

# Relational Databases

## Subqueries

8 February 2024

克明峻德，格物致知

# Quick Recap: Aggregation and Grouping

## Key Concepts:

- Aggregation functions, combined with GROUP BY, compress large datasets into concise summaries. This allows us to uncover patterns and trends, providing vital insights for data-driven decision-making.
- The application of SQL's FWHOS structure, incorporating aggregation and grouping, facilitates detailed and accurate data analysis. This approach highlights the relational model's efficacy in generating comprehensive and useful business intelligence.

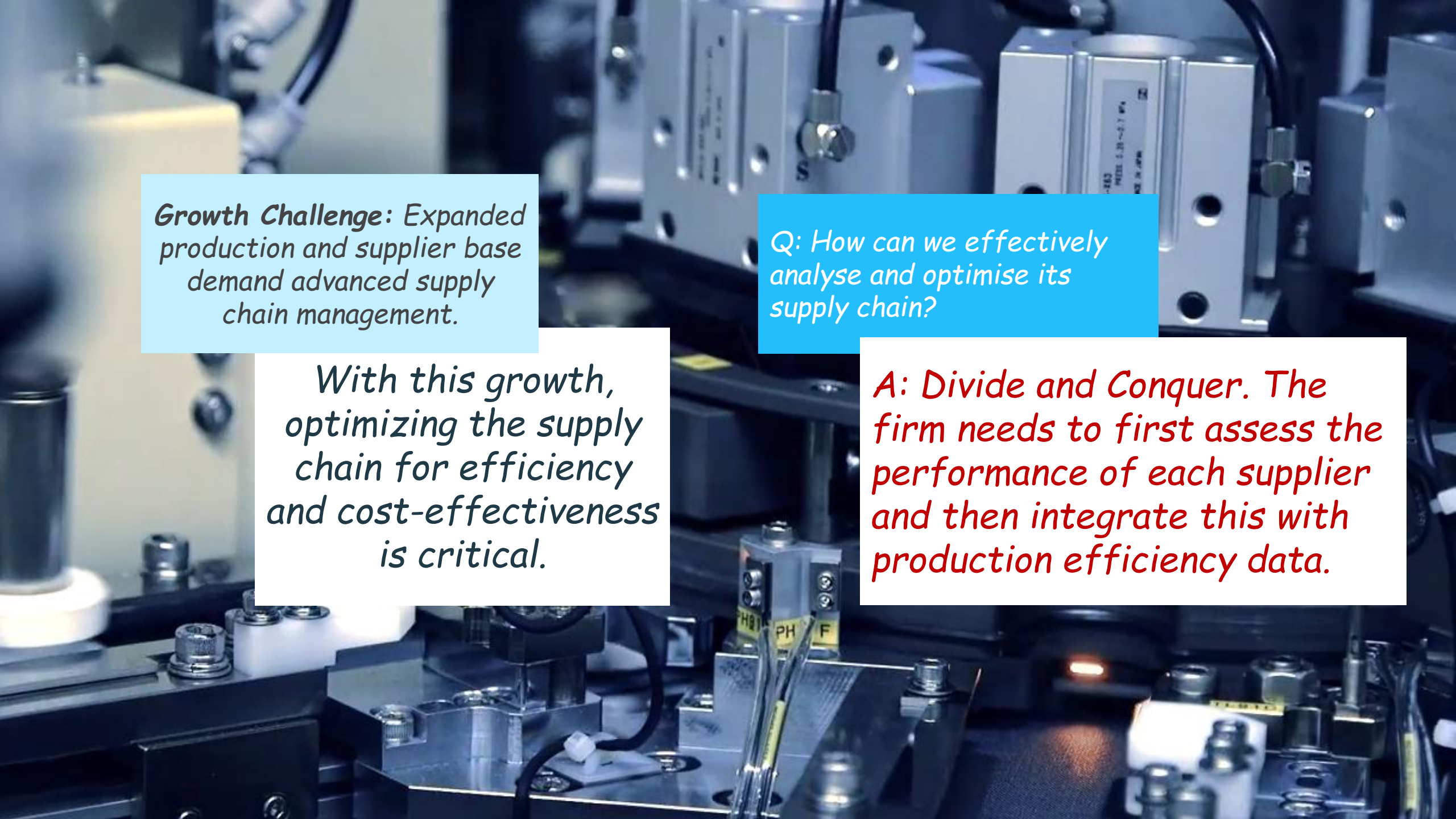
## SQL Query Example

```
SELECT Genre, AVG(Rating) AS AverageRating
FROM Books
INNER JOIN BookRatings
ON Books.BookID = BookRatings.BookID
GROUP BY Genre
HAVING COUNT(BookID) >= 3;
```

- Output Table:

Genre	AverageRating
Technology	4.0
Fiction	4.0

- **Purpose:** To identify genres that are highly rated by readers (above a certain threshold).

The background of the entire slide is a close-up, slightly blurred image of industrial machinery, likely a CNC machine or a similar manufacturing equipment. It features various metal components, bolts, and a complex arrangement of parts, with a blueish tint overall.

***Growth Challenge:** Expanded production and supplier base demand advanced supply chain management.*

*Q: How can we effectively analyse and optimise its supply chain?*

*With this growth, optimizing the supply chain for efficiency and cost-effectiveness is critical.*

*A: Divide and Conquer. The firm needs to first assess the performance of each supplier and then integrate this with production efficiency data.*

# Complex Query

## Identifying Highest Ratings for Each Book

- Books Table:

BookID	Title	Author
101	Journey Through SQL	A. Coder
102	The History of Databases	D. Base
103	Adventures in Coding	P. Programmer
...	...	...

- BookRatings Table:

BookID	MemberID	Rating
101	501	4
101	502	5
102	501	3
102	502	4

- SQL Query:

```
SELECT B.Title, BR.Rating
FROM Books B
INNER JOIN BookRatings BR ON B.BookID = BR.BookID
ORDER BY B.Title, BR.Rating DESC;
```

## Challenges:

- The result includes every rating for each book, not just the top rating.
- Identifying the highest rating for each book requires additional filtering or aggregation, which is not straightforward with this method.

# Breaking Down Complex Query

## Query Challenge Recap:

- **Objective:** Identify the highest rating received for each book in the library.
- **Complexity:** Direct JOINS produce a list of all ratings per book without isolating the top rating for each.

## Breaking Down the Problem:

- **Aggregate Ratings:** Determine the highest rating given to each book.
- **Match Books with Ratings:** Associate these top ratings with the corresponding books.
- **Present Final Results:** Display the book titles along with their highest ratings.

Step 1: Find Maximum Rating per Book

Query: `SELECT BookID, MAX(Rating) FROM BookRatings GROUP BY BookID`

|  
V

Step 2: Join with Books for Book Titles

Action: `INNER JOIN ON BookID`

|  
V

Final Output: Display Book Title and Highest Rating

# WITH Subqueries (CTEs)

## Identifying Highest Ratings for Each Book

- Books Table:

BookID	Title	Author
101	Journey Through SQL	A. Coder
102	The History of Databases	D. Base
103	Adventures in Coding	P. Programmer
...	...	...

- BookRatings Table:

BookID	MemberID	Rating
101	501	4
101	502	5
102	501	3
102	502	4

- SQL Query with WITH Subquery:

```
WITH MaxBookRatings AS (  
    SELECT BookID, MAX(Rating) AS HighestRating  
    FROM BookRatings  
    GROUP BY BookID  
)  
SELECT B.Title, MBR.HighestRating  
FROM MaxBookRatings MBR  
INNER JOIN Books B ON MBR.BookID = B.BookID;
```

- Output Table:

Title	Highest Rating
Journey Through SQL	5
The World of Fiction	4

## Explanation:

- WITH Subquery (MaxBookRatings):** Calculates each book's maximum rating.
- Main Query:** Joins MaxBookRatings with Books to match books to their top ratings.

# Correlated Subqueries

## Background

- A correlated subquery is a subquery that references columns from the outer query, making it dependent on the outer query.

- Correlated Query Example:

```
SELECT B.Title, B.PublishedYear,  
       (SELECT MAX(Rating)  
        FROM BookRatings BR  
        WHERE BR.BookID = B.BookID) AS MaxRating  
FROM Books B;
```

- Output Table:

Title	PublishedYear	MaxRating
SQL Essentials	2018	5
Advanced SQL	2020	5

- **Purpose:** Retrieves each book's title, publication year, and its highest rating.

## Explanation:

- The correlated subquery (SELECT MAX(Rating) FROM BookRatings BR WHERE BR.BookID = B.BookID) is dependent on the outer query, as it uses B.BookID from the Books table.
- Each time a row from Books is processed, the subquery executes, which can lead to inefficiencies, especially with large datasets.

# Decorrelated Subqueries

## Background

- A decorrelated subquery is a standalone query where the execution is not dependent on the outer query. This often leads to more efficient execution.
- Decorrelated Query Example:

```
WITH MaxRatings AS (  
    SELECT BookID, MAX(Rating) AS MaxRating  
    FROM BookRatings  
    GROUP BY BookID  
)  
  
SELECT B.Title, B.PublishedYear, MR.MaxRating  
FROM Books B  
INNER JOIN MaxRatings MR ON B.BookID = MR.BookID;
```

- Output Table:

Title	PublishedYear	MaxRating
SQL Essentials	2018	5
Advanced SQL	2020	5

- **Purpose:** Retrieves each book's title, publication year, and its highest rating.

## Explanation:

- The WITH clause creates a decorrelated subquery MaxRatings that independently calculates the maximum rating for each book.
- The main query then joins the result with the Books table.
- The subquery is executed only once, and its result is reused in the main query, making it more efficient.



# Comparing Subqueries in JOINS vs. CTEs (WITH)

---

## Subqueries in JOINS:

- Embedded directly in the JOIN clause.
- Executed as part of the join operation, which can be inefficient for large data.
- Example Use Case: Joining a table with aggregated ratings for each book.

## Common Table Expressions (CTEs) with WITH:

- Defined separately and then referenced in the main query.
- Executed once, making it more efficient for large data or repeated use.
- Example Use Case: Aggregating book ratings and then joining with books and authors tables.

## Conclusion:

- Use Subqueries in JOINS when dealing with smaller datasets or complex join conditions that don't necessitate repeated execution.
- Use CTEs (WITH) for larger datasets, complex queries requiring multiple steps, or when intermediate results are reused.

# WHERE EXISTS and WHERE NOT EXIST

## Background

- EXISTS and NOT EXISTS are SQL operators used in WHERE clauses to test for the existence or non-existence of related data in a another table.

- Members Table:

MemberID	Name
501	Alice Johnson
502	Bob Smith
503	Carol Martinez

- SQL Query with WITH EXISTS:

```
SELECT M.Name
FROM Members M
WHERE EXISTS (
    SELECT *
    FROM Borrow B
    WHERE B.MemberID = M.MemberID AND B.ReturnDate IS NULL
);
```

- **Puprose:** This query finds members who currently have at least one book borrowed that hasn't been returned yet.
- The EXISTS subquery checks for records in the Borrow table with ReturnDate as NULL for each member.

- Borrow Table:

BookID	MemberID	BorrowDate	ReturnDate
101	501	2022-01-01	2022-01-10
102	502	2022-02-01	NULL

- Output Table:

Name
Bob Smith

# WHERE NOT EXISTS for Data Ingestion

## Preventing Duplicate Entries

- In data ingestion, particularly when importing data from external sources, it's essential to ensure that duplicate records are not inserted. WHERE NOT EXISTS is a powerful tool for this purpose.
- SQL Query with WITH NOT EXISTS:

```
INSERT INTO Books (BookID, Title, Genre, PublishedYear)
SELECT NewData.BookID, NewData.Title, NewData.Genre, NewData.PublishedYear
FROM (VALUES
    (103, 'Data Science 101', 'Technology', 2021),
    (102, 'Advanced SQL', 'Technology', 2020)
) AS NewData (BookID, Title, Genre, PublishedYear)
WHERE NOT EXISTS (
    SELECT BookID
    FROM Books B
    WHERE B.BookID = NewData.BookID
);
```

- **Purpose:** WHERE NOT EXISTS is used to filter out any books that already exist in our database, preventing duplicate entries.
- **Efficiency:** WHERE EXISTS/NOT EXISTS is superior in scenarios where the early termination of the query is possible. Particularly efficient for filtering operations.

# Set Operations

---

## Background:

- Set Operations in SQL, including UNION, INTERSECT, and EXCEPT, are used to combine the results of two different queries. Unlike subqueries, these operations are more akin to mathematical set theory.

## Key Set Operations:

### 1. UNION:

- Combines the results of two queries into a single result set, excluding duplicate rows.
- Use Case: Retrieve a list of all distinct values appearing in either query.

### 2. INTERSECT:

- Returns only the rows that appear in both query result sets.
- Use Case: Find common elements between two sets of data.

### 3. EXCEPT:

- Returns rows from the first query that are not present in the second query's result set.
- Use Case: Identify unique elements in the first set that don't exist in the second set.

# Set Operation Example on Books Table

## Scenario:

- Using the Books table from a library database, we'll apply SQL set operations to categorize books as either recent (published after 2000) or classic (published before 2000).

## EXCEPT - Titles Exclusive to Recent Books

- SQL Query:

```
SELECT Title FROM Books WHERE PublishedYear >= 2000  
EXCEPT  
SELECT Title FROM Books WHERE PublishedYear < 2000;
```

- Purpose:** Find titles that are exclusively in the recent books category.
- Output Table:**

Title
Modern Data Science
Contemporary Fiction
SQL Today

# Hand-On SQL Demonstration

- **Focus:** Utilising Subqueries with WITH in Advanced Data Analysis
- **Goal:** Demonstrate the effective use of WITH subqueries for sophisticated data retrieval and analysis in a library database.

## Example Query

- Find Books with Ratings Above Their Genre's Average:

```
WITH GenreAvg AS (  
    SELECT Genre, AVG(Rating) AS AvgRating  
    FROM BookRatings  
    GROUP BY Genre  
)  
  
SELECT b.Title, br.Rating  
FROM Books b  
INNER JOIN BookRatings br ON b.BookID = br.BookID  
INNER JOIN GenreAvg g ON b.Genre = g.Genre  
WHERE br.Rating > g.AvgRating;
```

## Try It Yourself

1. Identify Genres with Above Average Book Ratings:
  - **Task:** Write a query using WITH to find genres whose books have an average rating above the overall average rating.
2. List Highest Rated Books in Each Genre:
  - **Task:** Modify the query to list the highest-rated book(s) in each genre, using WITH for genre-specific maximum ratings.

# Recap and Key Takeaways

- Subqueries in SQL, both correlated and decorrelated, enable detailed and conditional data extraction, facilitating complex data operations while maintaining relational database integrity.
- Correlated subqueries enable precise filtering, while decorrelated ones and CTEs (using WITH) boost SQL query efficiency. Subqueries are key for executing complex, conditional logic, vital in business operations and strategic decisions.

```
WITH AverageRating AS (  
    SELECT AVG(Rating) AS AvgRating FROM Books  
)  
  
SELECT Title  
FROM Books  
WHERE Rating > (SELECT AvgRating FROM AverageRating);
```

- **Purpose:** Identify books in the library that have a rating above the overall average.

## Preparing for Entity-Relationship Diagram:

- Next Lecture Preview:
  - Explore Conceptual Database Design using Entity-Relationship Diagrams (ERDs).
  - Understand how ERDs visually represent database schemas, illustrating entities, relationships, and key constraints.

*Imagine you're working with a database containing customer orders and products. How would you use a subquery to find products that have never been ordered?*