SQL 3: Alias, Subqueries, Aggregate Functions & Grouping

Databases and Interfaces

Matthew Pike & Yuan Yao

University of Nottingham Ningbo China (UNNC)

Overview

This Lecture

- In this lecture we will cover:
    - Aliases - to make queries more readable
    - Subqueries - for more complex queries
    - Aggregate functions - to summarise data
    - `GROUP BY` - grouping data and applying aggregate functions

The Database Schema for this Lecture

```
CREATE TABLE Student(
    sID INTEGER PRIMARY KEY,
    firstName VARCHAR(20) NOT NULL,
    lastName VARCHAR(20) NOT NULL
);
```

```
CREATE TABLE Module(
    mCode CHAR(8) PRIMARY KEY,
    title VARCHAR(30) NOT NULL,
    credits INTEGER NOT NULL
);
```

```
CREATE TABLE Grade(
    sID INTEGER NOT NULL,
    mCode CHAR(8) NOT NULL,
    grade INTEGER NOT NULL,
    PRIMARY KEY (sID, mCode),
    FOREIGN KEY (sID)
        REFERENCES Student(sID),
    FOREIGN KEY (mCode)
        REFERENCES Module(mCode)
);
```

The Database Content for this Lecture

sID	firstName	lastName
1	John	Smith
2	Jane	Doe
3	Mary	Jones
4	David	Smith

Table 1: `Student` Table

mCode	title	credits
COMP1036	Fundamentals	20
COMP1048	Databases	10
COMP1038	Programming	20

Table 2: `Module` Table

sID	mCode	grade
1	COMP1036	35
1	COMP1048	50
2	COMP1048	65
2	COMP1038	70
3	COMP1036	35
3	COMP1038	65

Table 3: `Grade` Table

Using Aliases to Rename Columns and Tables

Aliases

An alias is a temporary name for a table or column. It can be used to:

- Make queries more readable
- Shorten column names
- Resolve ambiguous names

Aliases are specified using the AS keyword. For example:

```
SELECT column_name AS alias_name
FROM table_name;
```

Using Aliases to Rename Columns

We can use aliases to rename columns in a query. This is useful for:

- Making queries more readable
- Shortening or standardising column names
- Resolving ambiguous names, for example, when joining tables (covered in a later lecture)

```
SELECT
    sID AS 'ID',
    firstName AS 'First',
    lastName AS 'Last'
FROM
    Student;
```

ID	First	Last
1	John	Smith
2	Jane	Doe
3	Mary	Jones
4	David	Smith

Table 4: Rename Columns using Aliases

Using Aliases to Rename Tables

! Alias in WHERE Clause

You **cannot** create a column alias in a **WHERE** clause, because the column value might not yet be determined when the **WHERE** clause is executed.

```
SELECT
    s.sID AS 'ID',
    s.lastName AS 'Last'
FROM
    Student AS s
WHERE
    s.sID < 3;
```

ID	Last
1	Smith
2	Doe

Table 5: Associating student names with grades via a table Alias.

Subqueries

Subqueries in SQL

A SELECT statement can be nested inside another SELECT statement.

- The inner SELECT statement is called a **subquery**.
- The outer SELECT statement is called a **main query** or **outer query**.

Subqueries are useful when you need to:

- Filter a table based on the results of another query
- Calculate a value based on the results of another query

Subqueries are specified using parentheses:

```
SELECT *
FROM table
WHERE column [IN|EXISTS|=] (SELECT column FROM table);
```

Subqueries in SQL: Example

- The subquery returns the IDs of students who have scored 70 or higher in a module, as recorded in the `Grade` table.
- The outer query then returns the `firstName` and `lastName` of students whose `sID` is found in the subquery results.

```sql
SELECT
    firstName, lastName
FROM Student
WHERE sID IN (
    SELECT sID
    FROM Grade
    WHERE grade >= 70
);
```

firstName	lastName
Jane	Doe

Table 6: Names of students who have achieved >= 70 in a module.

Subqueries Processing Order

- Subqueries are processed **before** the outer query. If there are multiple levels of subqueries, they are processed from the **innermost subquery** to the outermost query.
- The results of the subquery are treated as an **intermediate result set**, which is then used by the outer query.

```sql
SELECT grade
FROM Grade
WHERE mCode = (
    SELECT mCode
    FROM Module
    WHERE title =
    'Databases'
);
```

```sql
SELECT grade
FROM Grade
-- The result of the subquery
-- is used in the outer query
WHERE mCode = 'COMP1048';
```

Subqueries with Sets of Values

A subquery often returns a set of values, not just a single value. These sets can be used to filter data in the outer query.

Common Operators

- `IN`: Returns TRUE if a **value matches any** in the set.
- `EXISTS`: Returns TRUE if the subquery returns **any rows**.
- `NOT IN`: Returns TRUE if a **value does not match** any in the set.
- `NOT EXISTS`: Returns TRUE if the subquery **returns no rows**.

Set Specification

- Using a subquery:

```
SELECT * FROM table WHERE column [IN|EXISTS] (SELECT column FROM table);
```

- Using a list of values:

```
SELECT * FROM table WHERE column [IN|EXISTS] (value1, value2, ...);
```

Using IN with Subqueries

- The IN operator checks if a value exists within a set of values returned by a subquery.
- It simplifies filtering by matching values from one table with those from a subquery result.

```sql
SELECT title AS 'Module Title'
FROM Module
WHERE mCode IN (
    SELECT mCode
    FROM Grade
    WHERE grade >= 70
);
```

Module Title
Programming

Table 7: The names of modules in which a student has scored >= 70.

Using NOT IN with Subqueries

- The NOT IN operator checks if a value is not in a set of values returned by a subquery or a list.
- It allows filtering by excluding certain values from the result set.

```sql
SELECT
    firstName AS 'First',
    lastName AS 'Last'
FROM Student
-- Note here, we could also have
-- used a subquery as in the
-- previous example.
WHERE sID NOT IN (1, 2);
```

First	Last
Mary	Jones
David	Smith

Table 8: List of students whose IDs are not 1 or 2

Using EXISTS with Subqueries

- The EXISTS operator checks whether a subquery returns any rows.
- If the subquery returns at least one row, the condition is TRUE; otherwise, it's FALSE.

```sql
SELECT
    firstName AS 'First',
    lastName AS 'Last'
FROM Student s
WHERE EXISTS (
    SELECT sID
    FROM Grade
    WHERE sID = s.sID
);
```

First	Last
John	Smith
Jane	Doe
Mary	Jones

Table 9: This query returns the names of students who have at least one entry in the `Grade` table, i.e., students with at least one grade recorded.

Using NOT EXISTS with Subqueries

- The NOT EXISTS operator is used to check if no rows are returned by a subquery.
- It evaluates to TRUE when the subquery **finds no matching rows**.

```sql
SELECT
    title AS 'Module Title'
FROM Module m
WHERE NOT EXISTS (
    SELECT mCode
    FROM Grade
    WHERE grade >= 70
    AND mCode = m.mCode
);
```

Module Title
Fundamentals
Databases

Table 10: The names of modules in which no student has scored >= 70.

`IN` vs `EXISTS`

DBMS Query Optimisation

The DBMS will attempt to optimise the query to use the most efficient method, but performance differences can still arise based on the specific DBMS and the size of the datasets involved.

- `IN` is used to check if a value is in a set of values
    - `IN` is suited to static or small sets of values (e.g., lists of constants)
    - It is often more efficient than `EXISTS` for these cases
- `EXISTS` is used to check if a subquery returns any rows
    - `EXISTS` is suited to dynamic or large sets of values
    - It is typically more efficient than `IN` when dealing with subqueries, especially for large datasets, as it stops evaluating once a match is found

Nested Subqueries

- Each subquery is evaluated step-by-step, and its results are used in the next subquery.
- This hierarchical approach allows narrowing down results in stages.

```sql
SELECT firstName AS 'First',
        lastName AS 'Last'
FROM Student WHERE sID IN (
    SELECT sID FROM Grade
    WHERE mCode IN (
        SELECT mCode FROM Module
        WHERE title IN
        ('Fundamentals',
        'Programming')
    ));
```

First	Last
John	Smith
Jane	Doe
Mary	Jones

Table 11: The names of students who have enrolled in the Fundamentals or Programming modules.

An Aside: Testing for NULL Values

- NULL is a special value in SQL that represents an unknown or missing value.
- NULL is not equal to any other value, including another NULL.
- Therefore, we cannot use = or != to test for NULL values because these comparisons return UNKNOWN.
- Instead, we must use the IS NULL or IS NOT NULL operators to check for NULL values:
  - Examples:

```
SELECT * FROM table WHERE column IS NULL;
SELECT * FROM table WHERE column IS NOT NULL;
```

- **Common Mistake**:
  - The following query will not work:

```
  SELECT * FROM table WHERE column = NULL;  -- Incorrect!
```

  - This is because direct comparisons with NULL using = or != will not evaluate as expected.

Handling NULL with IFNULL

In SQL, the `IFNULL` (or `COALESCE` in some databases) function can be used to replace `NULL` values with a default value. This is helpful when we want to avoid `NULL` results in query outputs.

- Example of `IFNULL`:
  - If we want to return a default value when encountering `NULL`:

```sql
SELECT IFNULL(column, 'Default Value') FROM table;
```

  - If we wanted grades to be displayed as ‘No Grade’ when they are `NULL`:

```sql
SELECT IFNULL(grade, 'No Grade') FROM Grade;
```

This query will return ‘Default Value’ if `column` contains `NULL`, otherwise, it will return the value of `column`.

Aggregate Functions

Arithmetic Operators

- Arithmetic operators are used to perform calculations on numeric values.
- The following arithmetic operators are available:
  - `+` - addition
  - `-` - subtraction
  - `*` - multiplication
  - `/` - division
  - `%` - modulus (remainder of division)
- For exponentiation, SQL uses the `POWER()` function. Example:

```sql
SELECT POWER(2, 3);  -- Returns 8.0
```

Arithmetic Operators: Example

Handling Spaces in Names

Column or alias names with spaces must be enclosed in single quotes, square brackets, or backticks. You are suggested to use single quotes.

```sql
SELECT
    Grade AS 'Original',
    Grade - 5 AS 'G - 5',
    Grade + 5 AS [G + 5]
FROM Grade
 -- Only show the first 5 rows
LIMIT 5;
```

Original	G - 5	G + 5
35	30	40
50	45	55
65	60	70
70	65	75
35	30	40

Table 12: Adjusting all grades.

Aggregate Functions

- Aggregate functions are used to perform calculations on a set of values
- The following aggregate functions are available (not an exhaustive list):
    - `COUNT` - returns the number of rows
        - `COUNT(*)` counts all rows, including duplicates and `NULL`s.
        - `COUNT(column)` counts non-`NULL` values in a specific column.
    - `SUM` - returns the sum of a column.
    - `AVG` - returns the average of a column.
    - `MIN` - returns the minimum value of a column.
    - `MAX` - returns the maximum value of a column.
- We can also control the presentation of the results using the `ROUND` function:
    - `ROUND` - rounds a number to a specified number of decimal places. This example rounds the average grade to two decimal places:

```sql
SELECT ROUND(AVG(Grade), 2) AS 'Average Grade' FROM Grade;
```

Aggregate Functions: Example

```sql
SELECT
    SUM(grade) AS 'Sum',
    AVG(grade) AS 'Average',
    ROUND(AVG(grade),2)
        AS 'Rounded Avg',
    MIN(grade) AS 'Low',
    MAX(grade) AS 'High'
FROM Grade;
```

Sum	Average	Rounded Avg	Low	High
320	53.33333	53.33	35	70

Table 13: Student Grade Statistics

Using COUNT

- The COUNT function returns the number of rows in a table.
    - COUNT(*) counts all rows, including those with NULL values.
    - COUNT(column) counts only non-NULL values in a specific column.
    - COUNT(DISTINCT column) counts unique non-NULL values in a column.

```sql
SELECT COUNT(*)
    AS 'Number of Students'
FROM Student;
```

Number of Students
4

Table 14: How many students are in our DB?

```sql
SELECT COUNT(DISTINCT mCode)
    AS 'Unique Modules with Grades'
FROM Grade
WHERE grade < 40;
```

Unique Modules with Grades
1

Table 15: How many modules have students with a grade of <40?

Combining Aggregate Functions

Aggregate functions can also be combined for simple statistical calculations, such as calculating the range of values or comparing averages with extremes.

```
SELECT
    MAX(Grade) - MIN(Grade)
        AS 'Range of Marks',
    AVG(Grade) - MIN(Grade)
        AS 'Average - Lowest'
FROM Grade;
```

```
SELECT
    SUM(Grade)
        AS 'Total Grades',
    ROUND(AVG(Grade), 0)
        AS 'Average Grade'
FROM Grade;
```

Range of Marks	Average - Lowest
35	18.33333

Table 16: Student Grade Statistics

Total Grades	Average Grade
320	53

Table 17: Grade Sum and Average

String Functions

- String functions are used to perform calculations/operations on string values.
- The following string functions are available:
  - `||` - concatenates two or more strings.
    - Example: `'Hello' || ' ' || 'World'` results in `'Hello World'`.
  - `LENGTH` - returns the length of a string.
    - Example: `LENGTH('Hello')` returns `5`.
  - `LOWER` - converts a string to lowercase.
    - Example: `LOWER('Hello')` returns `'hello'`.
  - `UPPER` - converts a string to uppercase
    - Example: `UPPER('Hello')` returns `'HELLO'`

Example: Student Names as a Single Column

We can use the `||` (concatenation) function to combine the first and last names of students into a single column. Note that other DBMSs have a `CONCAT` function for this purpose.

In this example, we also use the `UPPER` function to convert last names to uppercase, and the `LENGTH` function to calculate the combined length of the first and last names. We add 1 to the length to account for the space between the names.

```sql
SELECT
    firstName || ' ' || UPPER(lastName)
        AS 'Student Name',
    LENGTH(firstName) + LENGTH(lastName) + 1
        AS 'Length'
FROM Student;
```

Student Name	Length
John SMITH	10
Jane DOE	8
Mary JONES	10
David SMITH	11

Table 18: Combining first and last names.

Grouping Data

Grouping Data using GROUP BY

- Often, we want to perform calculations on subsets of data, such as finding the average, sum, or count for specific categories.
- The `GROUP BY` clause is used to group data by one or more columns, allowing you to perform calculations for each group.
- The `GROUP BY` clause is typically used in conjunction with aggregate functions like `AVG`, `SUM`, `COUNT`, etc. = Note: Any column in the `SELECT` list that is not an aggregate function must be included in the `GROUP BY` clause.

Example: Find the Average Grade for each Module

```
SELECT
    mCode
        AS 'Module Code',
    AVG(grade)
        AS 'Average Grade'
FROM Grade
GROUP BY mCode;
```

Module Code	Average Grade
COMP1036	35.0
COMP1038	67.5
COMP1048	57.5

Table 19: Average Grade for each Module

HAVING Clause

When to use HAVING vs WHERE

The WHERE clause is used to filter rows before grouping and aggregation. Whereas, the HAVING clause is used to filter groups after aggregation.

- The HAVING clause is used to filter groups of data based on conditions applied to aggregate functions.
- The HAVING clause is used after the GROUP BY clause, and it's typically applied to aggregate functions like AVG, SUM, COUNT, etc.

Example: Average Grade for Modules Whose Average Grade >= 60

```
SELECT
    mCode
        AS 'Module Code',
    AVG(grade)
        AS 'Average Grade'
FROM Grade
GROUP BY mCode
HAVING AVG(grade) >= 60;
```

Module Code	Average Grade
COMP1038	67.5

Table 20: Average Grade for Modules with an average grade >= 60.

The `UNION` Clause

The `UNION` Clause: Combining Results

The `UNION` Clause: Adding Rows

The `UNION` clause allows us to add rows from multiple `SELECT` statements into a single result set, unlike other clauses that are primarily used for filtering rows or columns.

- The `UNION` clause combines the results of two or more `SELECT` queries into a single dataset.
- It's particularly useful when you want to combine data from different sources or results into one report.

```sql
SELECT MAX(grade) AS Grade
FROM Grade
UNION
SELECT MIN(grade) AS Grade
FROM Grade;
```

Example: Generate a Report of Module and Overall Average Grades

```
SELECT
    mCode AS 'Module Code',
    ROUND(AVG(grade),2)
        AS 'Average Grade'
FROM Grade
GROUP BY mCode
UNION
SELECT
    'Overall' AS 'Module Code',
    ROUND(AVG(grade),2)
        AS 'Average Grade'
FROM Grade;
```

Module Code	Average Grade
COMP1036	35.00
COMP1038	67.50
COMP1048	57.50
Overall	53.33

Table 21: Average Grade for Each Module and Overall Average Grade

References

Learning Resources

Online Tutorials

These are clickable links to the online tutorials:

- SQLite `IN` - https://sqlitetutorial.net/sqlite-in/
- SQLite `EXISTS` - https://sqlitetutorial.net/sqlite-exists/
- SQLite Subqueries - https://sqlitetutorial.net/sqlite-subquery/
- SQLite `Aggregate Functions` - https://sqlitetutorial.net/sqlite-aggregate-functions/
- SQLite `UNION` Operator - https://sqlitetutorial.net/sqlite-union/
- SQLite `GROUP BY` - https://sqlitetutorial.net/sqlite-group-by/
- SQLite `HAVING` - https://sqlitetutorial.net/sqlite-having/

Textbooks and Documentation

- Chapters 6 & 7 of the Database Systems textbook
- SQLite Documentation

Reference Materials

Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H., & Cilimdzic, M. (2004). Robust query processing through progressive optimization. *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 659–670.

Zhao, Y., Deshpande, P. M., & Naughton, J. F. (1997). An array-based algorithm for simultaneous multidimensional aggregates. *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, 159–170.