



The slide features a light gray background with a dark blue header bar at the top. The title "SQL DML" is centered in a large, bold, dark blue font, with the subtitle "Querying a Single Table" centered below it in a smaller, dark blue font. In the bottom left corner, there is a dark blue square containing the Greek letter π in white. In the bottom right corner, there is a small red square containing the text "SWINBURNE UNIVERSITY OF TECHNOLOGY" in white.

SQL DML

Querying a Single Table

π

SWINBURNE
UNIVERSITY OF TECHNOLOGY

In this presentation we'll look at how to make queries that extract information from a single table.

Finding Information

Product

id	name	location	category
LW24453	Grass eliminator V5	A5B17	Lawn mowers
T00053	Makita Screwdriver M7	A13B5	Electric tools
CL19005	Kärcher HP5005	A2B22	Cleaners

```
SELECT name, location, category
FROM Product;
```

Projection

Result set:

name	location	category
Grass eliminator V5	A5B17	Lawn mowers
Makita Screwdriver M7	A13B5	Electric tools
Kärcher HP5005	A2B22	Cleaners

When you add information to a relational database, you chop it up into very small pieces - like name and location - whose meaning and structure – and their relationships – are exactly defined. When you want to find information in these pieces of data, you run a query that combines the pieces to something meaningful. SQL has been invented as a powerful tool that can combine pieces of information in many different ways.

Queries always start with the keyword **SELECT**. **SELECT** statements retrieve data from one or more tables, therefore you have to specify at least one table that has the required data. The **FROM** keyword identifies the tables, therefore the **FROM** clause is never missing in a valid query.

Before the **FROM** keyword you can see a number of column names. The select statement specifies the attributes to include. They are also mandatory. Picking the attributes is called a projection. Now we have projected three of the attributes of the **Product** table into our result set.

The result of a query in SQL is often called a result set, because of the roots of SQL in relational algebra. This is actually slightly misleading, because sets don't have duplicates, but in SQL the results do.

You can see that projections create a type of vertical split of the original table – some columns have been left out, in this case the **id** column. But we haven't imposed any horizontal split – the result has just as many rows as the original table. This isn't always desirable. Tables can have lots of entries, and chances are you're only interested in a few of them.

Finding a Subset of Tuples

Product

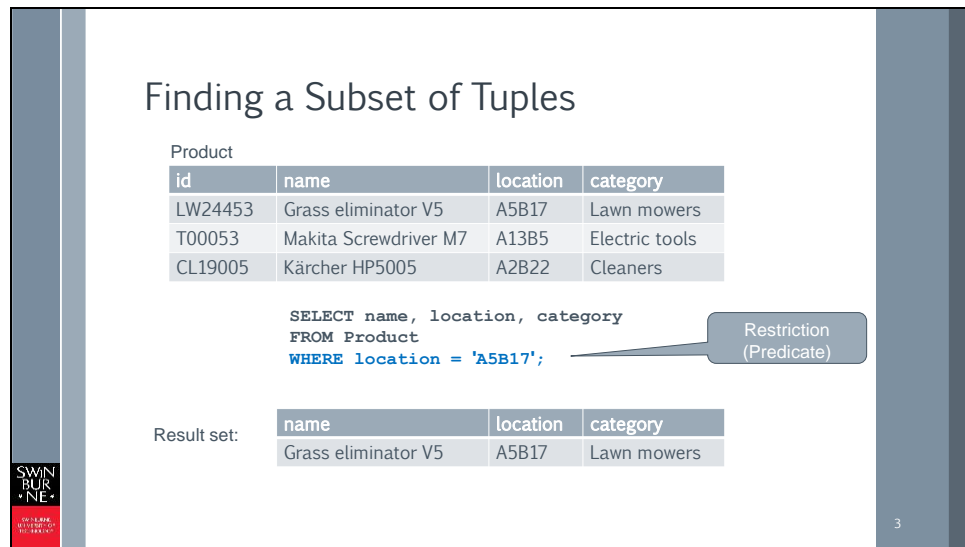
id	name	location	category
LW24453	Grass eliminator V5	A5B17	Lawn mowers
T00053	Makita Screwdriver M7	A13B5	Electric tools
CL19005	Kärcher HP5005	A2B22	Cleaners

```
SELECT name, location, category
FROM Product
WHERE location = 'A5B17';
```

Restriction (Predicate)

Result set:

name	location	category
Grass eliminator V5	A5B17	Lawn mowers



SWINBURNE
UNIVERSITY
TECHNOLOGY

3

If you want to find only a subset of the rows in a table, you simply add a WHERE clause to your SELECT statement. Where clauses impose conditions that restrict the number of tuples included in the result. Therefore they are also known as restrictions. To add more terminology, where the conditions in WHERE clauses are also called predicates.

If you use string values in your predicate, you have to place the literal value in single quotes. The string is case sensitive.

There are a number of comparison operators you can use in your WHERE clause.

Predicates

```
SELECT name, location, category
FROM Product
WHERE location <> 'A5B17';
```

Result set:

name	location	category
Makita Screwdriver M7	A13B5	Electric tools
Kärcher HP5005	A2B22	Cleaners


```
SELECT name, location, category
FROM Product
WHERE location LIKE 'A%';
```

Result set:

name	location	category
Grass eliminator V5	A5B17	Lawn mowers
Makita Screwdriver M7	A13B5	Electric tools
Kärcher HP5005	A2B22	Cleaners

not equals

using wildcards

SWINBURNE
UNIVERSITY
TECHNOLOGY


4


Naturally, WHERE clauses can use less than and greater than comparisons. If you combine the two symbols, it means not equal to. But the not equals symbol can be different depending on the database. MSSQLServer uses Java-style not equals, DB2 uses yet another symbol.

If you want to use wildcards (symbols that mean any character), you have to use the LIKE expression. A percentage sign means any number of digits or characters (or none), an underscore means exactly one character or digit.


More Predicate Comparison Operators

```
...WHERE startDate BETWEEN '2015-10-01' AND '2015-10-31';  
  
...WHERE state (NOT) IN ('WA', 'VIC', 'NSW');  
  
...WHERE salary >= 1000 AND age <= 50;  
  
...WHERE salary >= 1000 OR position = 'manager';  
  
...WHERE state IS NULL;  
  
...WHERE state IS NOT NULL;
```

 = null
doesn't
work



The syntax for using regular expressions is different in every database product!

 5

BETWEEN AND can be used for numbers and dates. It includes the extreme values, so in this case values from the first and 31st October are included.

The IN keyword lets you enumerate several options for the matches. At this stage you may wonder whether you can use regular expressions to search databases. Naturally you can, but the operator is different in each database product! Just do a web search for regular expressions and the database you are using.

You can use the AND and OR operators to combine restrictions in predicates. You can also use parentheses if you need to make more complex options.

If you need to check for null values, you have to use is null or is not null, the equals comparator doesn't work in this case.


Traditionally, all SQL keywords are capitalized. These days, most database systems will accept lower case keywords. The upper case on the slides is just there to improve legibility.

The 'All Columns' Shorthand

```
SELECT *  
FROM Product  
WHERE location NOT LIKE 'A5%';
```

id	name	location	category
T00053	Makita Screwdriver M7	A13B5	Electric tools
CL19005	Kärcher HP5005	A2B22	Cleaners

That's just lazy. Cats like lazy.



SWINBURNE
UNIVERSITY
TECHNOLOGY

6


Select star or asterisk is the shorthand notation if you want to include all attributes in a projection. It's a good place to start writing your queries. First you refine all your predicates and other details. When the query does what you want, you can decide which columns are actually needed. When you know what you want to see in the result, you name all the attributes you want to include.

Projection, Aliases and Concatenation

alias

```
SELECT category || ' - ' || name AS description, location AS shelf
FROM Product;
```

description	shelf
Lawn mowers - Grass eliminator V5	A5B17
Electric tools - Makita Screwdriver M7	A13B5
Cleaners - Kärcher HP5005	A2B22


7

According to the SQL standard, you can concatenate strings using a double pipe sign. If one of the values that is being concatenated is not a character field, depending on the database product you may have to use a CAST to characters first.

In the example, category and name are both character fields. We want to put them into the same column in the result set, and separate them by a dash. The dash and the two spaces on either side are a literal string (a string constant) The literal string is printed in single quotes, like all character values. So the category value goes first, then a dash and then the name field.

The AS keyword indicates we are renaming the column. The column is the result of computation, so it doesn't actually have a name. But if we don't mention a name, the DBMS will make one up. Most database systems show the function that creates the value as a heading.

It is better to decide ourselves what the column's heading should be. When we say AS description, we mean the column heading should be description.

We can use aliases for any column in a result set, it doesn't have to be a generated one. In the example, we rename 'location' to 'shelf' because we are assuming that the location means some shelf space in the warehouse.

What do you think the result of the query would look like? You may want to pause now to think about it.

Sorting Results


Product

id	name	location	category
LW24453	Grass eliminator V5	A5B17	Lawn mowers
T00053	Makita Screwdriver M7	A13B5	Electric tools
CL19005	Kärcher HP5005	A2B22	Cleaners

```
SELECT name, location, category
FROM Product
ORDER BY category;
```

Result set:

name	location	category
Kärcher HP5005	A2B22	Cleaners
Makita Screwdriver M7	A13B5	Electric tools
Grass eliminator V5	A5B17	Lawn mowers



8

You can sort the result set by any column. You can have any number of columns in your `ORDER BY` clause. This means that the sort happens on the first column, and if the first column has duplicates, these duplicates are sorted by the second column, etc.

Clearly, if your column has unique values, there is no point in using secondary columns for your sort operation, like in this case. The more columns you use, the more unique combinations you'll have, so anything beyond three columns is usually unnecessary. Sort operations are expensive, so think about what you are doing before you include such SQL in an application.

Null

- › is not zero (0)
- › is not an empty string ('')
- › is not space characters (' ')
- › is an unknown value

"ternary logic"


Customer
id number [PK] not null
firstname VARCHAR not null
lastname VARCHAR not null
address VARCHAR not null
gender CHAR

What is your gender?

☐ Male

☐ Female

☐ Prefer not to say



9

We have already discussed NULL values in a previous module, so you know they mean we don't know what the actual value is when a field is null. You have also been told you should avoid having columns that allow null values whenever you can, because nulls can cause trouble. Now we are going to look into the trouble nulls can cause in queries.

Nulls in Predicates

Customer

id	firstname	lastname	address	gender
111	Susan	Jones	Melbourne	F
222	Ming	Wong	Hawthorn	M
333	Kalpita	Chaudhuri	Fitzroy	null
444	Adam	Bauer	Sunshine	M
555	Mary	Liu	Kew	F

```
SELECT *  
FROM Customer  
WHERE gender = 'F';
```

id	firstname	lastname	address	gender
111	Susan	Jones	Melbourne	F
555	Mary	Liu	Kew	F

Suppose we are querying the customer table by gender. We want to find all the females. There is one null in the table for the person with the id of 333. The person appears to be female based on the first name, but in our query we are using the gender field and a value of F for female, so person number 333 is not picked up.

That's probably what you expected, right? So no problems then.

Nulls in Negative Comparisons

Customer

id	firstname	lastname	address	gender
111	Susan	Jones	Melbourne	F
222	Ming	Wong	Hawthorn	M
333	Kalpita	Chaudhuri	Fitzroy	null
444	Adam	Bauer	Sunshine	M
555	Mary	Liu	Kew	F

```

SELECT *
FROM Customer
WHERE gender <> 'F' OR gender IS NULL;

```

id	firstname	lastname	address	gender
222	Ming	Wong	Hawthorn	M
444	Adam	Bauer	Sunshine	M

11

What if we want to fetch all customers that are not female. We can apply the not equals symbol (the combination of less than and greater than) and expect to get the three rows that are not female.

Now here is the catch – if a value is null we can't say that it is not female.

The value is unknown, so there is a chance it might be female. There is also a chance that it might be male. We simply don't know. The only problem is that people tend to forget this and expect nulls to be included in this case. If you really want to pick up the nulls as well as the male tuples here, you need to check for nulls explicitly.

Aggregate Functions



id	name	location	category
LW24453	Grass eliminator V5	A5B17	Lawn mowers
T00053	Makita Screwdriver M7	A13B5	Electric tools
CL19005	Kärcher HP5005	A2B22	Cleaners

same as
COUNT(id) → `SELECT COUNT(*) as productCount
FROM Product;`

Result set:

productCount
3

MAX()
MIN()
COUNT()
SUM()
AVG()

SWINBURNE
UNIVERSITY
OF TECHNOLOGY

12

All database products implement the aggregate functions SUM, AVG, COUNT, MIN and MAX which you have already seen in the relational algebra. There are several more, for example the ABS() function that returns the absolute value of a number. You can use the COUNT(*) function to count all the rows in a table. You can also use count on an individual column, such as the ID, to get the same result. But you have to be careful – if the column is nullable, the null values are generally not counted.

If you use SUM, AVG, MIN or MAX, you can't use the star because you have to say which values need to be totalled. You can't sum up the values of all columns of a table. So all other aggregate functions need a specific column in the parenthesis behind them.

Aggregate functions are clearly useful, but having to apply them to all rows of the table at the same time, and get just one single value back is not all that helpful. Suppose you want to know how many products you have in each category.

Aggregate Functions

Product

id	name	location	category
LW24453	Grass eliminator V5	A5B17	Lawn mowers
T00053	Makita Screwdriver M7	A13B5	Electric tools
CL19005	Kärcher HP5005	A2B22	Cleaners

```

SELECT COUNT(*) as productCount, category
FROM Product
WHERE category='Lawn mowers';

```

Result set:

productCount	category
1	Lawn mowers

MySQL lets you do this

This could be a very tiresome task. You would have to query the table first to find all possible entries in the category column, then use each category entry in turn in a restriction (a WHERE clause) to find out the number of the products in each category.

Naturally, SQL has a better solution for this.

But first, have a look at the category projection in the SELECT statement. Mowers is a literal value, so it is not fetched from the database at all. It is given an alias of category, and you can see it appears in the result set as a column of its own.

Making up new columns in the query on the fly is sometimes helpful.

In this case, there is a chance that you need to do this because your database product may not be happy to let you simply include the category column from the database. Why do you think that might be? Well, if you think of it, the COUNT function actually takes a number of rows, because most likely there are several lawn mowers. Although the value of category will be the same for each of these rows (because of the restriction), the database system doesn't realise this. It just assumes there are several rows involved in the count function's calculations. But the database system also knows that because of the count function, the result of these rows is combined into one row in the result set. Which row's category entry should be shown next to the single outcome of count?

If the DBMS is clever enough to figure out that the category is the same field as the restriction, and that the values have to be the same, it will be happy for you just to include the category field if you want to show it in your result. As it happens, MySQL is that clever.

Now suppose you want to have the results for all categories in the same table.



UNION

id	name	location	category
LW24453	Grass eliminator V5	A5B17	Lawn mowers
T00053	Makita Screwdriver M7	A13B5	Electric tools

```

SELECT COUNT(*) as productCount, 'Mowers' as category
FROM Product WHERE category='Lawn mowers'
UNION
SELECT COUNT(*) as productCount, 'E-tools' as category
FROM Product WHERE category='Electric tools'
UNION
SELECT COUNT(*) as productCount, 'Cleaners' as category
FROM Product WHERE category='Cleaners';
  
```

productCount	category
1	Lawn mowers
3	E-tools
2	Cleaners

SWINBURNE
UNIVERSITY
OF TECHNOLOGY

14

First, we'll add a few more rows to our example table to make the query more interesting.

What we could do is use UNION. Union unites several results in the same result set. Obviously, this only makes sense when each query returns a result that can be placed in the same columns in the result. So from that perspective, this example is perfect for demonstrating how UNION works.

If we write a query that returns a count for each row that has a particular category, we can put all queries for the individual categories into the same query by putting 'UNION' between them. As long as we use the same columns for the result of each query, database systems will be happy to execute the combined query and show the result in a single result set.

This is cool if we want to show how UNION works, but rather silly as a solution for the task of finding the number of products by category.

Grouping

Product			
id	name	location	category
LW24453	Grass eliminator V5	A5B17	Lawn mowers
T00053	Makita Screwdriver M7	A13B5	Electric tools
CL19005	Kärcher HP5005	A2B22	Cleaners
CL18006	Kärcher HP6001	A2B23	Cleaners
T00132	Bosch Hammer Drill	A13B5	Electric tools
T00202	Ryobi Angle Grinder	A13B5	Electric tools

count	category
1	Lawn mowers
3	E-tools
2	Cleaners

```

SELECT COUNT(*) as count, category
FROM Product
GROUP BY category;
```

The way we should really be solving this problem is by using the GROUP BY clause. If you use group by, the aggregate function you use is applied to the groups rather than the entire table. So you can get the count of the products per category without having to put a restriction and mention each of the categories separately.

You can also group by several columns, if you want to count the number of electric tools that are found in the same location, for example. The more columns you use, the more groups you usually get.

With grouping, the rule is that every column you use in the projection, except the aggregate one, also has to be mentioned in the group by clause. When you work with aggregate functions and they just give you an error message all the time, you've probably forgotten to include a column in the group by clause.

If you look at it from the logical angle, the point with aggregate functions is to combine several rows into one value. The grouping function tells the database system what defines the groups. If any column that doesn't define a group is mentioned in the projection, this means we have several rows to pick our value from.

Suppose you wanted to include the name of the product in the result set. Naturally, this won't work because for each category, there will be several product names. The database system would have to choose a random one of them to

include in the listing. So it forces you to group by a column you include. But if you include the name column in the listing, naturally the list becomes much longer, because we now no longer count by category, but by individual product. In short, if you have a column that is not the aggregate missing in the GROUP BY statement, you can't include it in the projection.

Restrictions and Aggregates

Product

id	name	location	category
LW24453	Grass eliminator V5	A5B17	Lawn mowers
T00053	Makita Screwdriver M7	A13B5	Electric tools
CL19005	Kärcher HP5005	A2B22	Cleaners
CL18006	Kärcher HP6001	A2B23	Cleaners
T00132	Bosch Hammer Drill	A13B5	Electric tools
T00202	Ryobi Angle Grinder	A13B5	Electric tools


```

SELECT COUNT(*) as count, category
FROM Product
GROUP BY category
HAVING COUNT(id) > 1;

```

count	category
3	E-tools
2	Cleaners



SWINBURNE
UNIVERSITY
VICTORIA

16

If you need a restriction that uses aggregates, the WHERE clause doesn't work. You'll have to use HAVING instead. WHERE clauses are static predicates that can only work on existing columns. If you do a COUNT, or use any other aggregate function, you are essentially creating a new value that doesn't exist in the database.

Aggregate functions have to do a grouping while they calculate the result. They have to loop through a subset of rows to calculate. Therefore the grouping has to exist before the value can be returned. WHERE cannot work on groups, therefore we need the HAVING clause here.

Summary



- › Basic queries take the form
 - › `SELECT <column1, column2, ...> FROM <tablename>;`
- › Restrictions are added using a WHERE clause.
- › Null values never match a condition, so you have to include them explicitly in your WHERE clause using `IS (NOT) NULL`.
- › Databases have some useful aggregation functions. They are often used with GROUP BY clauses to aggregate over groups of tuples.
- › If you need to use aggregations in restrictions, you need to use a HAVING instead of a WHERE clause.

17

Here is the summary of the most important points. You may want to read them before you go on to do the quiz.