate information on their products.

The domain of the project Excelsior is the comic book industry, with a focus on online retail of both new comics and past issues of interest to collectors. The project aims to design a database system that can efficiently manage and store information on the vast inventory of comics and graphic novels available at Excelsior.

The intended application of the database is to provide a user-friendly interface for customers to browse and purchase comics and graphic novels, as well as for employees to manage inventory, track sales, and update pricing and availability. The database will need to handle a large amount of data, including information on comic series, issue numbers, publication dates, writers, artists, characters, and condition grades.

In addition, the database will need to provide tools for tracking the value of individual comics and graphic novels, as well as for managing the physical location of items in the warehouse. The goal of the project is to create a robust and efficient database system that can support Excelsior's online retail operations and help the company serve its customers and collectors more effectively.

The database design should facilitate efficient data storage, retrieval, and analysis, which will support the larger system's goals of improving business processes, increasing productivity, and enhancing decision-making capabilities.

The database design should also ensure data integrity and security, as it will contain sensitive and confidential information. The database should have appropriate access controls to ensure that only authorized personnel can access and modify the data.

Furthermore, the Excelsior database design should integrate with other systems and applications used by the company. For example, it should enable data exchange between the

database and other software applications used by different departments within the company, such as the ERP system, CRM software, and financial management tools. This integration will facilitate seamless data flow and improve the accuracy and reliability of the information used for decision-making and operational processes.

2. Database Plan: A Schematic View

An ER diagram (Entity-Relationship diagram) is a graphical representation of entities and their relationships to each other within a database. It illustrates the structure of the database and helps to visualize how the different tables and fields relate to each other. ER diagrams use various symbols such as rectangles, diamonds, and lines to represent entities, attributes, and relationships between entities. They are useful for both developers and stakeholders to understand the design of a database and ensure that it meets the requirements of the system.

According to the relational model, data in a relational database is stored in relations, which are perceived by the user as tables. Each relation is composed of tuples (records) and attributes (fields). (2, *SQL Queries for Mere Mortals*, Page – 6).

E R Diagram

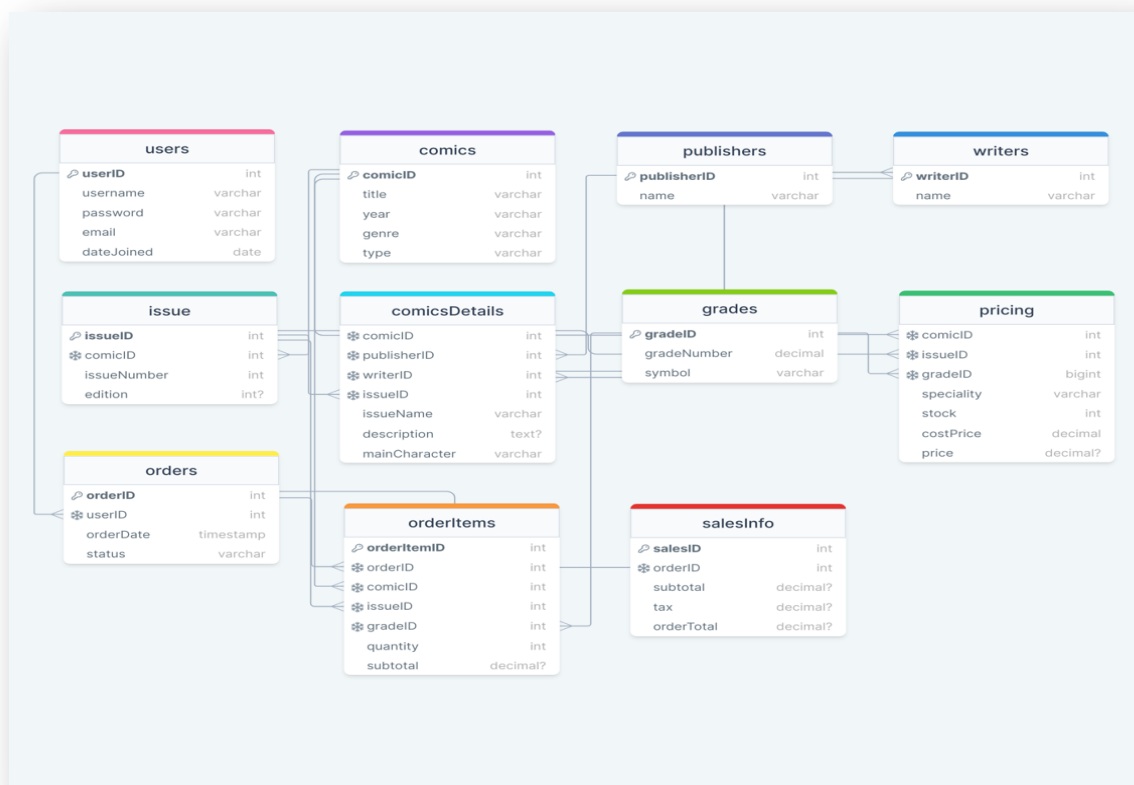


Figure 1: ER Diagram

- **Users and Orders:** Each user can place multiple orders, and each order belongs to a single user (One-to-many).
- **Orders and OrderItems:** Each order can have multiple order items, and each order item belongs to a single order (One-to-many).
- **Comics and Issue:** Each comic can have multiple issues, and each issue belongs to a single comic (One-to-many).
- **Comics and ComicsDetails:** Each comic can have a single set of detailed information, and each set of detailed information belongs to a single comic (One-to-one).
- **Publishers and ComicsDetails:** Each publisher can have multiple comics, and each comic has a single publisher (one-to-many).
- **Writers and ComicsDetails:** Each writer can have multiple comics, and each comic can have multiple writers (Many-to-many).
- **Issue and ComicsDetails:** Each comic can have multiple issues, and each issue belongs to a single comic (One-to-many).
- **Pricing and ComicsDetails:** Each comic can have multiple pricing, and each pricing information belongs to a single comic issue (One-to-many).
- **Grades and Pricing:** Each grade can have multiple pricing information, and each pricing information belongs to a single grade (One-to-many).
- **Sales_info and Orders:** Each order can have one sales info record and vice versa (One-to-one).

Motivation for the design

Normalization: The design appears to follow standard normalization practices, which can help to prevent data inconsistencies and anomalies. Each entity has its own table, with appropriate primary and foreign keys established to ensure referential integrity.

Flexibility: The design appears to be relatively flexible, with separate tables for each entity that can be expanded or modified as needed. For example, new comics or writers can be added to their respective tables without affecting other parts of the database.

Clarity: The design includes clear and consistent naming conventions for the tables and attributes, which can help to ensure that the data is easily understandable and interpretable by others.

Separation of concerns: The design separates the various entities into their own tables, which can make it easier to manage and maintain the data. For example, pricing information can be stored separately in the pricing table, while order information can be stored in the orders table.

3. Database Structure: A Normalised View

Tables

users: This table stores information about the users of the application, including their usernames, passwords, emails, and the date they joined the application. It is used for user authentication and authorization, as well as for storing user data such as order history and personal information.

comics: This table stores information about the comics available in the application, including their titles, publication years, genres, and types. It is used to display the list of available comics and provide search and filter functionality.

publishers: This table stores information about the publishers of the comics, including their names. It is used to provide additional information about the comics and allow users to search and filter based on the publisher.

writers: This table stores information about the writers of the comics, including their names. It is used to provide additional information about the comics and allow users to search and filter based on the writer.

issue: This table stores information about the issues of the comics, including the issue number and edition. It is used to track the availability of different issues of a comic and allow users to search and filter based on the issue.

comicsDetails: This table stores additional details about the comics, including the publisher, writer, issue, issue name, description, and main character. It is used to provide more detailed information about the comics and allow users to search and filter based on these details.

grades: This table stores information about the condition or grade of the comics, including the grade number and symbol. It is used to allow users to search and filter based on the condition of the comics.

pricing: This table stores information about the pricing of the comics, including the comic, issue, grade, specialty, stock, cost price, and selling price. It is used to track the inventory of the comics and allow users to place orders based on the pricing and availability.

orders: This table stores information about the orders placed by the users, including the user ID, order date, and status. It is used to track the order history and allow users to view their past orders.

orderItems: This table stores information about the items in each order, including the order ID, comic, issue, grade, quantity, and subtotal. It is used to track the details of each order and calculate the total cost.

salesInfo: This table stores information about the sales, including the order ID, subtotal, tax, and total. It is used to track the sales data and calculate the revenue generated by the application.

First Normal Form (1NF)

A relation is in first normal form if all of its attributes contain only atomic values.

users: The users table has a primary key (userID) and all the columns in the table are atomic, meaning that they contain only one piece of information. Therefore, the users table meets the definition of 1NF.

comics: The comics table has a primary key (comicID) and all the columns in the table are atomic, meaning that they contain only one piece of information. Therefore, the comics table meets the definition of 1NF.

publishers: The publishers table has a primary key (publisherID) and all the columns in the table are atomic, meaning that they contain only one piece of information. Therefore, the publishers table meets the definition of 1NF.

writers: The writers table has a primary key (writerID) and all the columns in the table are atomic, meaning that they contain only one piece of information. Therefore, the writers table meets the definition of 1NF.

issue: The issue table has a primary key (issueID) and all the columns in the table are atomic, meaning that they contain only one piece of information. The issueNumber and edition columns are also atomic because they contain only one piece of information each. Therefore, the issue table meets the definition of 1NF.

comicsDetails: The comicsDetails table has a composite primary key consisting of comicID, publisherID, writerID, and issueID, which uniquely identifies each row in the table. All the columns in the table are atomic, meaning that they contain only one piece of information. Therefore, the comicsDetails table meets the definition of 1NF.

grades: The grades table has a primary key (gradeID) and all the columns in the table are atomic, meaning that they contain only one piece of information. Therefore, the grades table meets the definition of 1NF.

pricing: The pricing table has a composite primary key consisting of comicID, issueID, and gradeID, which uniquely identifies each row in the table. All the columns in the table are atomic, meaning that they contain only one piece of information. Therefore, the pricing table meets the definition of 1NF.

orders: The orders table has a primary key (orderID) and all the columns in the table are atomic, meaning that they contain only one piece of information. Therefore, the orders table meets the definition of 1NF.

orderItems: The orderItems table has a primary key (orderItemID). The other columns in the table are all atomic, meaning that they contain only one piece of information. Therefore, the orderItems table meets the definition of 1NF.

salesInfo: The salesInfo table has a primary key (salesID) and all the columns in the table are atomic, meaning that they contain only one piece of information. Therefore, the salesInfo table meets the definition of 1NF.

Second Normal Form (2NF)

In order for the tables to be in 2NF, they have to satisfy the below mentioned criteria.

- The table is in First Normal Form (1NF)
- All non-key columns are fully dependent on the primary key
- There are no partial dependencies, meaning that non-key attributes depend on part of the primary key.

users: This table has a single primary key (userID), which uniquely identifies each row. All columns are atomic and contain a single value. All non-key columns (username, password, email, dateJoined) are fully dependent on the primary key (userID). Therefore, this table is in 2NF.

comics: This table has a single primary key (comicID), which uniquely identifies each row. All columns are atomic and contain a single value. All non-key columns (title, year, genre, type) are fully dependent on the primary key (comicID). Therefore, this table is in 2NF.

publishers: This table has a single primary key (publisherID), which uniquely identifies each row. All columns are atomic and contain a single value. There are no non-key columns in this table. Therefore, this table is in 2NF.

writers: This table has a single primary key (writerID), which uniquely identifies each row. All columns are atomic and contain a single value. There are no non-key columns in this table. Therefore, this table is in 2NF.

issue: This table has a single primary key (issueID), which uniquely identifies each row. All columns are atomic and contain a single value. All non-key columns (comicID, issueNumber, edition) are fully dependent on the primary key (issueID). Therefore, this table is in 2NF.

comicsDetails: This table has a composite primary key (comicID, publisherID, writerID, issueID), which uniquely identifies each row. All columns are atomic and contain a single value. All non-key columns (issueName, description, mainCharacter) are fully dependent on the composite primary key. Therefore, this table is in 2NF.

grades: This table has a single primary key (gradeID), which uniquely identifies each row. All columns are atomic and contain a single value. All non-key columns (gradeNumber, symbol) are fully dependent on the primary key (gradeID). Therefore, this table is in 2NF.

pricing: This table has a composite primary key (comicID, issueID, gradeID), which uniquely identifies each row. All columns are atomic and contain a single value. All non-key columns (speciality, stock, costPrice, price) are fully dependent on the composite primary key. Therefore, this table is in 2NF.

orders: This table has a single primary key (orderID), which uniquely identifies each row. All columns are atomic and contain a single value. All non-key columns (userID, orderDate, status) are fully dependent on the primary key (orderID). Therefore, this table is in 2NF.

orderItems: This table has a composite primary key (orderItemID, orderID, comicID, issueID, gradeID), which uniquely identifies each row. All columns are atomic and contain a single value. All non-key columns (quantity, subtotal) are fully dependent on the composite primary key. Therefore, this table is in 2NF.

salesInfo: This table has a composite primary key (salesID), which uniquely identifies each row. All columns are atomic and contain a single value. All non-key columns (subtotal, tax, orderTotal) are fully dependent on the primary key. Therefore, this table is in 2NF.

Third Normal Form (3 NF)

A relational database is said to be in 3NF (Third Normal Form) if it satisfies the following conditions:

- The table must be in 2NF.
- There must be no transitive functional dependencies.

users: The users table contains only a single candidate key (the userID column), which is also the primary key. There are no repeating groups, and all columns are atomic. Hence, it satisfies 1NF and 2NF. There are no transitive functional dependencies in this table, so it also satisfies 3NF.

comics: The comics table has a single candidate key (the comicID column) which is also the primary key. There are no repeating groups, and all columns are atomic. Hence, it satisfies 1NF and 2NF. There are no transitive functional dependencies in this table, so it also satisfies 3NF.

publishers: The publishers table has a single candidate key (the publisherID column) which is also the primary key. There are no repeating groups, and all columns are atomic. Hence, it satisfies 1NF and 2NF. There are no transitive functional dependencies in this table, so it also satisfies 3NF.

writers: The writers table has a single candidate key (the writerID column) which is also the primary key. There are no repeating groups, and all columns are atomic. Hence, it satisfies 1NF and 2NF. There are no transitive functional dependencies in this table, so it also satisfies 3NF.

issue: The issue table has two columns (comicID and issueNumber) that together form a candidate key. There are no repeating groups, and all columns are atomic. Hence, it satisfies 1NF. The issueID column is a surrogate key that does not have any functional dependency on any other column. The comicID column has a foreign key constraint referencing the comics table. There are no transitive functional dependencies in this table, so it satisfies 2NF and 3NF.

comicsDetails: The comicsDetails table has the following candidate keys: (comicID, issueID), (comicID, publisherID, writerID, mainCharacter). There are no repeating groups, and all columns are atomic. Hence, it satisfies 1NF. The issueName column is dependent on the issueID column, which is part of a candidate key. The description column is dependent on the comicID column, which is part of a candidate key. The mainCharacter column is dependent on the comicID column, which is part of a candidate key. There are no transitive functional dependencies in this table, so it satisfies 2NF and 3NF.

grades: The grades table has a single candidate key (the gradeID column) which is also the primary key. There are no repeating groups, and all columns are atomic. Hence, it satisfies 1NF and 2NF. There are no transitive functional dependencies in this table, so it also satisfies 3NF.

pricing: The table has a single primary key comicID, issueID, gradeID. All the other columns in the table are directly dependent on the primary key. There are no transitive dependencies in the table, meaning that each non-key column is dependent only on the primary key. Therefore, pricing table meets the requirements of 3NF.

orders: The table has a single primary key orderID. All the non-key columns in the table are directly dependent on the primary key. Therefore, orders table meets the requirements of 3NF.

orderItems: The table has a single primary key orderItemID. The columns comicID, issueID, and gradeID are foreign keys referencing other tables, meaning they cannot create a transitive dependency. The columns quantity and subtotal are directly dependent on the primary key orderItemID. Therefore, orderItems table meets the requirements of 3NF.

salesInfo: The table has a single primary key salesID. The columns subtotal, tax, and orderTotal are directly dependent on the primary key salesID. Therefore, salesInfo table meets the requirements of 3NF.

Boyce Codd Normal Form (BCNF)

The BCNF (Boyce-Codd Normal Form) is a database normalization technique that ensures that there are no functional dependencies between non-key attributes in a relation.

A relation schema R is in BCNF with respect to a set F of functional dependencies if, for all functional dependencies in F+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency (i.e., $\beta \subseteq \alpha$).
- α is a superkey for schema R (1, *Database System Concepts" Seventh Edition by Abraham Silberschatz, Henry Korth, and S. Sudarshan, Page - 313*).

users: The users table has a single primary key attribute (userID), which is used to uniquely identify each record. The other attributes (username, password, email, dateJoined) are dependent on the userID and have no functional dependencies on each other. Hence, it satisfies BCNF.

comics: The comics table has a single primary key attribute (comicID), which is used to uniquely identify each record. The other attributes (title, year, genre, type) are dependent on the comicID and have no functional dependencies on each other. Hence, it satisfies BCNF.

publishers and writers: The publishers and writers tables both have a single primary key attribute (publisherID and writerID, respectively), which is used to uniquely identify each record. They contain no other attributes, and hence they satisfy BCNF.

issue: The issue table has a primary key attribute (issueID) and a foreign key attribute (comicID), which references the comics table. The other attributes (issueNumber, edition) are

dependent on the issueID and have no functional dependencies on each other. Hence, it satisfies BCNF.

comicsDetails: The comicsDetails table has multiple foreign key attributes (comicID, publisherID, writerID, issueID) that reference the corresponding primary keys in their respective tables. The other attributes (issueName, description, mainCharacter) are dependent on the comicID, issueID, and writerID and have no functional dependencies on each other. Hence, it satisfies BCNF.

grades: The grades table has a single primary key attribute (gradeID), which is used to uniquely identify each record. The other attributes (gradeNumber, symbol) are dependent on the gradeID and have no functional dependencies on each other. Hence, it satisfies BCNF.

pricing: The pricing table has multiple foreign key attributes (comicID, issueID, gradeID) that reference the corresponding primary keys in their respective tables. The other attributes (speciality, stock, costPrice, price) are dependent on the comicID, issueID, and gradeID and have no functional dependencies on each other. Hence, it satisfies BCNF.

orders: The orders table has a single primary key attribute (orderID), which is used to uniquely identify each record. The other attributes (userID, orderDate, status) are dependent on the orderID and have no functional dependencies on each other. Hence, it satisfies BCNF.

orderItems: The orderItems table has multiple foreign key attributes (orderID, comicID, issueID, gradeID) that reference the corresponding primary keys in their respective tables. The other attributes (quantity, subtotal) are dependent on the orderID, comicID, issueID, and gradeID and have no functional dependencies on each other. Hence, it satisfies BCNF.

salesInfo: The salesInfo table has a primary key attribute (salesID) and a foreign key attribute (orderID), which references the orders table. The other attributes (subtotal, tax, orderTotal) are dependent on the orderID and have no functional dependencies on each other. Hence, it satisfies BCNF.

Why BCNF is necessary?

Update anomalies: If the tables are not in BCNF, it can lead to update anomalies, which means that updating a record can result in inconsistent data. For example, if a publisher changes their name, and the publisher name is stored in multiple tables, each occurrence of the name will need to be updated. Failure to do so will result in inconsistency in the data.

Insertion anomalies: If the tables are not in BCNF, it can lead to insertion anomalies, which means that it may not be possible to insert certain data without inserting other data at the

same time. For example, if the comicsDetails table has a foreign key reference to the issue table, it may not be possible to insert a record into comicsDetails unless there is already a record in the issue table with the same issue ID.

Deletion anomalies: If the tables are not in BCNF, it can lead to deletion anomalies, which means that deleting a record can result in the loss of other data that should not have been deleted. For example, if a writer who wrote several comics is deleted from the writers table, all of the comics that the writer wrote will also be deleted unless some mechanism is put in place to prevent this.

In summary, the above tables need to be in BCNF to ensure that the data is consistent, accurate, and without anomalies. This helps to avoid problems such as data redundancy, update, insertion, and deletion anomalies, which can result in data inconsistencies and lead to serious errors and issues.

4. Database Views

1. View to fetch all the comics and display the prices of all the grades.

```
CREATE OR REPLACE VIEW all_comics AS
SELECT c.comicID,c.title,c.year,i.issueNumber,g.symbol,CONCAT('$',p.price) AS price
FROM comics c
INNER JOIN pricing p on c.comicID=p.comicID
INNER JOIN issue i on p.comicID=i.comicID and i.issueID=p.issueID
INNER JOIN grades g on g.gradeID=p.gradeID
ORDER BY c.comicID;
```

The above SQL statement creates a view named all_comics which selects data from multiple tables: comics, pricing, issue, and grades. The all_comics view combines the data from these tables to provide a more comprehensive and readable view of the information related to the comics available in the database. It includes the comic ID, title, year, issue number, grade symbol, and price.

By creating this view, users of the database can easily access the important details of the comics, without the need to join multiple tables or perform complex queries. The view provides a simple and efficient way to retrieve the desired information, making it easier for users to interact with the database and making the data more accessible and user-friendly. The all_comics view adds value to the database design by improving the usability and accessibility of the data related to the comics available in the database.

The reason this is defined as an SQL view rather than a table in its own right is that the information presented in the view is based on data from multiple tables. Rather than duplicating this information in a new table, which would require constant updating to keep it synchronized with the original data, a view can be created that combines the necessary information from the existing tables in a logical and organized manner. This makes it easier to manage and update the data, and also allows for greater flexibility in terms of customizing the view to meet specific needs. Additionally, views can be used to limit access to certain data or to present data in a way that is more easily understood by users who may not have direct access to the underlying database structure.

comicID	title	year	issueNumber	symbol	price
1	007 JAMES BOND FOR YOUR EYES ONLY PB	1981	1	FAIR (FR)	\$18.75
1	007 JAMES BOND FOR YOUR EYES ONLY PB	1981	1	GOOD (GD)	\$37.50
1	007 JAMES BOND FOR YOUR EYES ONLY PB	1981	1	VERY GOOD (VG)	\$56.25
1	007 JAMES BOND FOR YOUR EYES ONLY PB	1981	1	FINE (FN)	\$75.00
1	007 JAMES BOND FOR YOUR EYES ONLY PB	1981	1	VERY FINE (VF)	\$85.00
1	007 JAMES BOND FOR YOUR EYES ONLY PB	1981	1	NEAR MINT (NM)	\$93.75
1	007 JAMES BOND FOR YOUR EYES ONLY PB	1981	1	NEAR MINT (NM)	\$93.75
2	SPIDER MAN	1963	1	FINE (FN)	\$250000.00
2	SPIDER MAN	1963	1	NEAR MINT (NM)	\$162.50
2	SPIDER MAN	1963	2	GOOD (GD)	\$10400.00
2	SPIDER MAN	1963	3	VERY FINE (VF)	\$122.50
2	SPIDER MAN	1963	3	NEAR MINT (NM)	\$162.50
2	SPIDER MAN	1963	4	VERY GOOD (VG)	\$13000.00
3	AVENGERS	2016	1	VERY FINE (VF)	\$66.25
3	AVENGERS	2016	2	NEAR MINT (NM)	\$116.25
3	AVENGERS	2016	3	VERY FINE (VF)	\$82.50
3	AVENGERS	2016	3	NEAR MINT (NM)	\$100.00
3	AVENGERS	2016	4	NEAR MINT (NM)	\$33.75
4	FANTASTIC FOUR	2013	1	GOOD (GD)	\$5.00
4	FANTASTIC FOUR	2013	1	VERY GOOD (VG)	\$6.25
4	FANTASTIC FOUR	2013	1	FINE (FN)	\$7.81
4	FANTASTIC FOUR	2013	1	VERY FINE (VF)	\$11.25
4	FANTASTIC FOUR	2013	1	NEAR MINT (NM)	\$15.00
4	FANTASTIC FOUR	2013	2	NEAR MINT (NM)	\$170.00
4	FANTASTIC FOUR	2013	3	GOOD (GD)	\$5.31
4	FANTASTIC FOUR	2013	3	VERY GOOD (VG)	\$8.13

Figure 2: All Comics

2. View to fetch all the Marvel Comics

```
CREATE OR REPLACE VIEW marvel_comics AS
SELECT DISTINCT a.title,a.year
FROM all_comics a
JOIN comicsDetails cd on cd.comicID=a.comicID
JOIN publishers p ON p.publisherID=cd.publisherID
WHERE p.name='Marvel Comics'
ORDER BY a.title;
```

The "marvel_comics" view is designed to extract information from the "all_comics" view, specifically to display all titles and years of the comics that are published by Marvel Comics. It does this by joining the "all_comics" view with the "comicsDetails" table on the comic ID and then joining the result with the "publishers" table on the publisher ID to filter the results based on the publisher name. Finally, the results are sorted alphabetically by title.

The value this view adds to the database design is that it provides a convenient and efficient way to extract information about Marvel Comics titles from the existing data in the database without having to manually write complex SQL queries every time this information is needed. This view saves time and effort for anyone who needs to access this information regularly.

The "marvel_comics" view is defined as an SQL view rather than a table in its own right because it is based on data that already exists in the database, specifically the "all_comics" view and the "comicsDetails" and "publishers" tables. Defining the view as an SQL view allows for a more flexible and dynamic way of querying the data, as any changes made to the underlying tables will automatically be reflected in the view. Additionally, defining the view as a separate table would require duplicating data that already exists in the database, which can lead to data inconsistencies and maintenance issues.

title	year
007 JAMES BOND FOR YOUR EYES ONLY PB	1981
ALL NEW X-MEN	2015
AVENGERS	2016
FANTASTIC FOUR	2013
MADROX	2005
SPIDER MAN	1963

Figure 3: Marvel Comics

3. View to fetch all the DC Comics

```
CREATE OR REPLACE VIEW dc_comics AS
SELECT DISTINCT a.title,a.year
FROM all_comics a
JOIN comicsDetails cd on cd.comicID=a.comicID
JOIN publishers p ON p.publisherID=cd.publisherID
WHERE p.name='DC Comics'
ORDER BY a.title;
```

This view, named "dc_comics", retrieves the distinct titles and years of all comics published by DC Comics from the "all_comics" view. The view is created using joins between "all_comics" and "comicsDetails" on the comic ID, and between "comicsDetails" and "publishers" on the publisher ID. The WHERE clause is used to filter the results to include only those comics with a publisher name of "DC Comics". The results are sorted in alphabetical order by title.

The "dc_comics" view adds value to the database design by providing a convenient way to retrieve a list of all comics published by DC Comics. This is useful for users who are specifically interested in comics published by DC Comics and want to quickly and easily view a list of all such comics in the database. The view eliminates the need for users to manually write complex SQL queries that involve joins and filtering on multiple tables.

The reason why this relation is defined as an SQL view rather than a table in its own right is that the view is simply a subset of data that is already available in the "all_comics" view. By creating a view instead of a table, the database designer can avoid duplicating data and can ensure that the view always reflects the current state of the underlying data in the database. Additionally, views are typically used to provide a simplified and/or customized view of the data in a database, which is precisely the case with the "dc_comics" view.

title	year
BATMAN ADVENTURES	2003
FLASH	2011
JUSTICE LEAGUE OF AMERICA	2006
SUPERMAN: FOR THE ANIMALS	2000
WONDER WOMAN	1942

Figure 4: DC Comics

4. View to display all the products ordered by a particular user

```
CREATE OR REPLACE VIEW orders_by_user AS
SELECT DISTINCT
o.userID,u.username,o.orderID,c.title,c.year,i.issueNumber,oi.quantity,oi.subtotal
FROM orders o
JOIN orderItems oi ON o.orderID = oi.orderID
JOIN pricing p ON oi.comicID = p.comicID AND oi.gradeID = p.gradeID AND oi.issueID =
p.issueID
JOIN users u ON u.userID = o.userID
JOIN comics c ON c.comicID = oi.comicID
JOIN issue i on p.comicID=i.comicID and i.issueID=p.issueID
ORDER BY u.username;
```

The view "orders_by_user" provides a way to retrieve information about orders made by users, including the user ID, username, order ID, comic title, year of publication, issue number, quantity ordered, and subtotal. By joining several tables in the database, this view provides a consolidated view of all the relevant information related to user orders.

This view can be valuable for analyzing sales trends and customer behavior, as it allows for the tracking of which comics are popular among different users and the quantity ordered. It can also help identify high-spending users or those who have placed multiple orders.

Defining this information as a view rather than a table allows for easy retrieval of up-to-date order information, without the need for constantly updating a separate table. The view

definition is based on existing tables in the database, so any changes made to those tables will automatically be reflected in the view results.

userID	username	orderID	title	year	issueNumber	quantity	subtotal
2	davidlee	2	SPIDER MAN	1963	2	3	31200.00
2	davidlee	5	AVENGERS	2016	3	2	200.00
2	davidlee	2	FANTASTIC FOUR	2013	1	1	6.25
2	davidlee	5	FANTASTIC FOUR	2013	1	1	15.00
2	davidlee	2	FLASH	2011	2	2	32.50
2	davidlee	6	MADROX	2005	1	5	225.00
1	emilybrown	1	007 JAMES BOND FOR YOUR EYES ONLY PB	1981	1	2	35.00
1	emilybrown	4	007 JAMES BOND FOR YOUR EYES ONLY PB	1981	1	1	85.00
1	emilybrown	1	AVENGERS	2016	1	4	265.00
5	katewilson	8	BATMAN ADVENTURES	2003	4	2	87.50
5	katewilson	8	FLASH	2011	3	1	16.88
5	katewilson	8	JUSTICE LEAGUE OF AMERICA	2006	2	4	40.00
4	michaeladams	7	SPIDER MAN	1963	2	3	31200.00
4	michaeladams	7	FANTASTIC FOUR	2013	1	1	6.25
4	michaeladams	7	WONDER WOMAN	1942	102	3	3217.50
3	sarahjohnson	3	AVENGERS	2016	2	1	116.25
3	sarahjohnson	3	FANTASTIC FOUR	2013	2	3	510.00
3	sarahjohnson	3	BATMAN ADVENTURES	2003	2	2	80.00
3	sarahjohnson	3	SUPERMAN: FOR THE ANIMALS	2000	1	4	30.00

Figure 5: Order by user

5. View to display the top 5 best-selling comics.

```
CREATE OR REPLACE VIEW top_selling AS
SELECT oi.comicID,c.title, SUM(oi.quantity) AS total_sold
FROM orderItems oi
JOIN comics c ON c.comicID = oi.comicID
GROUP BY oi.comicID
ORDER BY total_sold DESC
LIMIT 5;
```

The view `top_selling` is designed to provide valuable information on the top-selling comics in the database. It selects the `comicID`, `title`, and total number of units sold for the top 5 best-selling comics by aggregating data from the `orderItems` and `comics` tables. This view allows the management team to quickly identify the most popular comics in the inventory.

This view adds value to the database design by providing insight into the most in-demand comics. It can help in several ways, including:

Inventory Management: It provides valuable information on which comics are most popular and should be stocked up on, and which comics may not be selling as well.

Sales Analysis: It helps in analyzing the sales trends of the most popular comics and identifying any changes in demand over time.

Marketing: It can help in marketing decisions such as identifying which comics to promote more aggressively or which comics to offer discounts or promotions on.

This view is defined as an SQL view and not a table in its own right because it aggregates data from existing tables (`orderItems` and `comics`) rather than creating new data. Defining it as a

view allows for easy access to this valuable information without duplicating data in the database.

comicID	title	total_sold
3	AVENGERS	7
2	SPIDER MAN	6
4	FANTASTIC FOUR	6
11	MADROX	5
10	JUSTICE LEAGUE OF AMERICA	4

Figure 6: Top 5 Best Selling Comics

6. View to display most popular comic book writers based on number of issues sold.

```
CREATE OR REPLACE VIEW popular_comic_book_writers AS
SELECT w.name, c.comicID, c.title, SUM(oi.quantity) as total_sold
FROM comics c
JOIN orderItems oi ON c.comicID = oi.comicID
JOIN comicsDetails cd ON cd.comicID = oi.comicID
JOIN writers w ON w.writerID = cd.writerID
GROUP BY w.name, c.comicID, c.title
ORDER BY total_sold DESC;
```

The `popular_comic_book_writers` view provides information on the popular comic book writers based on the total number of comic books sold that were written by them. This view joins the `comics`, `orderItems`, `comicsDetails`, and `writers` tables to get the total quantity of each comic book sold and the name of the writer who wrote it. It then groups the results by writer name, comic ID, and comic title, and sorts the results in descending order based on the total number of comic books sold by the writer.

This view adds value to the database design as it provides insights into the popularity of comic book writers and the demand for their work. This information can be used by the publishers to make decisions regarding future publications and marketing strategies. Additionally, it can be used by comic book enthusiasts to explore the work of popular writers and to make informed purchasing decisions.

Defining this relation as an SQL view rather than a table in its own right has a few advantages. Firstly, creating a view allows the query logic to be encapsulated and reused easily across different applications. Secondly, a view provides a dynamic representation of the underlying data, allowing for changes to be made to the underlying tables without affecting the view's

output. This means that the view can remain current and accurate even as the underlying data changes. Finally, by using a view, the query logic and data retrieval can be optimized, which can lead to better performance compared to running the same query on the underlying tables directly.

name	comicID	title	total_sold
Mark Waid	3	AVENGERS	28
Stan Lee	2	SPIDER MAN	24
Matt Fraction	4	FANTASTIC FOUR	24
Dan Slott	6	BATMAN ADVENTURES	16
Francis Manapul	7	FLASH	12
Brad Meltzer	10	JUSTICE LEAGUE OF AMERICA	12
Robert Kanigher	9	WONDER WOMAN	9
Peter Allen David	11	MADROX	5
Mark Millar	8	SUPERMAN: FOR THE ANIMALS	4
Ian Fleming	1	007 JAMES BOND FOR YOUR EYES ONLY PB	3
Mark Waid	1	007 JAMES BOND FOR YOUR EYES ONLY PB	3

Figure 7: Popular Comic Book Writers

5. Procedural Elements

Triggers

1. Trigger to compute the selling price of the comics based on the cost price. The selling price is set to be 125% of the cost price.

```
CREATE TRIGGER set_selling_price
BEFORE INSERT ON pricing
FOR EACH ROW
BEGIN
    IF NEW.price IS NULL THEN
        SET NEW.price = NEW.costPrice * 1.25;
    END IF;
END;
```

A trigger is a special type of stored procedure that automatically executes in response to certain events or changes to data in a database. In this case, the trigger is set to execute before a new row is inserted into the "pricing" table.

The purpose of this trigger is to ensure that a selling price is always set for new entries in the "pricing" table. If a new row is inserted into the "pricing" table and the "price" column is left blank, the trigger will automatically set the selling price to be 25% higher than the cost price.

This trigger helps to enforce data consistency and accuracy, as it ensures that all pricing information in the database includes a selling price. It also simplifies the data entry process for users, as they do not have to manually calculate the selling price for each entry they create.

Overall, the use of this trigger helps to maintain the integrity of the pricing data in the database and streamline the data entry process for users.

comicID	issueID	gradeID	speciality	stock	costPrice	price
1	1	56	No	100	15.00	18.75
1	1	47	No	90	30.00	37.50
1	1	35	No	75	45.00	56.25
1	1	28	No	86	60.00	75.00
1	1	21	No	105	68.00	85.00
1	1	4	No	25	75.00	93.75
1	1	4	No	5	75.00	93.75
2	2	23	No	10	200000.00	250000.00
2	2	4	No	14	130.00	162.50
2	3	47	No	1	8320.00	10400.00
2	4	12	Yes	56	98.00	122.50
2	4	4	No	72	130.00	162.50
2	5	37	Yes	91	10400.00	13000.00
3	6	21	No	55	53.00	66.25
3	7	4	No	48	93.00	116.25
3	8	11	Yes	7	66.00	82.50
3	8	5	No	19	80.00	100.00
3	9	9	No	28	27.00	33.75
4	10	47	No	33	4.00	5.00
4	10	35	No	41	5.00	6.25
4	10	23	No	53	6.25	7.81
4	10	11	Yes	12	9.00	11.25

Figure 8: Set Selling Price

2. Trigger to update the selling price when the cost price is updated.

```

CREATE TRIGGER update_selling_price
BEFORE UPDATE ON pricing
FOR EACH ROW
BEGIN
    IF NEW.costPrice <> OLD.costPrice THEN
        SET NEW.price = NEW.costPrice * 1.25;
    END IF;
END;
```

The trigger update_selling_price is designed to automatically update the selling price of a comic book when the cost price of that comic book changes. This trigger is important because it ensures that the selling price is always calculated correctly and that the price is updated in

a timely manner. Without this trigger, the system would rely on manual updates, which could be time-consuming and error-prone.

The trigger is defined as a BEFORE UPDATE trigger, which means that it is executed before any updates are made to the pricing table. The trigger is also defined as a FOR EACH ROW trigger, which means that it is executed once for each row that is being updated.

The trigger checks if the costPrice value of the new row is different from the costPrice value of the old row. If there is a difference, then the trigger calculates the new selling price by multiplying the new costPrice value by 1.25. This calculation is based on the assumption that the selling price is always set at 25% above the cost price.

By using this trigger, the system can ensure that the selling price is always updated correctly and that all necessary calculations are done automatically. This helps to reduce errors and increase efficiency, ultimately providing a better user experience for the customers and employees who interact with the database.

comicID	issueID	gradeID	speciality	stock	costPrice	price
1	1	56	No	100	15.00	18.75
1	1	47	No	98	38.00	47.50
1	1	35	No	75	45.00	56.25
1	1	28	No	86	60.00	75.00
1	1	21	No	105	68.00	85.00

Figure 9: Update Selling Price - Before

comicID	issueID	gradeID	speciality	stock	costPrice	price
1	1	56	No	98	14.00	17.50
1	1	47	No	98	38.00	47.50
1	1	35	No	75	45.00	56.25
1	1	28	No	86	60.00	75.00
1	1	21	No	104	68.00	85.00
1	1	4	No	25	75.00	93.75

Figure 10: Update Selling Price - After

Procedures and Functions

1. Procedure to place an order which inserts data into orders, orderItems, salesInfo table and updates stock information.

```
CREATE PROCEDURE place_order(
```

```
IN user_id INT,
IN comic_ids VARCHAR(100),
IN issue_ids VARCHAR(100),
```

Project Excelsior

```
IN grade_ids VARCHAR(100),
IN quantities VARCHAR(100)
)
BEGIN
  DECLARE price_1 DECIMAL(10,2);
  DECLARE tax DECIMAL(10,2);
  DECLARE total_cost DECIMAL(10,2);
  DECLARE sub_total DECIMAL(10,2);
  DECLARE tax_sales DECIMAL(10,2);
  DECLARE total_sub DECIMAL(10,2);
  -- Insert the order
  INSERT INTO orders (userID,orderDate,status) VALUES (user_id, NOW(), 'new');
  SET @order_id = LAST_INSERT_ID();
  SET tax=0;
  SET total_cost=0;
  SET tax_sales=0;
  SET total_sub=0;

  SET @comic_ids = comic_ids;
  SET @issue_ids = issue_ids;
  SET @grade_ids = grade_ids;
  SET @quantities = quantities;

  WHILE (LENGTH(@comic_ids) > 0 AND LENGTH(@issue_ids) > 0 AND LENGTH(@grade_ids)
> 0 AND LENGTH(@quantities) > 0) DO
    SET @comma_pos = INSTR(@comic_ids, ',');
    SET @comic_id = IF(@comma_pos = 0, @comic_ids, SUBSTR(@comic_ids, 1,
@comma_pos - 1));
    SET @comic_ids = IF(@comma_pos = 0, '', SUBSTR(@comic_ids, @comma_pos + 1));

    SET @comma_pos = INSTR(@issue_ids, ',');
    SET @issue_id = IF(@comma_pos = 0, @issue_ids, SUBSTR(@issue_ids, 1, @comma_pos -
1));
    SET @issue_ids = IF(@comma_pos = 0, '', SUBSTR(@issue_ids, @comma_pos + 1));

    SET @comma_pos = INSTR(@grade_ids, ',');
    SET @grade_id = IF(@comma_pos = 0, @grade_ids, SUBSTR(@grade_ids, 1,
@comma_pos - 1));
    SET @grade_ids = IF(@comma_pos = 0, '', SUBSTR(@grade_ids, @comma_pos + 1));

    SET @comma_pos = INSTR(@quantities, ',');
```

Project Excelsior

```
SET @quantity = IF(@comma_pos = 0, @quantities, SUBSTR(@quantities, 1,
@comma_pos - 1));
SET @quantities = IF(@comma_pos = 0, "", SUBSTR(@quantities, @comma_pos + 1));

INSERT INTO orderItems (orderId,comicID,issueID,gradeID,quantity) VALUES
(@order_id,@comic_id,@issue_id,@grade_id,@quantity);

SELECT price INTO price_1 FROM pricing WHERE comicID = @comic_id AND issueID =
@issue_id AND gradeID = @grade_id;

UPDATE orderItems
SET subtotal = @quantity * price_1
WHERE comicID = @comic_id and gradeID=@grade_id and issueID = @issue_id;

SELECT subtotal INTO sub_total FROM orderItems WHERE orderId = @order_id and
gradeID=@grade_id and issueID = @issue_id;

UPDATE pricing
SET stock = stock - @quantity
WHERE comicID = @comic_id and gradeID=@grade_id and issueID = @issue_id;
SET tax = sub_total * 0.1;
SET tax_sales = tax_sales+sub_total * 0.1;

SET total_cost = (total_cost)+(sub_total + tax);
SET total_sub = total_sub+@quantity*price_1;
END WHILE;
SELECT CONCAT("Order ID: ",@order_id," Order Total: ", total_cost) AS message;
INSERT INTO salesInfo(orderID,subtotal,tax,orderTotal) VALUES
(@order_id,total_sub,tax_sales,total_cost);
UPDATE orderItems SET orderId = @order_id WHERE orderId IS NULL;
END;
```

The "place_order" stored procedure is used in a database to perform a sequence of operations that occur when a customer places an order for comics. The procedure accepts several input parameters such as user_id, comic_ids, issue_ids, grade_ids, and quantities. The procedure begins by inserting a new order record in the "orders" table and setting the @order_id variable to the ID of the newly inserted order.

The procedure then uses a WHILE loop to iterate through the comic, issue, grade, and quantity parameters. For each iteration, it extracts the individual comic, issue, grade, and quantity values and inserts them into the "orderItems" table with the corresponding @order_id.

The procedure then queries the "pricing" table to retrieve the price of the comic for the given issue and grade. It calculates the subtotal for the order item by multiplying the quantity with the price. It also updates the stock value in the "pricing" table by subtracting the quantity ordered.

Next, the procedure calculates the tax value for the order item by multiplying the subtotal by 0.1 and adds it to the tax_sales variable which stores the total tax amount for the order. It also updates the total_cost variable by adding the subtotal and tax for the current order item.

The procedure continues the WHILE loop until all order items have been processed. It then returns a message that includes the order ID and the total cost of the order. Finally, it inserts a new record into the "salesInfo" table with the order details such as subtotal, tax, and orderTotal.

In summary, this stored procedure streamlines the process of placing an order for comics by automating several operations and simplifying the code required to perform these tasks. It also helps to ensure data integrity by updating the necessary tables in a transactional manner.

```

+-----+
| message |
+-----+
| Order ID: 1 Order Total: 343.07 |
+-----+
1 row in set (0.01 sec)

Query OK, 0 rows affected (0.02 sec)

+-----+
| message |
+-----+
| Order ID: 2 Order Total: 34362.63 |
+-----+

```

Figure 11: Result after placing an order

orderID	userID	orderDate	status
1	1	2023-05-07 18:45:39	new
2	2	2023-05-07 18:45:39	new

Figure 12: Data inserted into orders table

orderItemID	orderID	comicID	issueID	gradeID	quantity	subtotal
1	1	1	1	56	2	46.88
2	1	3	6	21	4	265.00
3	2	2	3	47	3	31200.00
4	2	4	10	35	1	6.25
5	2	7	23	5	2	32.50

Figure 13: Data inserted into orderItems table

salesID	orderID	subtotal	tax	orderTotal
1	1	311.88	31.19	343.07
2	2	31238.75	3123.88	34362.63

Figure 14: Data inserted into salesInfo table

2. Procedure to update the order status in the orders table.

```

CREATE PROCEDURE update_order (
  IN order_id INT,
  IN new_status VARCHAR(20)
)
BEGIN
  DECLARE order_status VARCHAR(20);
  SELECT status INTO order_status FROM orders WHERE orderID = order_id;
  IF order_status = 'new' AND new_status='processed' THEN
    UPDATE orders SET status = new_status WHERE orderID = order_id;
    SELECT CONCAT("Order ",order_id, " status updated sucessfully.") AS message_1;
  ELSEIF order_status = 'processed' AND new_status='shipped' THEN
    UPDATE orders SET status = new_status WHERE orderID = order_id;
    SELECT CONCAT("Order ",order_id, " status updated sucessfully.") AS message_1;
  ELSEIF order_status = 'shipped' AND new_status='delivered' THEN
    UPDATE orders SET status = new_status WHERE orderID = order_id;
    SELECT CONCAT("Order ",order_id, " status updated sucessfully.") AS message_1;
  ELSE
    SELECT CONCAT('Order ', order_id, ' is already processed, cannot revert back.') AS
message_2;
  END IF;
END;

```

The procedure starts by declaring a variable "order_status" of type VARCHAR(20) to hold the current status of the order. It then queries the "orders" table to retrieve the current status of the order with the specified order_id and stores it in the "order_status" variable.

Next, the procedure uses a series of conditional statements (IF-ELSEIF-ELSE) to determine which status transition is valid based on the current status and the new status provided. If the transition is valid, the procedure updates the "status" column of the "orders" table with the new status for the specified order_id.

If the transition is not valid, the procedure returns an error message to indicate that the order cannot be updated to the new status. The message includes the order_id and the appropriate error message.

This stored procedure can be useful in managing the status of orders within a database application. It simplifies the process of updating an order's status by allowing the application to call the procedure and pass the necessary parameters, rather than having to write the update query every time. The procedure also helps to ensure that the status transitions are valid and prevent the orders from being updated incorrectly.

message_1
Order 1 status updated sucessfully.

Figure 15: After status updated to processed

orderID	userID	orderDate	status
1	1	2023-05-07 19:37:04	processed
2	2	2023-05-07 19:37:04	new

Figure 16: Status updated in orders table

orderID	userID	orderDate	status
1	1	2023-05-07 19:37:04	shipped
2	2	2023-05-07 19:37:04	new

Figure 17: After status updated to shipped

orderID	userID	orderDate	status
1	1	2023-05-07 19:37:04	delivered
2	2	2023-05-07 19:37:04	new

Figure 18: After status updated to delivered

Functions

1. Function to cancel the order, update the status and update the stock.

```
CREATE FUNCTION cancel_order(order_id INT) RETURNS VARCHAR(20)
READS SQL DATA
DETERMINISTIC
BEGIN
    DECLARE order_status VARCHAR(20);
    SELECT status INTO order_status FROM orders WHERE orderID = order_id;
    IF order_status = 'cancelled' THEN
        RETURN "Failed";
    ELSEIF order_status = 'shipped' THEN
        RETURN "Failed";
    ELSEIF order_status = 'delivered' THEN
        RETURN "Failed";
    ELSE
        UPDATE orders SET status = 'cancelled' WHERE orderID = order_id;
```

```

DELETE FROM salesInfo WHERE orderID=order_id;
UPDATE pricing p
JOIN orderItems oi ON p.comicID = oi.comicID
SET p.stock = p.stock + oi.quantity
WHERE oi.orderID = order_id;
RETURN "Success";
END IF;
END;

```

This is a function called "cancel_order" that is used in a database to cancel an order. The function takes an input parameter of "order_id" which is used to identify the order that needs to be cancelled. The function returns a VARCHAR value which indicates whether the cancellation was successful or not.

The function first declares a variable "order_status" to store the status of the order identified by the input parameter. It then queries the "orders" table to retrieve the current status of the order with the given "order_id".

If the order is already cancelled, shipped or delivered, the function returns "Failed". If the order is not in any of those states, it proceeds to update the "orders" table by setting the status of the order to "cancelled".

The function then deletes the corresponding sales information from the "salesInfo" table and updates the stock in the "pricing" table by adding the quantity of the canceled items back to the available stock.

Finally, the function returns "Success" to indicate that the order has been successfully canceled.

This function is useful in the database as it provides a way to cancel an order while ensuring that the inventory is updated accordingly. The function also ensures that an order can only be canceled if it is not already shipped or delivered, which helps to maintain the integrity of the data in the database.

cancel_order(1)
Failed

Figure 19: Cancel failed since order 1 is delivered already

orderID	userID	orderDate	status
1	1	2023-05-07 19:37:04	delivered
2	2	2023-05-07 19:37:04	cancelled

Figure 20: Orders table after cancelling order 2

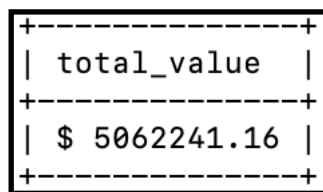
6. Example Queries : Database in Action

1. Query to get the total value of the inventory (comics).

```
SELECT CONCAT("$ ",SUM(stock * price)) as total_value  
FROM pricing;
```

This SQL query calculates the total value of the inventory in the database by multiplying the stock and price of each item in the "pricing" table and then summing them up.

The query can be useful for the application in several ways. For example, it can help the business to keep track of the total value of their inventory, which can help them make informed decisions about their stock levels and pricing strategies. It can also help them to identify trends in their inventory value over time, which can inform forecasting and budgeting decisions. Additionally, it can be used to calculate the cost of goods sold (COGS) for accounting purposes, which can be important for tax reporting and financial statements.



total_value
\$ 5062241.16

Figure 21: Total Value of Inventory

2. Query to get all the products that are out of stock.

```
SELECT p.comicID, p.issueID,c.title,p.gradeID  
FROM pricing p  
LEFT JOIN comics c ON p.comicID = c.comicID  
WHERE p.stock IS NULL OR p.stock = 0;
```

The query starts by specifying the columns to be returned: comicID, issueID, title, and gradeID. The SELECT statement includes the pricing (p) and comics (c) tables, joined on the comicID field. The LEFT JOIN ensures that all records from the pricing table are included in the results, even if there are no matching records in the comics table.

The WHERE clause filters the results to only include records where the stock field is null or 0. This is useful for identifying which comics are out of stock or have not been priced properly in the database. By joining with the comics table, it's also possible to see the title and other relevant details for each comic.

This query is useful for the application because it can help identify issues with the inventory management system, allowing the business to quickly take action to ensure that the proper inventory levels are maintained, and the pricing is correct for all the comic books.

comicID	issueID	title	gradeID
5	14	ALL NEW X-MEN	35
10	30	JUSTICE LEAGUE OF AMERICA	5

Figure 22: Comics out of stock

3. Query to list all the comics sold in the last 30 days.

```
SELECT c.title,oi.comicID,oi.gradeID,SUM(oi.quantity) as total_sold
FROM orderItems oi
JOIN comics c ON oi.comicID = c.comicID
JOIN orders o ON o.orderID = oi.orderID
WHERE o.orderDate >= DATE_SUB(NOW(), INTERVAL 30 DAY)
GROUP BY c.comicID,oi.gradeID
ORDER BY total_sold DESC
LIMIT 10;
```

This query retrieves the top 10 best-selling comic books in the last 30 days. It does this by joining the orderItems table with the comics and orders tables to get the relevant information. The GROUP BY clause groups the results by comic ID and grade ID, and the SUM function calculates the total quantity sold for each comic. The results are then ordered in descending order by total quantity sold, and the top 10 results are returned using the LIMIT clause.

This query is useful for the application as it provides valuable insights into the most popular comic books, allowing the business to make informed decisions on which comics to order more of, which to stock less of, and which to potentially promote to customers. By monitoring the top-selling comics, the business can better cater to the interests and demands of its customers, ultimately improving sales and customer satisfaction.

title	comicID	gradeID	total_sold
SPIDER MAN	2	47	6
MADROX	11	39	5
AVENGERS	3	21	4
JUSTICE LEAGUE OF AMERICA	10	38	4
SUPERMAN: FOR THE ANIMALS	8	30	4
WONDER WOMAN	9	47	3
FANTASTIC FOUR	4	5	3
007 JAMES BOND FOR YOUR EYES ONLY PB	1	56	2
FLASH	7	5	2
BATMAN ADVENTURES	6	7	2

Figure 23: Comics sold in last 30 days

4. Query to fetch Spider man comics where issue number is less than 150 and quality >=mint and price less than 200.00.

```

SELECT c.title, c.year, i.issueNumber, g.symbol, CONCAT('$',p.price) as price
FROM comics c
JOIN pricing p on p.comicID = c.comicID
JOIN issue i on i.issueID = p.issueID AND c.comicID = i.comicID
JOIN grades g on p.gradeID = g.gradeID
WHERE c.title = 'SPIDER MAN'
AND i.issueNumber < 150
AND g.gradeNumber <= (SELECT min(gradeNumber) from grades where symbol='MINT (MT)')
AND p.price <= 200.00;

```

This query retrieves information about comics with the title "SPIDER MAN", issue number less than 150, graded as "MINT (MT)" or lower, and priced at \$200 or less.

The SELECT statement specifies the columns to be returned, including the comic title, issue year, issue number, grade symbol, and price.

The FROM clause indicates the tables being used, including the comics, pricing, issue, and grades tables, and uses JOINS to connect them based on the appropriate keys.

The WHERE clause filters the results to only include comics with the specified criteria. The comic title must be "SPIDER MAN", the issue number must be less than 150, the grade symbol must be "MINT (MT)" or lower, and the price must be \$200 or less.

The purpose of this query is to help customers find comics that meet their specific criteria, based on title, issue number, grade, and price. It can be used by collectors who are looking for particular issues or grades of a comic, or by customers who want to stay within a certain budget.

title	year	issueNumber	symbol	price
SPIDER MAN	1963	1	NEAR MINT (NM)	\$ 162.50
SPIDER MAN	1963	3	VERY FINE (VF)	\$ 122.50
SPIDER MAN	1963	3	NEAR MINT (NM)	\$ 162.50

Figure 24: Spider Man comics with price less than \$200.

5. How many FANTASTIC FOUR books can be bought where the quality is greater than very good and issue before 150 and can be bought as a group for \$100 or less in total.

```
SELECT COUNT(*) as books_that_can_be_bought, CONCAT("$",SUM(p.price)) as total
FROM comics c
JOIN pricing p on p.comicID = c.comicID
JOIN issue i on i.issueID = p.issueID
JOIN grades g on p.gradeID = g.gradeID
WHERE c.title = 'FANTASTIC FOUR'
AND i.issueNumber < 100
AND g.gradeNumber <= (SELECT min(gradeNumber) from grades where symbol='VERY GOOD (VG)')
HAVING SUM(p.price)<100;
```

This SQL query is used to find out the number of books and the total cost of books that can be bought for a given comic series 'FANTASTIC FOUR'. The conditions for a book to be included in the result are that the book should have an issue number less than 100, its grade should be less than or equal to 'VERY GOOD (VG)', and its price should be less than 100 dollars.

The query first joins the 'comics', 'pricing', 'issue', and 'grades' tables using their respective IDs. Then it applies the specified conditions using WHERE clause. The HAVING clause is used to filter out the results based on the total cost of books.

The result of the query will have two columns: 'books_that_can_be_bought' which gives the count of books that satisfy the given conditions, and 'total' which gives the total cost of those books.

This query can be useful for a user who is interested in buying books of the 'FANTASTIC FOUR' series and has a budget of 100 dollars. It helps the user in finding out the number of books that they can buy and their total cost based on certain conditions.

books_that_can_be_bought	total
2	\$ 10.31

Figure 25: Books that can be bought below \$100

6. Query to get the number of orders and revenue by customer for a specified date range.

```
SELECT u.username, COUNT(o.orderID) AS total_orders, CONCAT("$",SUM(oi.subtotal)) AS
revenue
FROM users u
JOIN orders o ON u.userID = o.userID
JOIN orderItems oi ON o.orderID = oi.orderID
WHERE o.orderDate BETWEEN '2023-01-01' AND '2023-06-30'
GROUP BY u.userID;
```

This query retrieves information about the number of orders and revenue generated by each user between January 1, 2023 and June 30, 2023.

The users table is joined with the orders table and the orderItems table to obtain the necessary information. The COUNT() function is used to count the number of orders made by each user, while the SUM() function is used to calculate the total revenue generated by each user.

The results are grouped by userID and displayed along with the username and revenue generated in a formatted string.

This query can be used to analyze the performance of individual users over a specific time period and to identify the most profitable users. It can also help in making business decisions related to marketing strategies, customer service, and product offerings.

username	total_orders	revenue
emilybrown	3	\$ 385.00
davidlee	6	\$ 31678.75
sarahjohnson	4	\$ 736.25
michaeladams	3	\$ 34423.75
katewilson	3	\$ 144.38

Figure 26: Revenue by customer

7. Query to find the comics that are bought together.

```

SELECT CONCAT(c1.title, ' and ', c2.title) AS combo, COUNT(*) AS frequency
FROM orderItems oi1
JOIN orderItems oi2 ON oi1.orderId = oi2.orderId AND oi1.comicID < oi2.comicID
JOIN comics c1 ON oi1.comicID = c1.comicID
JOIN comics c2 ON oi2.comicID = c2.comicID
GROUP BY c1.title, c2.title
HAVING COUNT(*) >= 2
ORDER BY COUNT(*) DESC;

```

This query retrieves the frequency with which pairs of comics are sold together as part of an order. It joins the orderItems table with itself, matching each order with all the other orders that contain different comics, and then joins the comics table to retrieve the titles of the comics involved. The results are then grouped by the title of the two comics and filtered to only show those pairs that have been sold together at least twice, and finally sorted in descending order by frequency.

The purpose of this query is to identify which pairs of comics are commonly purchased together, which can be useful for making recommendations to customers or for bundling products to increase sales. The query can help the application to better understand customer behavior and tailor marketing strategies to increase revenue.

combo	frequency
SPIDER MAN and FANTASTIC FOUR	2
AVENGERS and FANTASTIC FOUR	2

Figure 27: Comics bought together

8. Query to list out all the comics with a speciality like signed by writer etc.

```
SELECT c.title, i.issueNumber
FROM comics c
JOIN issue i ON i.comicID = c.comicID
JOIN pricing p ON p.comicID = i.comicID and i.issueID = p.issueID
WHERE p.speciality = "Yes";
```

This SQL query retrieves all the titles and issue numbers of comics that have a speciality pricing flag set to "Yes".

In the database schema, the "pricing" table has a column named "speciality" that indicates whether a comic has a speciality pricing or not. By joining the "comics" table with the "issue" table, we can retrieve the title and issue number of the comic. Finally, by joining the "pricing" table with the "issue" table using the comic ID and issue ID, we can retrieve the speciality pricing flag for each comic issue.

The query is useful for the application because it allows users to easily find and purchase comics that have a speciality pricing, which may include rare or collectible issues that have a higher value than regular issues or is signed by the writer.

title	issueNumber
SPIDER MAN	3
SPIDER MAN	4
AVENGERS	3
FANTASTIC FOUR	1
FANTASTIC FOUR	3
ALL NEW X-MEN	3
ALL NEW X-MEN	4
BATMAN ADVENTURES	2
BATMAN ADVENTURES	3
FLASH	1
FLASH	4
SUPERMAN: FOR THE ANIMALS	1
WONDER WOMAN	100

Figure 28: List of all special comics

7. Conclusion

The database we have is excelsior comic book store. It has tables for storing information about comics, pricing, issues, orders, order items, users, and grades. The tables are related to each other by foreign keys and the database has been optimized to provide quick and efficient access to the data.

The queries we have examined are examples of how the database can be used to retrieve specific information from it. For example, we can find the total revenue generated by the store in a given time period, or we can identify which comics are frequently bought together. These queries demonstrate how the database can provide valuable insights into the store's operations.

The database can also support future developments by allowing the store to easily add new features to its online platform. For example, the store can use the data in the database to create personalized recommendations for customers or to offer discounts on specific items. The database can also be used to track customer preferences and purchase history, which can help the store identify new trends and adjust its inventory accordingly.

The database can be used to build a comic grading tool that enables users to evaluate the condition of a comic book and assign a grade to it based on industry standards. The tool can utilize the data on comic book grades, issue numbers, and pricing to provide a comprehensive grading system. The data on sales, orders, and customers in the database can be used to build a sales analytics dashboard that provides insights into the business performance. The dashboard can display key performance indicators such as revenue, sales volume, and customer acquisition.

Overall, the database is a powerful tool that can help excelsior make data-driven decisions and provide a better shopping experience for its customers.

Acknowledgements

I would like to take this opportunity to acknowledge my own efforts in completing the design of the Excelsior Comic Books database. Through careful planning and attention to detail, I was able to create a database that is structured to meet the specific needs of the Excelsior Comic Books. I worked diligently to ensure that the database was not only functional, but also user-friendly, making it easier to access and update important information. I am proud of the work that I have done and believe that the Excelsior Comic Books database will be a valuable asset moving forward.

References

- [1] Database System Concepts", Seventh Edition by Abraham Silberschatz, Henry Korth, and S. Sudarshan.
- [2] SQL Queries for Mere Mortals, Second Edition by John L. Viescas and Michael J. Hernandez.
- [3] MySQL Stored Procedure Programming , Harrison, G. and Feuerstein, S. (2006).