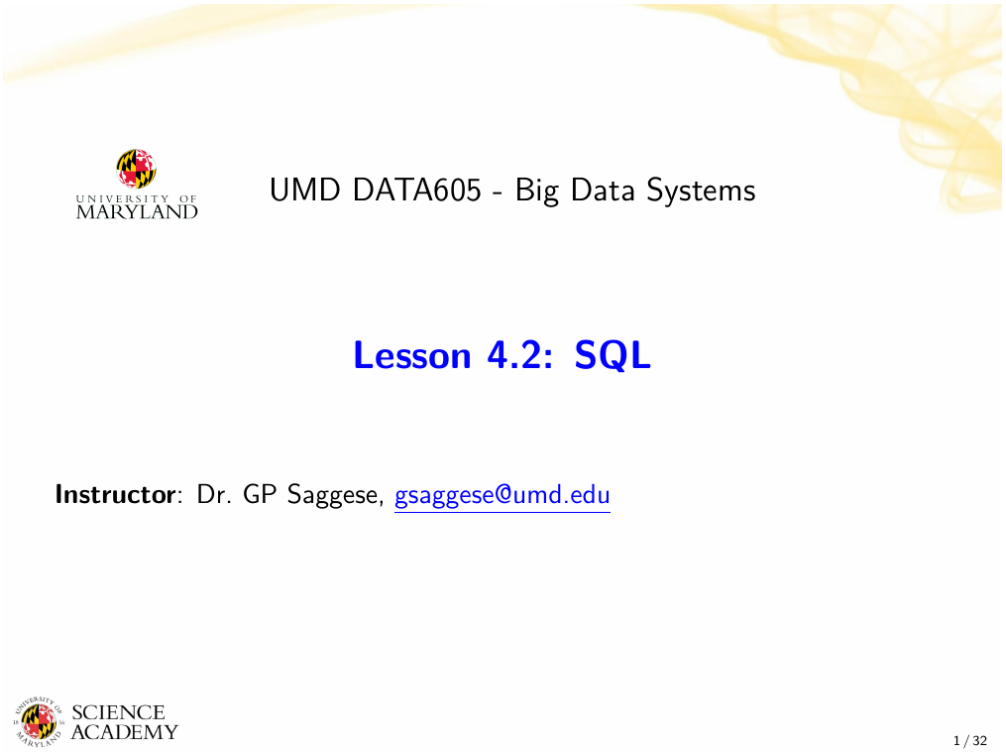
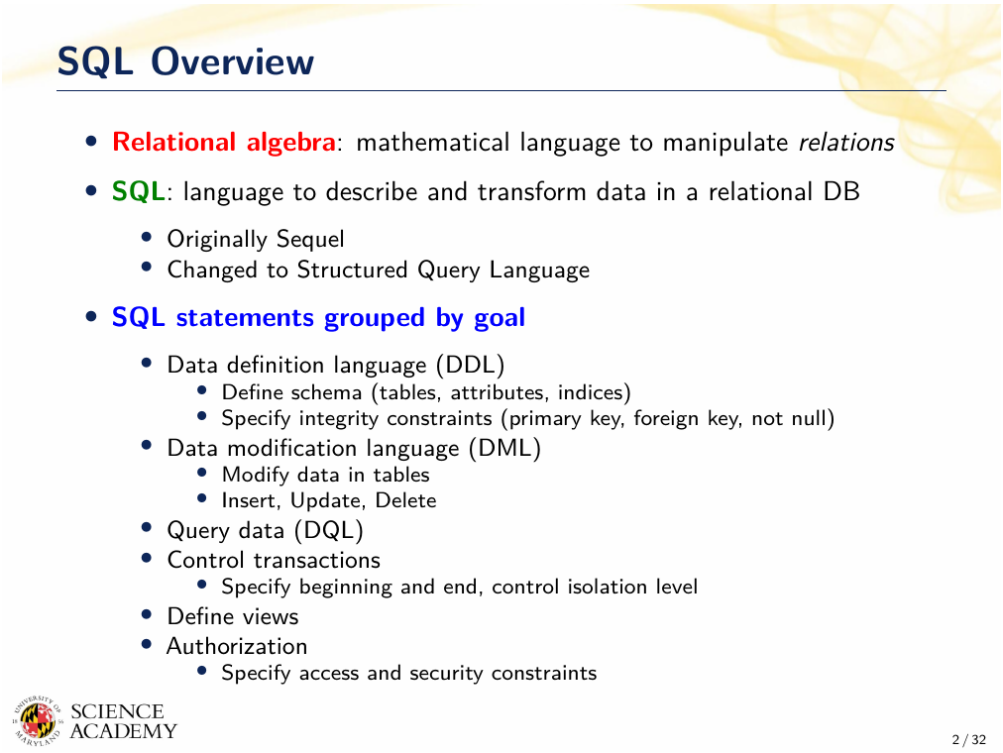
# Lesson 4.2: SQL



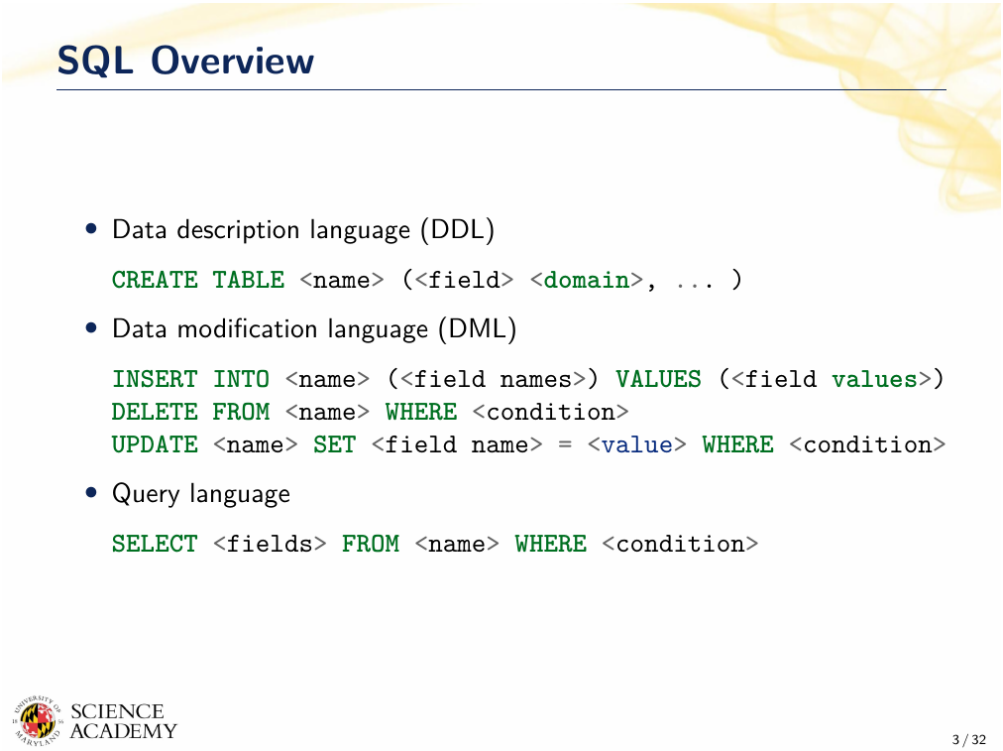
# 2 / 32: SQL Overview




- **Relational algebra**: This is a *mathematical framework* used to work with data organized in tables, known as relations. It provides a set of operations that allow you to manipulate and retrieve data in a structured way. Think of it as the theoretical foundation for how databases manage and process data.
- **SQL**: This stands for Structured Query Language, a powerful tool used to interact with relational databases. Originally called "Sequel," SQL allows users to describe, manipulate, and query data stored in databases. It's the standard language for managing and retrieving data in relational database systems.
- **SQL statements grouped by goal**:
  - **Data definition language (DDL)**: This part of SQL is used to define and manage the structure of the database. It includes creating tables, defining columns, and setting up rules like primary keys (unique identifiers for records) and foreign keys (links between tables).
  - **Data modification language (DML)**: These commands are used to change the data within the database. You can add new data (Insert), change existing data (Update), or remove data (Delete).
  - **Query data (DQL)**: This involves retrieving data from the database, typically using the SELECT statement to specify what data you want to see.
  - **Control transactions**: Transactions are sequences of operations performed as a single unit. SQL allows you to manage these transactions, ensuring data integrity and consistency, by marking the start and end of transactions and setting isolation levels to control how transactions interact.
  - **Define views**: Views are virtual tables created by a query. They allow you to simplify

complex queries and present data in a specific format without altering the actual tables.
  - **Authorization**: This involves setting permissions to control who can access or modify data, ensuring security and privacy within the database.

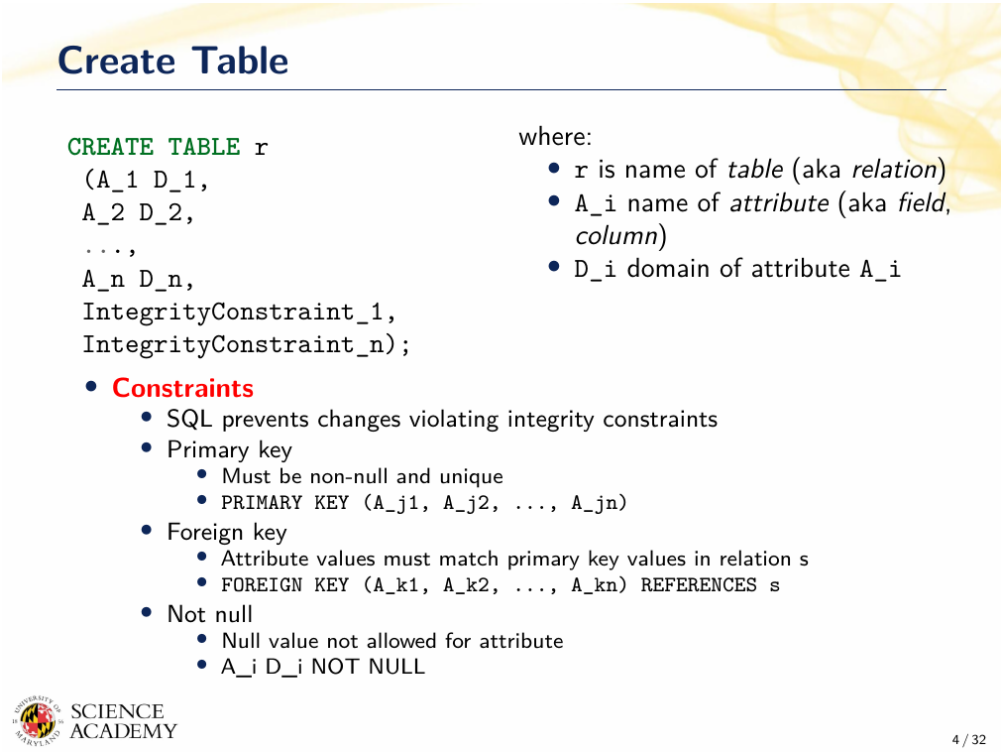
## 3 / 32: SQL Overview




- **SQL Overview**
  - **Data description language (DDL)**
    - DDL is used to define and manage database structures. The `CREATE TABLE` command is a fundamental part of DDL. It allows you to create a new table in the database by specifying the table's name and its fields (or columns) along with their data types, known as domains. For example, you might create a table for storing customer information with fields like `CustomerID`, `Name`, and `Email`.
  - **Data modification language (DML)**
    - DML is used for managing data within the database. The `INSERT INTO` command adds new records to a table. You specify the table name, the fields you want to populate, and the corresponding values. The `DELETE FROM` command removes records from a table based on a specified condition, allowing you to manage data by removing unnecessary or outdated entries. The `UPDATE` command modifies existing records, setting new values for specified fields where certain conditions are met. This is useful for keeping data current and accurate.
  - **Query language**
    - The `SELECT` statement is the core of SQL's query language, allowing you to retrieve data from one or more tables. You specify the fields you want to retrieve, the table name, and any conditions that filter the results. This is essential for analyzing and reporting data, as it enables you to extract specific information from large datasets

efficiently.

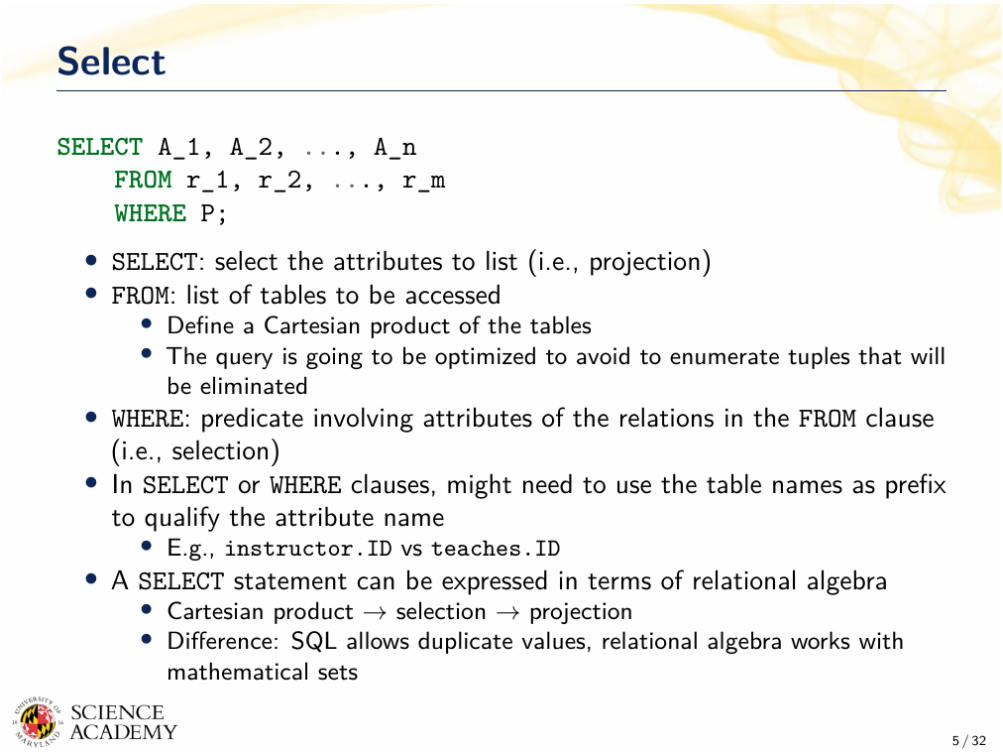
## 4 / 32: Create Table




- **Create Table Syntax**
  - The `CREATE TABLE` statement is used to define a new table in a database. This is a fundamental operation in SQL, which stands for Structured Query Language, used for managing and manipulating relational databases.
  - The syntax includes specifying the table name (`r`) and defining its structure with attributes (`A_1, A_2, ..., A_n`) and their respective data types (`D_1, D_2, ..., D_n`).
- **Table and Attributes**
  - `r`: This is the name of the table, also known as a *relation* in database terminology. It is important to choose a meaningful name that reflects the data stored in the table.
  - `A_i`: These are the names of the attributes, which are also referred to as *fields* or *columns*. Each attribute represents a specific piece of data within the table.
  - `D_i`: This represents the domain or data type of the attribute `A_i`. Common data types include integers, strings, and dates, which define what kind of data can be stored in each column.
- **Constraints**
  - Constraints are rules applied to table columns to ensure data integrity and accuracy.
  - **Primary Key**
    - A primary key is a unique identifier for each record in a table. It must be non-null and unique, ensuring that each entry can be uniquely identified.
    - The syntax `PRIMARY KEY (A_j1, A_j2, ..., A_jn)` specifies which attribute(s) serve as the primary key.
  - **Foreign Key**

    - A foreign key is used to link two tables together. It ensures that the values in one table match values in another table's primary key, maintaining referential integrity.
    - The syntax `FOREIGN KEY (A_k1, A_k2, ..., A_kn) REFERENCES s` indicates that the attributes must match primary key values in another table `s`.
  - **Not Null**
    - This constraint ensures that a column cannot have a null value, meaning every entry must have a value for this attribute.
    - The syntax `A_i D_i NOT NULL` is used to enforce this rule on a specific attribute.

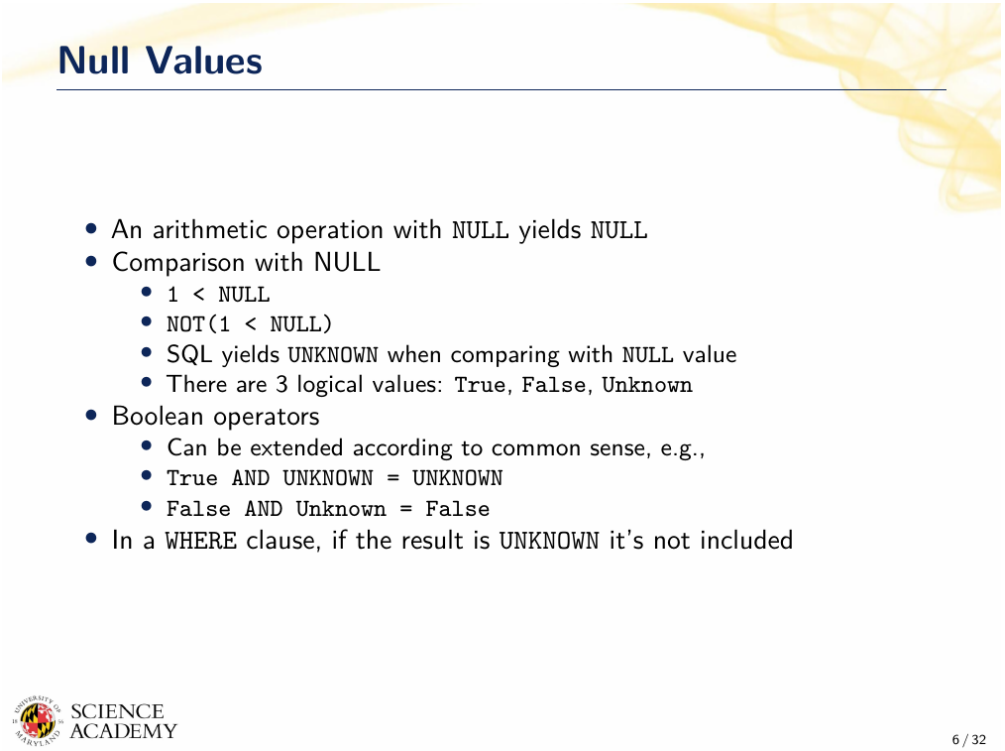
## 5 / 32: Select




- **`SELECT`:** This keyword is used to specify which columns or attributes you want to retrieve from the database. It's like choosing specific data points from a larger dataset. This process is known as *projection* because you're projecting only certain columns from the entire table.
- **`FROM`:** Here, you list the tables from which you want to retrieve data. When you mention multiple tables, SQL initially considers all possible combinations of rows from these tables, known as a Cartesian product. However, don't worry about inefficiency; SQL optimizes the query to avoid unnecessary combinations that won't be used.
- **`WHERE`:** This clause is used to filter the data based on specific conditions. It acts like a sieve, allowing only the rows that meet the criteria to pass through. This is known as *selection*.
- **Table Name Prefixes:** Sometimes, different tables might have columns with the same name. To avoid confusion, you can prefix the column name with the table name, like `instructor.ID` or `teaches.ID`.
- **Relational Algebra Connection:** A `SELECT` statement can be broken down into relational

algebra operations: Cartesian product, selection, and projection. However, a key difference is that SQL can return duplicate rows, while relational algebra typically deals with sets, which don't allow duplicates.

## 6 / 32: Null Values




- **Null Values**
  - When you perform any arithmetic operation with a `NULL` value, the result will always be `NULL`. This is because `NULL` represents an unknown or missing value, so any calculation involving it remains indeterminate.
- **Comparison with NULL**
  - If you try to compare a number, like `1`, with `NULL`, the result is not straightforward. For example, `1 < NULL` doesn't return `True` or `False`; instead, it returns `UNKNOWN`.
  - Similarly, negating the comparison, such as `NOT(1 < NULL)`, also results in `UNKNOWN`.
  - In SQL, when you compare anything with `NULL`, the outcome is `UNKNOWN` because `NULL` is not a value but a placeholder for missing information.
  - SQL logic includes three possible outcomes for comparisons: `True`, `False`, and `Unknown`.
- **Boolean operators**
  - Boolean logic in SQL can be extended to handle `UNKNOWN` values. For instance, if you have `True AND UNKNOWN`, the result is `UNKNOWN` because the unknown part makes the whole expression uncertain.
  - Conversely, `False AND UNKNOWN` results in `False` because the `False` value dominates the outcome regardless of the unknown part.
- **In a `WHERE` clause**
  - When using a `WHERE` clause in SQL, if the condition evaluates to `UNKNOWN`, the row is not included in the result set. This is because SQL only includes rows where the condition

is definitively `True`.

## 7 / 32: Group by Query

### Group by Query

- The attributes in `GROUP BY` are used to form groups
  - Tuples with the same value on all attributes are placed in one group
- Any attribute that is not in the `GROUP BY` can appear in the `SELECT` clause only as argument of aggregate function

```
SELECT dept_name, AVG(salary)
    FROM instructor
    GROUP BY dept_name;

-- Error.
SELECT dept_name, salary
    FROM instructor
    GROUP BY dept_name;
```

- `salary` is not in `GROUP BY` so it must be in an aggregate function

| *ID* | *name* | *dept_name* | *salary* |
|---|---|---|---|
| 76766 | Crick | Biology | 72000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 83821 | Brandt | Comp. Sci. | 92000 |
| 98345 | Kim | Elec. Eng. | 80000 |
| 12121 | Wu | Finance | 90000 |
| 76543 | Singh | Finance | 80000 |
| 32343 | El Said | History | 60000 |
| 58583 | Califieri | History | 62000 |
| 15151 | Mozart | Music | 40000 |
| 33456 | Gold | Physics | 87000 |
| 22222 | Einstein | Physics | 95000 |

| *dept_name* | *avg_salary* |
|---|---|
| Biology | 72000 |
| Comp. Sci. | 77333 |
| Elec. Eng. | 80000 |
| Finance | 85000 |
| History | 61000 |
| Music | 40000 |
| Physics | 91000 |

SCIENCE ACADEMY

7 / 32

- **Group by Query**: This slide explains how the `GROUP BY` clause works in SQL queries. The `GROUP BY` clause is used to arrange identical data into groups. This is particularly useful when you want to perform aggregate calculations on subsets of data.
- **Attributes in `GROUP BY`**: When you use `GROUP BY`, the attributes you specify are used to form groups. All rows with the same values in these attributes are grouped together. For example, if you group by `dept_name`, all instructors from the same department will be grouped together.
- **Attributes in `SELECT` Clause**: If an attribute is not included in the `GROUP BY` clause, it can only appear in the `SELECT` clause if it is used with an aggregate function like `AVG`, `SUM`, `COUNT`, etc. This is because SQL needs to know how to handle the non-grouped data.
- **Example Query**: The first SQL example shows a correct usage where `dept_name` is grouped, and `AVG(salary)` is calculated for each department. This is a valid query because `salary` is used within an aggregate function.
- **Error Example**: The second SQL example demonstrates an error. Here, `salary` is included in the `SELECT` clause without being part of an aggregate function or the `GROUP BY` clause. This is not allowed because SQL cannot determine which salary value to display for each department group.
- **Important Note**: Always ensure that any attribute not in the `GROUP BY` clause is used with an aggregate function in the `SELECT` clause to avoid errors. This is a common mistake when writing SQL queries involving grouping and aggregation.

# 8 / 32: Having

## Having

- State a condition that applies to groups instead of tuples (like WHERE)
- Any attribute in the HAVING clause must appear in the GROUP BY clause
- E.g., find departments with avg salary of instructors $> 42$k

```
SELECT dept_name, AVG(salary) AS avg_salary
    FROM instructor
    GROUP BY dept_name
    HAVING AVG(salary) > 42000;
```

- How does it work
  - FROM is evaluated to create a relation
  - (optional) WHERE is used to filter
  - GROUP BY collects tuples into groups
  - (optional) HAVING is applied to each group and groups are filtered
  - SELECT generates tuples of the results, applying aggregate functions to get a single result for each group

SCIENCE ACADEMY

8 / 32

- **Having**
  - The `HAVING` clause is used to filter data at the group level, unlike the `WHERE` clause, which filters individual rows or tuples. This means that `HAVING` is applied after the data has been grouped.
  - It's important to note that any attribute used in the `HAVING` clause must also be included in the `GROUP BY` clause. This ensures that the filtering condition is applied to the correct group of data.
  - For example, if you want to find departments where the average salary of instructors is greater than $42,000, you would use the `HAVING` clause to filter these groups after calculating the average salary for each department.
- **How does it work**
  - The SQL query execution starts with the `FROM` clause, which identifies the tables involved and creates a relation from them.
  - If a `WHERE` clause is present, it filters the rows based on specified conditions before any grouping occurs.
  - The `GROUP BY` clause then collects these filtered rows into groups based on one or more columns.
  - The `HAVING` clause is applied to these groups, allowing you to filter out entire groups based on aggregate conditions.
  - Finally, the `SELECT` clause generates the output, applying any aggregate functions to produce a single result for each group, such as calculating averages or sums.

# 9 / 32: Nested Subqueries

## Nested Subqueries

- SQL allows using the result of a query in another query
  - Use a subquery returning one attribute (scalar subquery) where a value is used
  - Use the result of a query for set membership in the WHERE clause
  - Use the result of a query in a FROM clause :::columns ::::{.column width=60%}

```
SELECT tmp.dept_name, tmp.avg_salary
FROM (
  SELECT dept_name,
      AVG(salary) AS avg_salary
  FROM instructor
  GROUP BY dept_name) AS tmp
WHERE avg_salary > 42000
:::: ::::{.column width=40%}
```

| dept_name | avg_salary |
|---|---|
| Finance | 85000.000000000000 |

SCII ACA

9 / 32

- **Nested Subqueries**
  - In SQL, you can use the result of one query as part of another query. This is known as a *nested subquery*. It allows for more complex data retrieval by building on simpler queries.
  - **Scalar Subquery**: This is a subquery that returns a single value. You can use it in places where you would normally use a single value, like in a comparison in the `WHERE` clause.
  - **Set Membership**: You can use a subquery to check if a value is part of a set of values. This is often done using the `IN` keyword in the `WHERE` clause.
  - **FROM Clause**: You can use a subquery to create a temporary table that can be used in the `FROM` clause of another query. This is useful for organizing complex queries and breaking them into manageable parts.
- **Example Explanation**
  - The SQL example provided demonstrates using a subquery in the `FROM` clause. The subquery calculates the average salary for each department from the `instructor` table and groups the results by `dept_name`.
  - The outer query then selects departments where the average salary is greater than 42,000. The subquery is aliased as `tmp`, which acts like a temporary table for the outer query to use.
  - This approach is useful for filtering data based on aggregated results, such as finding departments with higher-than-average salaries.

# 10 / 32: With

## With

- WITH clause allows to define a temporary relation containing the results of a subquery
- It can be equivalent to a nested subqueries, but clearer
- E.g., find department with the maximum budget :::columns ::::{.column width=80%}

```
WITH max_budget(value) as (
  SELECT MAX(budget) FROM department)
  SELECT department.dept_name, budget
    FROM department, max_budget
    WHERE department.budget = max_budget.value
```

:::: ::::{.column width=20%}

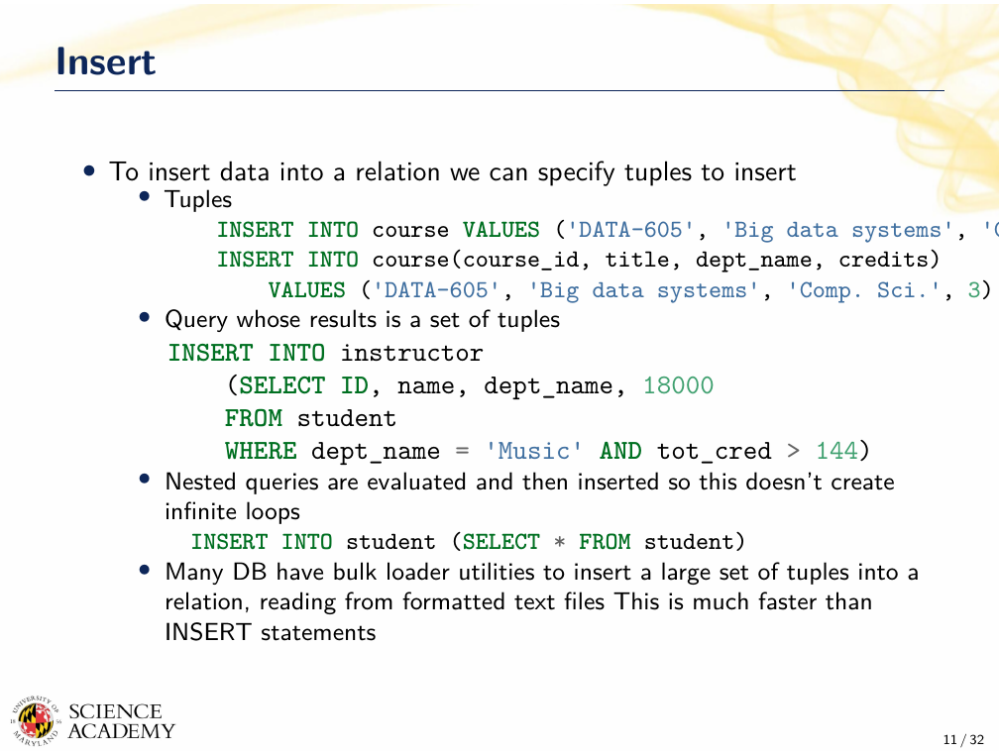
| dept_name | budget |
|---|---|
| Finance | 120000.00 |

:::: :::



- **WITH Clause**: The WITH clause in SQL is used to create a temporary table or relation that holds the results of a subquery. This can make complex queries easier to read and manage by breaking them down into simpler parts. Instead of writing a long, nested subquery, you can define it once with WITH and then reference it in your main query.
- **Clarity Over Nested Subqueries**: Using the WITH clause can be more readable than using nested subqueries. It allows you to name the result of a subquery, making your SQL code more understandable and maintainable.
- **Example - Finding Maximum Budget**: In the example provided, the WITH clause is used to find the department with the maximum budget. The subquery `SELECT MAX(budget) FROM department` calculates the maximum budget and stores it in a temporary relation called `max_budget`. The main query then selects the department name and budget where the department's budget matches this maximum value.
- **Code Explanation**: The SQL code first defines `max_budget` using the WITH clause. It then selects the department name and budget from the `department` table where the budget equals the maximum budget found. This approach simplifies the query by separating the logic into distinct parts.
- **Visual Aid**: The image on the right likely provides a visual representation of the query's logic or the database schema, helping to further clarify how the WITH clause is applied in this context.

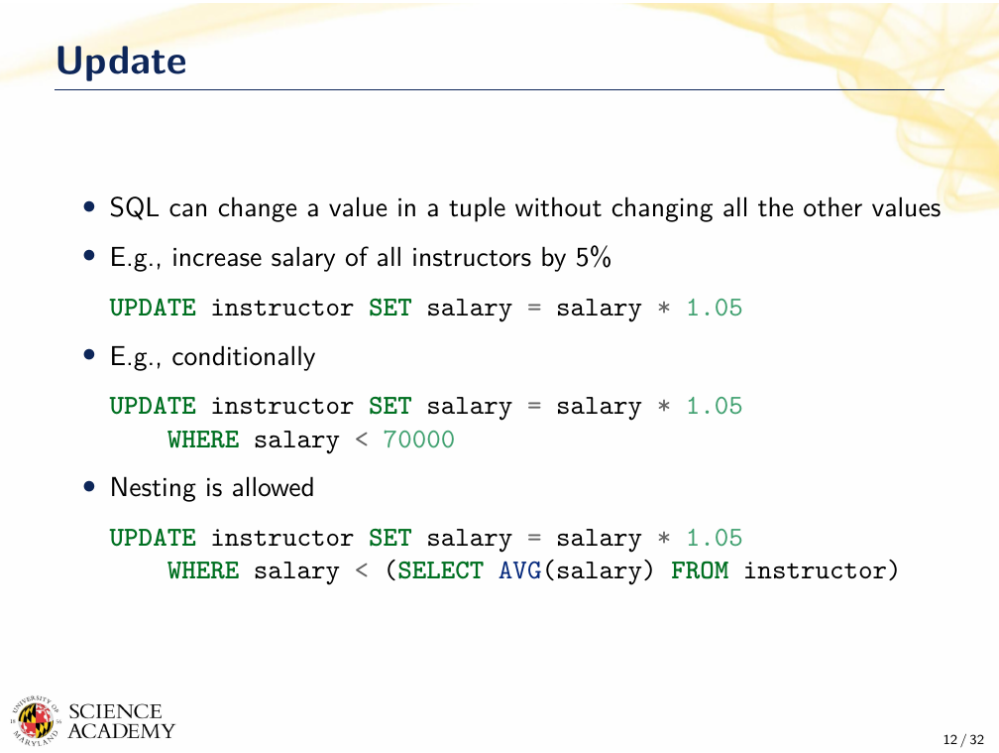
# 11 / 32: Insert




- **Insert Data into a Relation**
  - When we want to add new data to a database table, we use the *INSERT* command. This command allows us to specify the exact data, or *tuples*, we want to add.
  - **Tuples**
    - A tuple is essentially a single row of data. In the example provided, the SQL command is adding a new course to the 'course' table. The command specifies the course ID, title, department name, and credits. This is a straightforward way to add a single row of data.
    - The second example shows how you can specify the column names explicitly. This is useful when you want to ensure that the data is inserted into the correct columns, especially if the table has many columns.
  - **Query Result as a Set of Tuples**
    - Sometimes, instead of inserting data manually, you might want to insert data that is the result of a query. The example shows how to insert data into the 'instructor' table by selecting certain students from the 'student' table who meet specific criteria (e.g., students from the Music department with more than 144 credits).
  - **Nested Queries**
    - Nested queries are queries within queries. The example shows a nested query that selects all students and attempts to insert them back into the 'student' table. However, databases are designed to handle such operations without creating infinite loops, ensuring that the operation completes successfully.
  - **Bulk Loader Utilities**
    - For large datasets, using individual *INSERT* statements can be inefficient. Many databases offer bulk loader utilities that can read data from formatted text files and

insert it into the database much faster. This is particularly useful when dealing with big data, where the volume of data is too large for manual insertion.

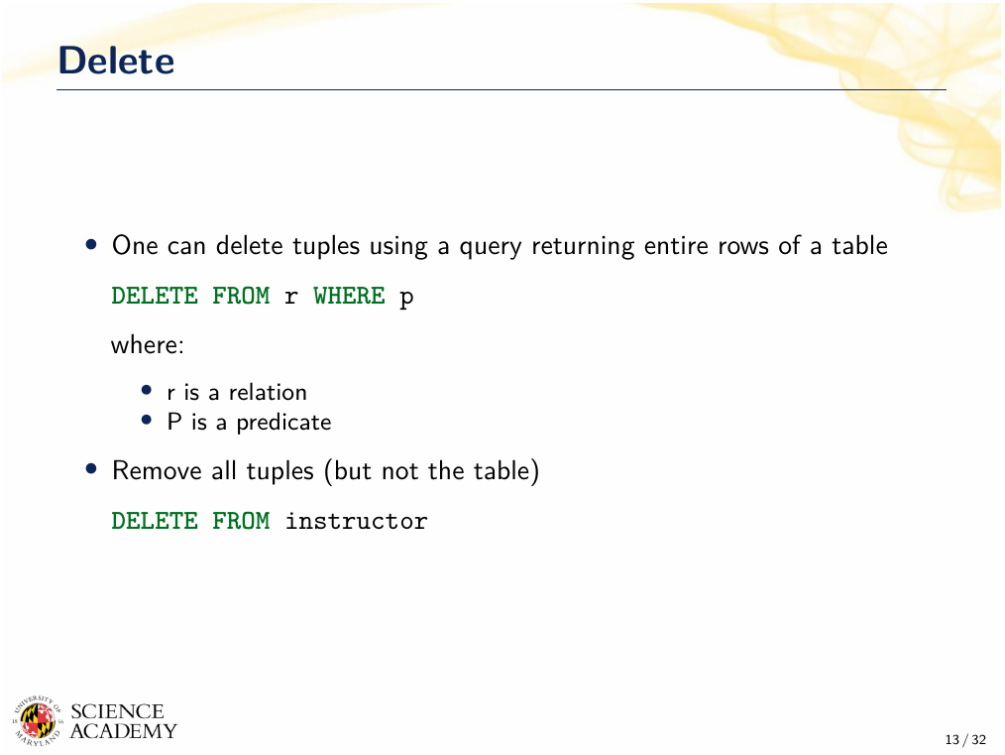
## 12 / 32: Update




- **Update**
  In databases, the *UPDATE* command is used to modify existing data within a table. This is particularly useful when you need to change specific values without altering the entire dataset. For instance, if you want to adjust the salary of instructors, you can do so without affecting other attributes like their names or departments.
- **SQL can change a value in a tuple without changing all the other values**
  This means you can target specific columns for updates. In our example, we focus on the salary column, leaving other columns untouched.
- **E.g., increase salary of all instructors by 5%**
  The SQL command `UPDATE instructor SET salary = salary * 1.05` demonstrates how to increase every instructor's salary by 5%. This operation multiplies each salary by 1.05, effectively giving a raise across the board.
- **E.g., conditionally**
  Sometimes, you only want to update certain rows based on a condition. The command `UPDATE instructor SET salary = salary * 1.05 WHERE salary < 70000` shows how to apply the raise only to instructors earning less than $70,000. This conditional update ensures that only specific entries are modified.
- **Nesting is allowed**
  SQL allows for more complex conditions using subqueries. For example, `UPDATE`

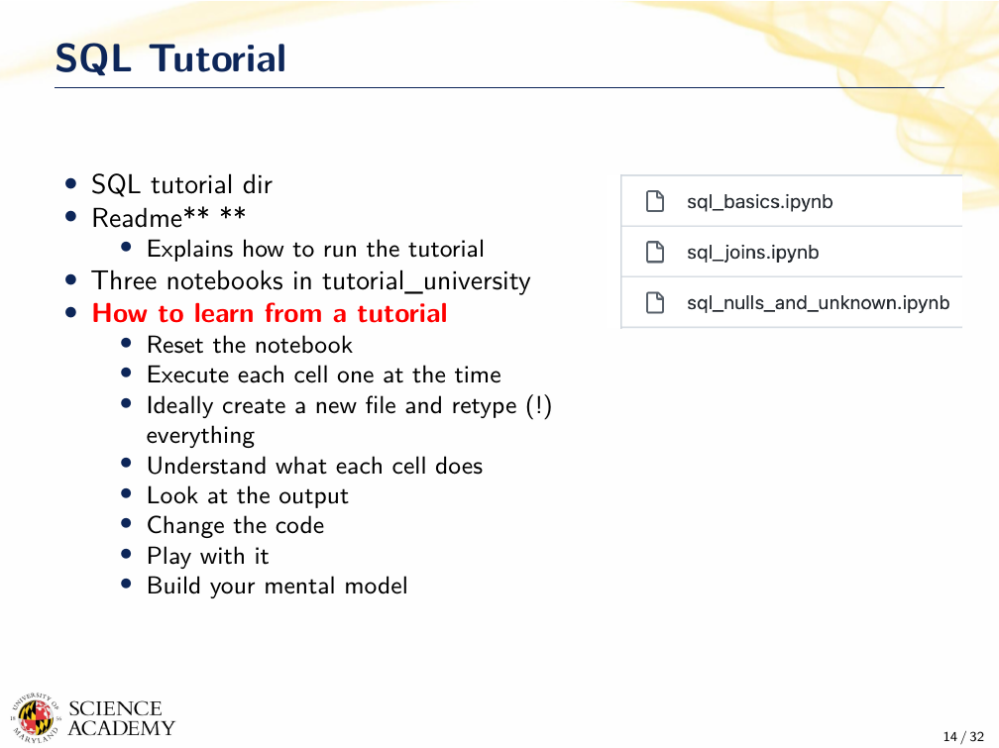
`instructor SET salary = salary * 1.05 WHERE salary < (SELECT AVG(salary) FROM instructor)` updates salaries for instructors earning below the average salary. This nested query first calculates the average salary and then applies the update conditionally, showcasing SQL's flexibility in handling complex data operations.

## 13 / 32: Delete




- **Delete**
  - The `DELETE` statement in SQL is used to remove data from a table. It allows you to delete specific rows based on a condition or even all rows if no condition is specified.
  - **One can delete tuples using a query returning entire rows of a table**
    - The basic syntax for deleting rows is `DELETE FROM r WHERE p`. Here, `r` represents the table (or relation) from which you want to delete data, and `p` is the condition (or predicate) that specifies which rows should be deleted.
    - For example, if you have a table of students and you want to delete all students who have graduated, you would specify a condition that identifies those students.
  - **Remove all tuples (but not the table)**
    - If you want to delete all rows from a table but keep the table structure intact, you can use the `DELETE FROM instructor` command without a `WHERE` clause. This will remove all data from the `instructor` table, but the table itself will remain in the database.
    - This is useful when you want to clear out old data but plan to reuse the table for new data in the future.

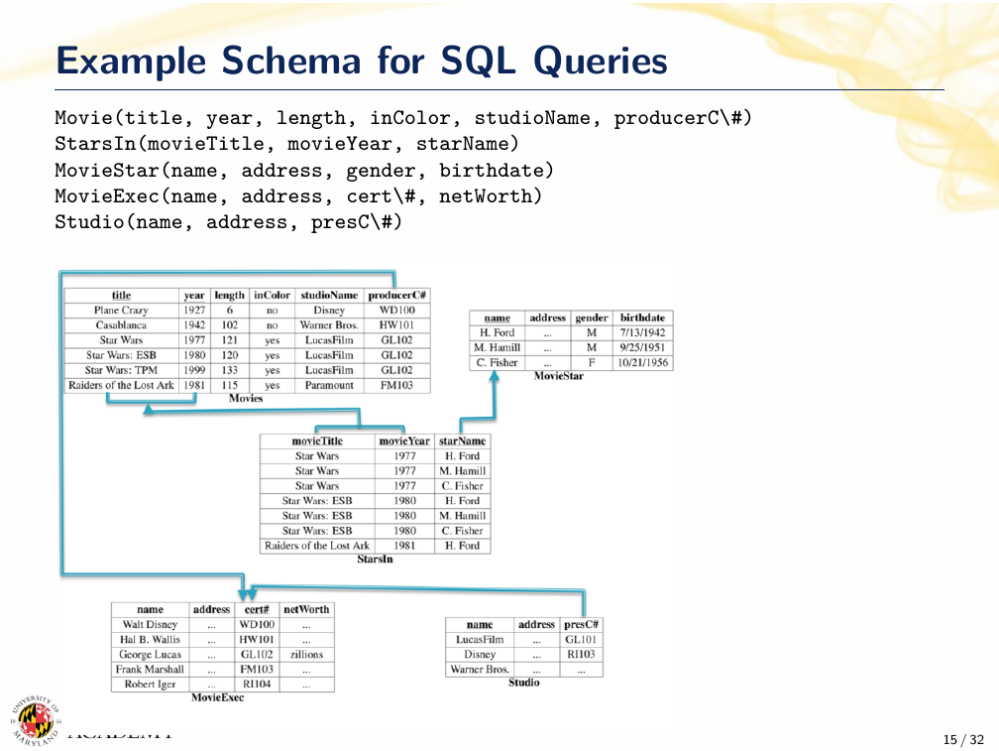
# 14 / 32: SQL Tutorial




- **SQL Tutorial Directory**
  - This is the main folder where all the materials for the SQL tutorial are stored. It acts as the starting point for anyone looking to learn SQL through this tutorial.
- **Readme**
  - The Readme file is crucial as it provides instructions on how to run the tutorial. It's the first document you should read to understand the setup and execution process.
- **Three Notebooks in tutorial_university**
  - These notebooks are part of the tutorial and contain practical exercises and examples. They are designed to help you learn SQL by doing, rather than just reading.
- **How to Learn from a Tutorial**
  - **Reset the Notebook**: Start fresh to ensure you don't have any leftover data or errors from previous sessions.
  - **Execute Each Cell One at a Time**: This helps you focus on understanding each part of the code.
  - **Retype Everything**: By creating a new file and retyping the code, you reinforce learning through practice.
  - **Understand Each Cell**: Take the time to comprehend what each piece of code is doing.
  - **Look at the Output**: Observing the results helps you connect the code with its effects.
  - **Change the Code**: Experimenting with the code helps deepen your understanding.
  - **Play with It**: Engaging with the code in a playful manner can lead to new insights and a better grasp of concepts.
  - **Build Your Mental Model**: Developing a mental framework of how SQL works will aid in solving problems and applying knowledge in real-world scenarios.

- **Movie Database Example (Optional)**
  - This is an additional resource that provides a practical example of using SQL with a movie database. It's optional but can be a valuable exercise for applying what you've learned.

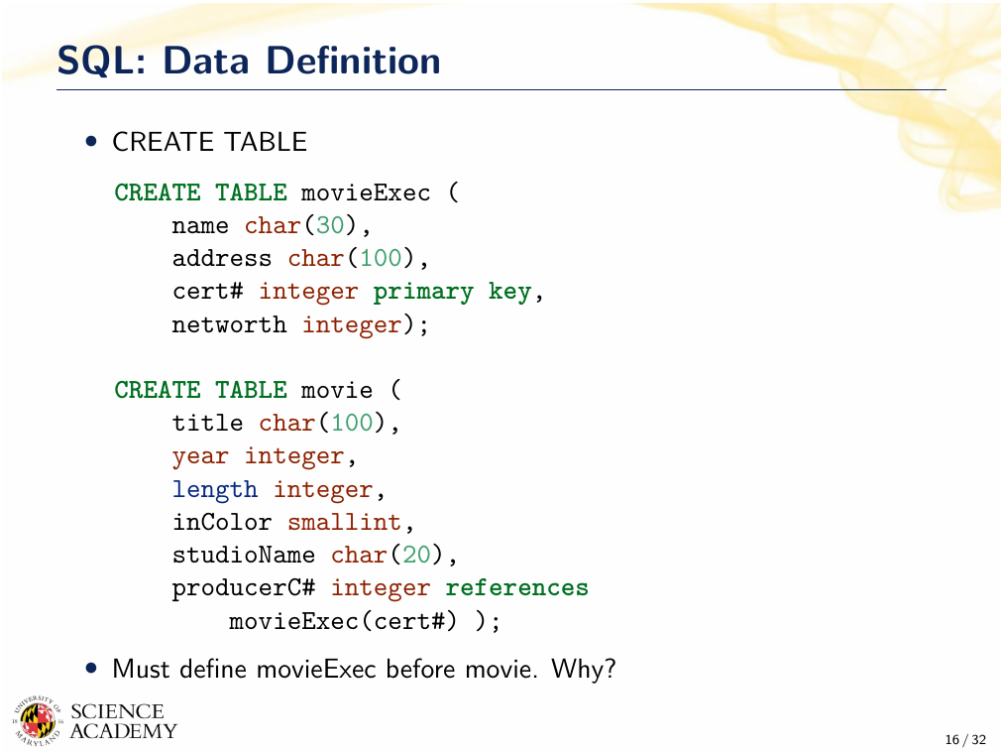
## 15 / 32: Example Schema for SQL Queries




- **Example Schema for SQL Queries**: This slide presents a schema, which is essentially a blueprint or structure for organizing data in a database. It defines how data is stored, the relationships between different data entities, and the constraints on the data.
- **Movie Table**: This table stores information about movies. It includes fields like *title*, *year*, *length*, *inColor* (indicating if the movie is in color), *studioName* (the studio that produced the movie), and *producerC#* (a unique identifier for the producer).
- **StarsIn Table**: This table captures the relationship between movies and the stars who acted in them. It includes *movieTitle*, *movieYear*, and *starName*. This helps in linking actors to the movies they have starred in.
- **MovieStar Table**: This table contains details about movie stars, including their *name*, *address*, *gender*, and *birthdate*. It helps in storing personal information about actors.
- **MovieExec Table**: This table is for movie executives, capturing their *name*, *address*, *cert#* (a unique certification number), and *netWorth*. It provides insights into the people managing the movie industry.
- **Studio Table**: This table holds information about movie studios, including *name*, *address*, and *presC#* (a unique identifier for the studio president). It helps in organizing data about the companies producing films.

- **Context**: Understanding this schema is crucial for writing SQL queries, as it defines the structure of the data you will be querying. Each table represents a different aspect of the movie industry, and the relationships between them allow for complex queries to extract meaningful insights.

## 16 / 32: SQL: Data Definition




- **SQL: Data Definition**
  - **CREATE TABLE**
    - The `CREATE TABLE` statement is used to define a new table in a database. In this example, two tables are being created: `movieExec` and `movie`.
    - **`movieExec` Table**: This table includes columns for `name`, `address`, `cert#`, and `networth`. The `cert#` column is designated as the primary key, which means it uniquely identifies each record in the table.
    - **`movie` Table**: This table includes columns for `title`, `year`, `length`, `inColor`, `studioName`, and `producerC#`. The `producerC#` column is a foreign key that references the `cert#` column in the `movieExec` table. This establishes a relationship between the two tables, linking each movie to a specific movie executive.
  - **Must define `movieExec` before `movie`. Why?**
    - The `movieExec` table must be defined before the `movie` table because the `movie` table contains a foreign key (`producerC#`) that references the `cert#` column in the `movieExec` table. In SQL, a table must exist before another table can reference it. This ensures that the database knows about the structure and constraints of the referenced table, allowing it to enforce referential integrity.

# 17 / 32: SQL: Data Manipulation

## SQL: Data Manipulation

- INSERT

```
INSERT INTO StarsIn values('King Kong', 2005, 'Naomi Watts
```

```
INSERT INTO StarsIn(starName, movieTitle, movieYear)
    values('Naomi Watts', 'King Kong', 2005);
```

- DELETE

```
DELETE FROM movies WHERE movieYear < 1980;
```

  - Syntax is fine, but this command will be rejected. Why?

```
DELETE FROM movies
    WHERE length < (SELECT avg(length) FROM movies);
```

  - Problem: as we delete tuples, the average length changes
  - Solution:
    - First, compute avg length and find all tuples to delete
    - Next, delete all tuples found above (without recomputing avg or retesting the tuples)

SCIENCE ACADEMY

17 / 32

- **INSERT**
  - The `INSERT` command is used to add new records to a table in a database. In the first example, we are adding a new entry to the `StarsIn` table with the values 'King Kong', 2005, and 'Naomi Watts'. This assumes the table's columns are in the order of movie title, year, and star name.
  - The second example shows a more explicit way to insert data by specifying the column names. This is useful when you want to ensure that the data is inserted into the correct columns, especially if the table structure changes or if you are not inserting values for all columns.
- **DELETE**
  - The `DELETE` command removes records from a table. The first example attempts to delete all movies released before 1980. However, the slide notes that this command will be rejected. This could be due to constraints like foreign keys or permissions that prevent deletion.
  - The second example aims to delete movies with a length less than the average length of all movies. The problem here is that as movies are deleted, the average length recalculates, potentially altering which movies should be deleted. The solution involves calculating the average length first, identifying all movies to delete, and then deleting them in one go without recalculating the average. This ensures consistency and avoids logical errors.

## SQL: Data Manipulation

- UPDATE
  - Increase all movieExec netWorth's over 100,000 USD by 6%, all other accounts receive 5%
  - Write two update statements:

```
UPDATE movieExec SET netWorth = netWorth * 1.06
    WHERE netWorth > 100000;
UPDATE movieExec SET netWorth = netWorth * 1.05
    WHERE netWorth <= 100000;
```

  - The order is important
  - Can be done better using the case statement

```
UPDATE movieExec SET netWorth =
    CASE
        WHEN netWorth > 100000 THEN netWorth * 1.06
        WHEN netWorth <= 100000 THEN netWorth * 1.05
    END;
```

SCIENCE ACADEMY

18 / 32

- **SQL: Data Manipulation**
  - **UPDATE**: The `UPDATE` statement in SQL is used to modify existing records in a table. In this context, we are focusing on updating the `netWorth` of movie executives based on certain conditions.
  - **Increase all movieExec netWorth's over 100,000 USD by 6%, all other accounts receive 5%**: This task involves adjusting the `netWorth` of movie executives. If an executive's `netWorth` is over 100,000 USD, it should be increased by 6%. If it is 100,000 USD or less, it should be increased by 5%.
  - **Write two update statements**: The slide provides two separate SQL statements to achieve this task. The first statement updates the `netWorth` for those with more than 100,000 USD, and the second statement updates the rest.

```sql
UPDATE movieExec SET netWorth = netWorth * 1.06
    WHERE netWorth > 100000;
```

```sql
UPDATE movieExec SET netWorth = netWorth * 1.05
    WHERE netWorth <= 100000;
```

  - **The order is important**: The order of these statements is crucial because executing them in the wrong order could lead to incorrect results. If the second statement is executed first, it would incorrectly apply a 5% increase to all records, including those that should receive a 6% increase.
  - **Can be done better using the case statement**: The slide suggests a more efficient approach using a `CASE` statement within a single `UPDATE` command. This method is more concise and reduces the risk of errors associated with executing multiple statements in the correct order.

```
UPDATE movieExec SET netWorth =
    CASE
        WHEN netWorth > 100000 THEN netWorth * 1.06
        WHEN netWorth <= 100000 THEN netWorth * 1.05
    END;
```

– *Important Point*: Using a `CASE` statement not only simplifies the code but also ensures that all updates are handled in one atomic operation, reducing the potential for mistakes and improving maintainability.

## 19 / 32: SQL Single Table Queries

### SQL Single Table Queries

- Movies produced by Disney in 1990: note the *rename*

```
SELECT m.title, m.year
    FROM movie m
    WHERE m.studioname = 'disney' AND m.year = 1990;
```

- The SELECT clause can contain expressions

```
SELECT title || ' (' || to_char(year) || ')' AS titleyear
```

```
SELECT 2014 - year
```

- The WHERE clause support a large number of different predicates and combinations thereof

```
year BETWEEN 1990 and 1995
```

```
title LIKE 'star wars%'
```

```
title LIKE 'star wars _'
```

SCIENCE ACADEMY

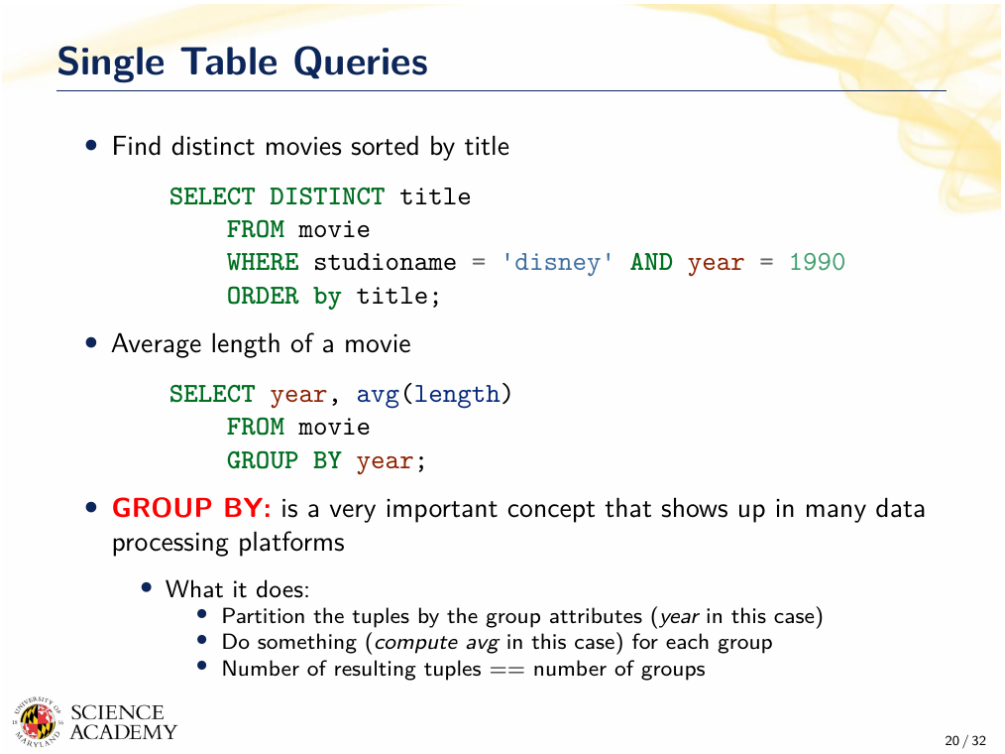
19 / 32

- **Movies produced by Disney in 1990: note the *rename***
  - The SQL query provided is designed to retrieve a list of movies produced by Disney in the year 1990. The `SELECT` statement specifies that we want to see the `title` and `year` of each movie. The `FROM` clause indicates that this data is being pulled from a table named `movie`, which is given an alias `m` for convenience. The `WHERE` clause filters the results to only include movies where the `studioname` is 'disney' and the `year` is 1990. This is a basic example of a single table query that uses filtering to narrow down results.
- **The SELECT clause can contain expressions**
  - SQL allows for expressions within the `SELECT` clause to manipulate or format the data being retrieved. For example, the expression `title || ' (' || to_char(year) || ')' AS titleyear` concatenates the movie title with its year in parentheses, creating a new column named `titleyear`. This is useful for creating more readable outputs directly from the query.
  - Another example is `SELECT 2014 - year`, which calculates the age of the movie by subtracting its release year from 2014. This demonstrates how SQL can be used to

perform calculations directly within the query.

- **The WHERE clause supports a large number of different predicates and combinations thereof**
  - The `WHERE` clause is powerful and flexible, allowing for various conditions to filter data. For instance, `year BETWEEN 1990 and 1995` filters movies released within this range of years. This is a straightforward way to specify a range condition.
  - The `LIKE` operator is used for pattern matching. For example, `title LIKE 'star wars%'` finds titles that start with "star wars", while `title LIKE 'star wars _'` looks for titles that start with "star wars" followed by exactly one additional character. These examples show how `LIKE` can be used to search for specific patterns in text data.

## 20 / 32: Single Table Queries




- **Find distinct movies sorted by title**
  - This SQL query is designed to retrieve a list of unique movie titles from a database. It specifically looks for movies produced by Disney in the year 1990.
  - The `SELECT DISTINCT` clause ensures that each movie title appears only once in the results, even if there are duplicates in the database.
  - The `WHERE` clause filters the results to include only those movies where the studio name is 'disney' and the release year is 1990.
  - Finally, the `ORDER BY title` clause sorts the resulting list of movie titles alphabetically, making it easier to read and analyze.
- **Average length of a movie**
  - This query calculates the average length of movies for each year.
  - The `SELECT` statement retrieves the year and the average length of movies released in that year.

- The `GROUP BY year` clause is crucial here as it organizes the data into groups based on the year each movie was released.
- The `avg(length)` function computes the average movie length for each group of movies released in the same year.

- **GROUP BY:** is a very important concept that shows up in many data processing platforms
  - **What it does:**
    * It partitions the data into groups based on specified attributes, which is *year* in this example.
    * For each group, it performs a specified operation, such as computing the average length of movies.
    * The number of resulting rows in the output corresponds to the number of unique groups formed, which in this case is the number of distinct years in the dataset.

## 21 / 32: Single Table Queries

### Single Table Queries

- Find movie with the maximum length

```
SELECT title, year
    FROM movie
    WHERE movie.length = (SELECT max(length) FROM movie);
```

  - The smaller "subquery" is called a "nested subquery"
- Find movies with at most 5 stars: an example of a correlated subquery

```
SELECT *
    FROM movies m
    WHERE 5 >= (SELECT count(*)
                FROM starsIn si
                WHERE si.title = m.title AND
                      si.year = m.year);
```

  - The "inner" subquery counts the number of actors for that movie.

SCIENCE ACADEMY

21 / 32

- **Single Table Queries**
- **Find movie with the maximum length**

```
SELECT title, year
    FROM movie
    WHERE movie.length = (SELECT max(length) FROM movie);
```

  - This query is designed to find the movie with the longest duration in the database.
  - The main part of the query selects the `title` and `year` from the `movie` table.
  - The condition `WHERE movie.length = (SELECT max(length) FROM movie)` uses a *nested subquery* to determine the maximum length of any movie in the table.

- A *nested subquery* is a query within another query. Here, it calculates the maximum length and uses that value to filter the main query.
- This approach is efficient for finding a single record that meets a specific criterion, like the longest movie.

- **Find movies with at most 5 stars: an example of a correlated subquery**

```
SELECT *
    FROM movies m
    WHERE 5 >= (SELECT count(*)
                FROM starsIn si
                WHERE si.title = m.title AND
                      si.year = m.year);
```

- This query retrieves movies that have five or fewer actors associated with them.
- The main query selects all columns from the `movies` table.
- The condition `5 >= (SELECT count(*) FROM starsIn si WHERE si.title = m.title AND si.year = m.year)` uses a *correlated subquery*.
- A *correlated subquery* is a subquery that depends on the outer query for its values. Here, it counts the number of actors (`starsIn`) for each movie.
- This technique is useful for filtering results based on related data, such as counting related records in another table.

## 22 / 32: Single Table Queries

### Single Table Queries

- Rank movies by their length

```
SELECT title, year,
    (SELECT count(*)
    FROM movies m2
    WHERE m1.length <= m2.length) AS rank
    FROM movies m1;
```

  - Key insight: A movie is ranked 5th if there are exactly 4 movies with longer length.
  - Most database systems support some sort of a *rank* keyword for doing this
  - The above query doesn't work in presence of ties, etc.

- Set operations

```
SELECT name
    FROM movieExec
union/intersect/minus
    SELECT name FROM
        movieStar
```

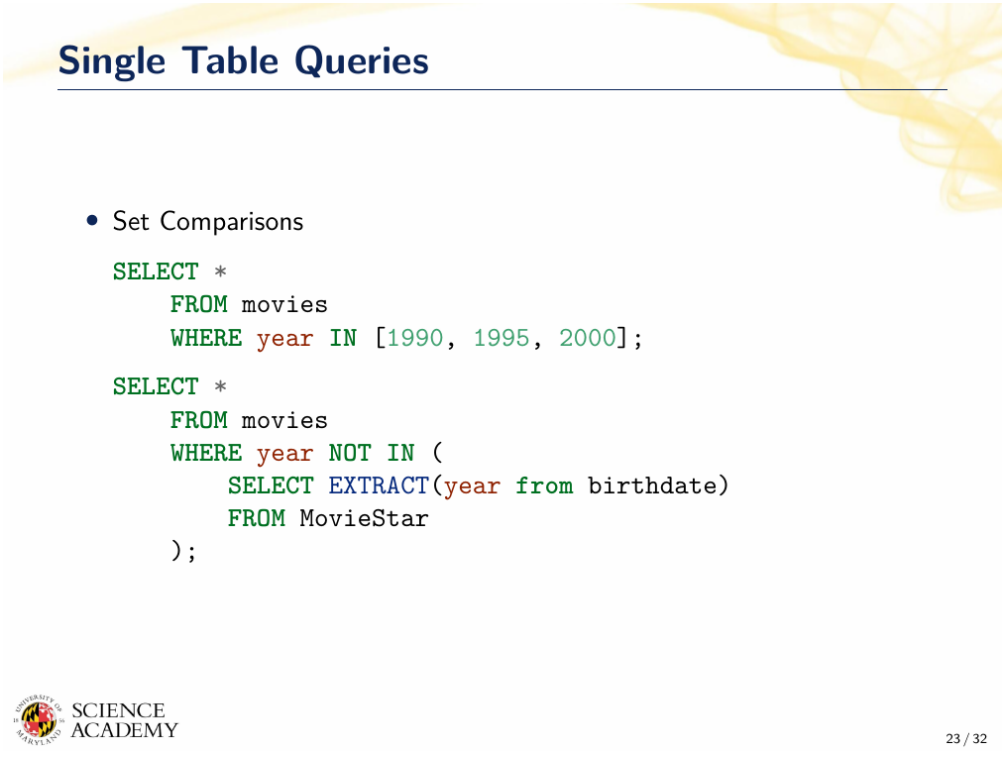
SCIENCE ACADEMY

22 / 32

- **Single Table Queries**
  - **Rank movies by their length**

    - The SQL query provided is designed to rank movies based on their length. It does this by comparing each movie's length to all other movies in the database.
    - The subquery `(SELECT count(*) FROM movies m2 WHERE m1.length <= m2.length)` counts how many movies have a length greater than or equal to the current movie (`m1`). This count effectively gives the rank of the movie.
    - **Key Insight**: If a movie is ranked 5th, it means there are exactly 4 movies that are longer than it.
    - Many modern databases have built-in functions like `RANK()` that simplify this process and handle ties more gracefully.
    - The provided query does not handle ties well, meaning if two movies have the same length, they might not be ranked correctly.
- **Set Operations**
  - The SQL snippet demonstrates how to use set operations to compare two lists of names from different tables.
  - **Union**: Combines results from both tables, removing duplicates, to show all unique names.
  - **Intersect**: Finds common names that appear in both `movieExec` and `movieStar`.
  - **Minus** (or **Except** in some databases): Shows names that are in `movieExec` but not in `movieStar`.
  - These operations are useful for comparing datasets and finding relationships or differences between them.

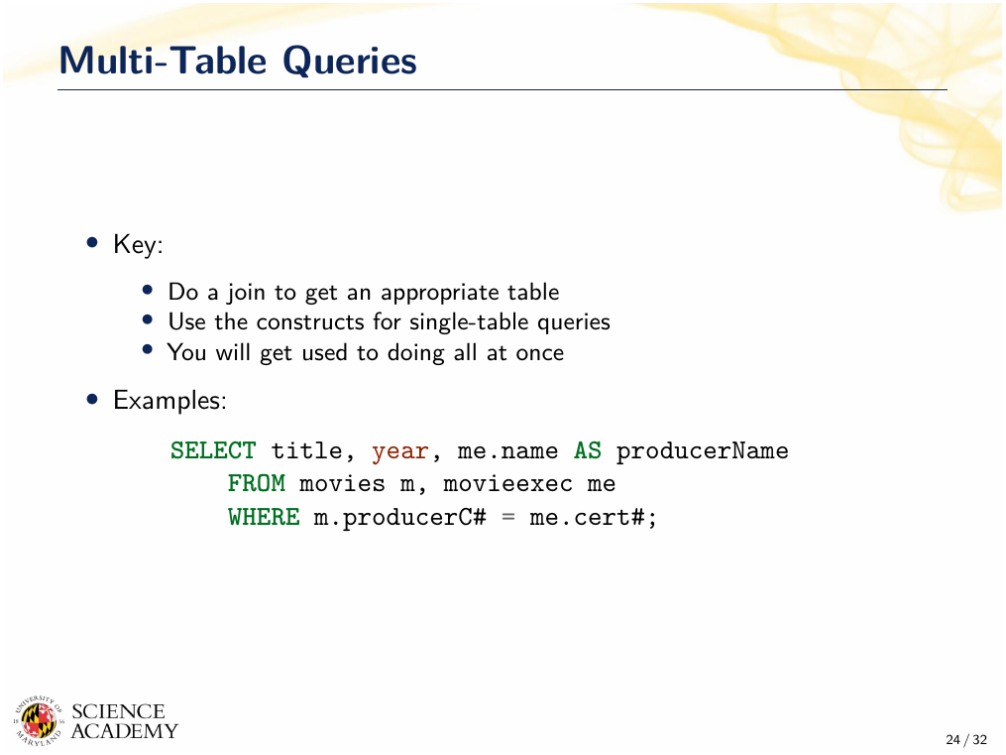
## 23 / 32: Single Table Queries




- **Single Table Queries**
  - *Set Comparisons*

* The first SQL query demonstrates how to retrieve all records from the `movies` table where the `year` column matches any of the specified values: 1990, 1995, or 2000. This is achieved using the `IN` keyword, which is a convenient way to filter results based on a list of values. It's particularly useful when you want to check if a column's value belongs to a specific set of options.
* The second SQL query is a bit more complex. It selects all records from the `movies` table where the `year` is *not* in a set of years extracted from the `birthdate` column of the `MovieStar` table. The `EXTRACT(year from birthdate)` function is used to pull out the year part from the `birthdate`. This query is an example of a subquery, where the inner query provides a list of years that are then used by the outer query to filter out movies released in those years. This is useful for excluding certain records based on a dynamic set of values derived from another table.

## 24 / 32: Multi-Table Queries

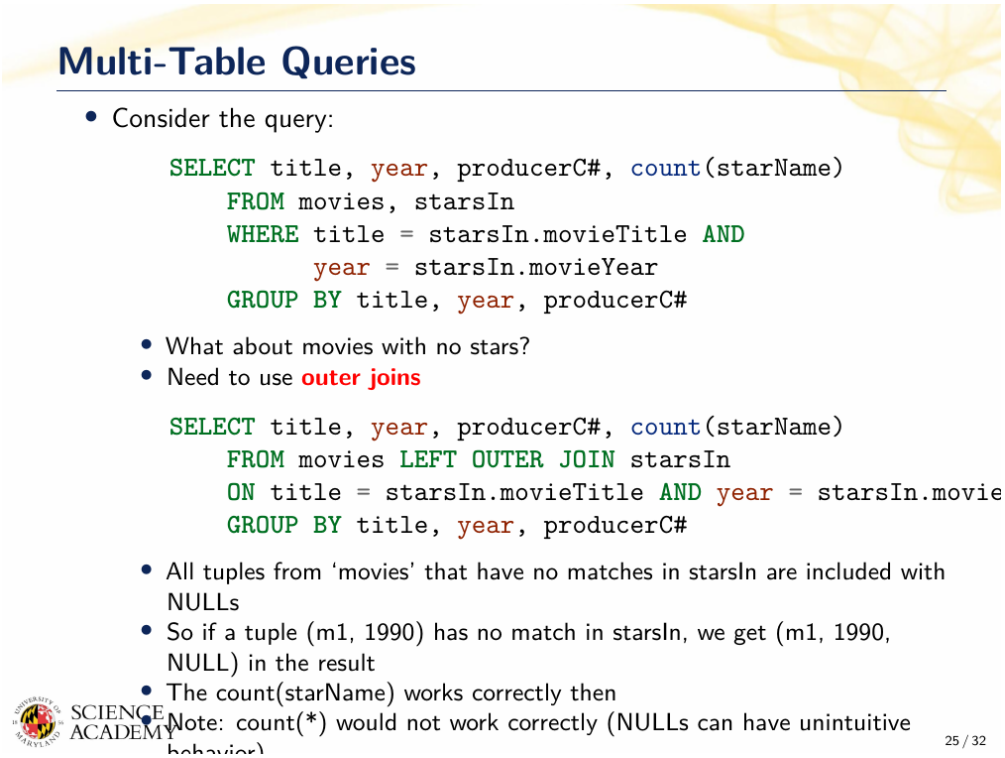



- **Multi-Table Queries**
  - **Key:**
    * *Do a join to get an appropriate table*: When working with databases, data is often spread across multiple tables. To extract meaningful information, you need to combine these tables. This process is called a "join." It allows you to link tables based on a related column, such as an ID or a key.
    * *Use the constructs for single-table queries*: Once you've joined the tables, you can use the same SQL commands you would use for a single table. This means you can filter, sort, and select data as if it were all in one table.
    * *You will get used to doing all at once*: Initially, working with multi-table queries might seem complex. However, with practice, you'll become comfortable with writ-

ing these queries and performing joins as a routine part of your data analysis.

- **Examples:**
  - The provided SQL query demonstrates a multi-table query. It selects the `title` and `year` of movies from the `movies` table and the `name` of the producer from the `movieexec` table. The `WHERE` clause specifies that the `producerC#` from the `movies` table should match the `cert#` from the `movieexec` table. This join allows you to see which producer is associated with each movie, combining data from both tables into a single, cohesive result.

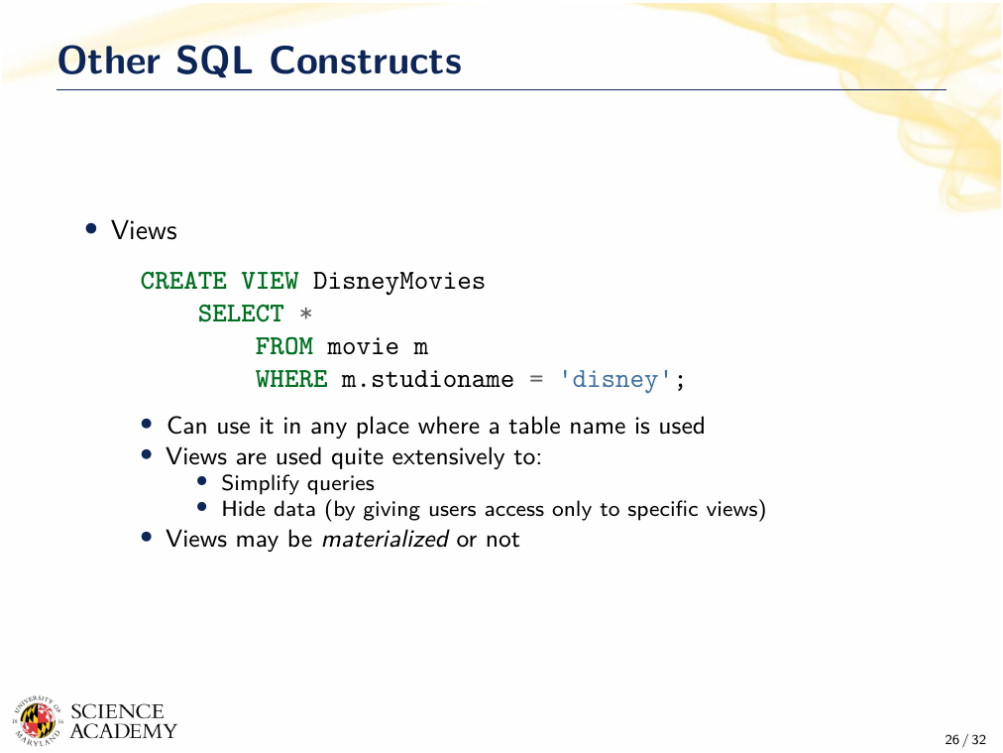
## 25 / 32: Multi-Table Queries




- **Multi-Table Queries**
  - The first query example demonstrates a basic SQL query that retrieves information from two tables: `movies` and `starsIn`. It selects the movie title, year, producer code, and counts the number of stars associated with each movie.
  - **Problem with the initial query**: It only considers movies that have entries in both tables. This means if a movie has no stars listed in the `starsIn` table, it won't appear in the results at all.
  - **Solution: Use Outer Joins**: To include movies without stars, we use a **LEFT OUTER JOIN**. This type of join ensures that all records from the `movies` table are included in the results, even if there are no corresponding entries in the `starsIn` table.
  - **Handling NULLs**: When there is no match in `starsIn`, the result will include NULLs for the star-related columns. For example, a movie with no stars will appear as `(m1, 1990, NULL)`.
  - **Counting Stars**: Using `count(starName)` correctly counts the number of stars, as it ignores NULLs. However, using `count(*)` would count all rows, including those with

NULLs, which might lead to misleading results.

## 26 / 32: Other SQL Constructs




- **Other SQL Constructs**
- **Views**

```
CREATE VIEW DisneyMovies
    SELECT *
        FROM movie m
        WHERE m.studioname = 'disney';
```

  - **Can use it in any place where a table name is used**: Once a view is created, you can use it just like a regular table in your SQL queries. This means you can perform operations like SELECT, JOIN, and more on the view.
  - **Views are used quite extensively to**:
    - **Simplify queries**: Views can encapsulate complex queries, making it easier to work with data. Instead of writing a long query every time, you can just reference the view.
    - **Hide data**: By creating views, you can control what data users can see. For example, if you only want users to see Disney movies, you can give them access to the `DisneyMovies` view instead of the entire `movie` table.
  - **Views may be *materialized* or not**: A *materialized view* is a view where the data is stored physically, which can improve performance for complex queries. Non-materialized views, on the other hand, are virtual and do not store data physically; they are computed on the fly when queried.

# 27 / 32: Other SQL Constructs

## Other SQL Constructs

- NULLs
  - Value of any attribute can be NULL
  - Because: value is unknown, or it is not applicable, or hidden, etc.
  - Can lead to counterintuitive behavior
  - For example, the following query does not return movies where length = NULL

```
SELECT * FROM movies
WHERE length >= 120 OR length <= 120
```

  - Aggregate operations can be especially tricky
- Transactions
  - A transaction is a sequence of queries and update statements executed as a single unit
  - For example, transferring money from one account to another
    - Both the *deduction* from one account and *credit* to the other account should happen, or neither should
- Triggers
  - A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database


27 / 32

- **NULLs**
  - In databases, any attribute can have a value of **NULL**, which essentially means that the value is unknown, not applicable, or intentionally hidden. This is important because it allows for flexibility in data representation, but it can also lead to unexpected results. For instance, when you run a query to find movies with a specific length, if the length is **NULL**, it won't be included in the results. This is because **NULL** is not considered equal to any value, including itself, which can be counterintuitive. Additionally, when performing aggregate operations like counting or averaging, **NULL** values can complicate the results, as they are typically ignored in these calculations.
- **Transactions**
  - A transaction in SQL is a sequence of operations that are executed as a single unit. This is crucial for maintaining data integrity, especially in scenarios like transferring money between bank accounts. In such cases, both the deduction from one account and the credit to another must occur together. If one part of the transaction fails, the entire transaction should be rolled back to ensure consistency. This all-or-nothing approach helps prevent errors and ensures that the database remains in a valid state.
- **Triggers**
  - Triggers are special types of SQL statements that are automatically executed in response to certain events on a table or view. They are used to enforce business rules, maintain audit trails, or automatically update related data. For example, a trigger might automatically update a stock inventory count when a new sale is recorded. Triggers help automate processes and ensure that certain actions are taken without requiring manual intervention, thus maintaining the integrity and consistency of the database.

# 28 / 32: Other SQL Constructs

## Other SQL Constructs

- Integrity Constraints
  - Predicates on the database that must always hold
  - Key Constraints: Specifying something is a primary key or unique

```
CREATE TABLE customer (
    ssn CHAR(9) PRIMARY KEY,
    cname CHAR(15),
    address CHAR(30),
    city CHAR(10),
UNIQUE (cname, address, city));
```

  - Attribute constraints: Constraints on the values of attributes

```
bname char(15) not null
balance int not null, check (balance>= 0)
```

SCIENCE ACADEMY

28 / 32

- **Integrity Constraints**
  - *Integrity constraints* are rules that ensure the accuracy and consistency of data within a database. They are conditions that the data must satisfy at all times. This is crucial for maintaining the reliability of the database.
  - **Key Constraints**: These are specific types of integrity constraints that ensure uniqueness and identify records within a table. For example, a *primary key* is a unique identifier for each record in a table. In the provided SQL example, `ssn` is defined as the primary key for the `customer` table, meaning no two customers can have the same social security number. Additionally, the `UNIQUE` constraint ensures that the combination of `cname`, `address`, and `city` is unique across all records, preventing duplicate entries with the same customer name, address, and city.

```
CREATE TABLE customer (
    ssn CHAR(9) PRIMARY KEY,
    cname CHAR(15),
    address CHAR(30),
    city CHAR(10),
UNIQUE (cname, address, city));
```

  - **Attribute Constraints**: These constraints apply to individual columns within a table. They specify rules for the values that can be stored in a column. For instance, the `NOT NULL` constraint ensures that a column cannot have a null value, meaning it must always contain data. In the example, `bname` is defined as `NOT NULL`, ensuring that every record must have a value for `bname`. Similarly, the `CHECK` constraint is used to enforce a condition on the values in a column. For example, `balance` must be a non-negative integer, as specified by `check (balance>= 0)`.

```
bname char(15) not null
```

```
balance int not null, check (balance>= 0)
```

## 29 / 32: Integrity Constraints

### Integrity Constraints

- Referential integrity: prevent dangling tuples

```
CREATE TABLE branch(bname CHAR(15) PRIMARY KEY, ...);
CREATE TABLE loan(..., FOREIGN KEY bname REFERENCES branch);
```

- Can tell the system what to do if a referenced tuple is being deleted

SCIENCE ACADEMY

29 / 32

- **Integrity Constraints**
  - **Referential integrity: prevent dangling tuples**
    - *Referential integrity* is a concept in databases that ensures relationships between tables remain consistent. When we have two tables that are related, like a `branch` table and a `loan` table, referential integrity makes sure that any reference to a row in one table corresponds to a valid row in the other table.
    - In the example provided, the `branch` table has a primary key `bname`, which uniquely identifies each branch. The `loan` table has a foreign key `bname` that references this primary key. This setup ensures that every loan is associated with a valid branch.
    - By enforcing referential integrity, we prevent "dangling tuples," which are records in the `loan` table that reference a non-existent branch. This is crucial for maintaining data accuracy and consistency.
  - **Can tell the system what to do if a referenced tuple is being deleted**
    - When a referenced tuple (like a branch) is deleted, we need to decide how to handle related records in other tables (like loans). Options include:
      - *Cascade*: Automatically delete or update the related records.
      - *Set Null*: Set the foreign key in related records to `NULL`.
      - *Restrict*: Prevent the deletion if related records exist.
    - These options help maintain data integrity and allow for flexible database management.

# 30 / 32: Integrity Constraints

## Integrity Constraints

- Global Constraints
  - Single-table

```
CREATE TABLE branch (..., bcity CHAR(15), assets INT,
    CHECK (NOT(bcity = 'Bkln') OR assets>5M))
```

  - Multi-table

```
CREATE ASSERTION loan-constraint
CHECK (NOT EXISTS
    (SELECT* FROM loan AS L WHERE NOT EXISTS
        (SELECT* FROM borrower B, depositor D, account A
        WHERE B.cname = D.cname AND D.acct_no = A.acct_no
            AND L.lno= B.lno)))
```

SCIENCE ACADEMY

30 / 32

- **Integrity Constraints**
  - Integrity constraints are rules that ensure data accuracy and consistency within a database. They are crucial for maintaining the quality and reliability of the data stored.
- **Global Constraints**
  - Global constraints are rules that apply to the entire database rather than just a single table. They help enforce data integrity across multiple tables or the entire database system.
  - **Single-table**
    - The example provided shows a constraint applied to a single table. In this case, the `branch` table has a constraint that checks if the city (`bcity`) is 'Bkln' (Brooklyn), then the `assets` must be greater than 5 million. This ensures that branches in Brooklyn have a minimum asset value, which might be a business rule for financial stability.
  - **Multi-table**
    - Multi-table constraints involve more than one table to enforce data integrity. The example uses an assertion named `loan-constraint` to ensure that every loan in the `loan` table has a corresponding borrower who is also a depositor with an account. This constraint prevents loans from existing without a valid borrower and depositor relationship, ensuring that all loans are properly linked to customer accounts. This is crucial for maintaining the integrity of financial transactions and relationships within the database.

# 31 / 32: Additional SQL Constructs

## Additional SQL Constructs

- SELECT subquery factoring
  - To allow assigning a name to a subquery, then use its result by referencing that name

```
WITH temp AS (
    SELECT title, avg(length)
    FROM movies
    GROUP BY year)
SELECT COUNT(*) FROM temp;
```

  - Can have multiple subqueries (multiple with clauses)
  - Real advantage is when subquery needs to be referenced multiple times in main select
  - Helps with complex queries, both for readability and maybe performance (can cache subquery results)


31 / 32

- **`SELECT` subquery factoring**
  - This concept is about using the `WITH` clause in SQL to create a temporary result set that you can refer to multiple times within your main query. It's like giving a name to a subquery so you can easily use it later.
  - In the example provided, a subquery is named `temp`, which calculates the average length of movies grouped by year. This result can then be used in the main query to count the number of entries in `temp`.
  - **Multiple subqueries**: You can define more than one subquery using multiple `WITH` clauses. This is useful when you have several intermediate results you want to use in your main query.
  - **Advantages**: This approach is particularly beneficial for complex queries. It improves readability by breaking down the query into understandable parts. Additionally, it might enhance performance because the database can cache the results of the subquery, avoiding redundant calculations. This is especially useful when the subquery is referenced multiple times.

# 32 / 32: Another SQL Construct

## Another SQL Construct

- `SELECT HAVING` clause
  - Used in combination with `GROUP BY` to restrict the groups of returned rows to only those where condition evaluates to true

```
SELECT year, count(*)
    FROM movies
    WHERE year > 1980
    GROUP BY year
    HAVING COUNT(*) > 10;
```

  - Difference from `WHERE` clause is that it applies to summarized group records, and where applies to individual records

SCIENCE ACADEMY

32 / 32

- **`SELECT HAVING` clause**
  - The `HAVING` clause is a powerful tool in SQL that is used alongside the `GROUP BY` clause. Its main purpose is to filter groups of data that have been aggregated, allowing you to specify conditions that these groups must meet to be included in the final result set.
  - In the provided SQL example, the query is selecting movies released after the year 1980. It groups these movies by their release year and then counts how many movies were released each year. The `HAVING` clause is then used to filter these groups, ensuring that only years with more than 10 movies are included in the results.
  - It's important to note that the `HAVING` clause is different from the `WHERE` clause. While `WHERE` filters individual rows before any grouping takes place, `HAVING` filters the groups after the aggregation has been performed. This distinction is crucial when working with grouped data in SQL.