

Ian Kenny

October 3, 2016

# Databases

## Lecture 2

# Module structure

There are two lectures per week

- Monday, 1600, Mechanical and Civil Engineering, G34
- Friday, 1000, Mechanical and Civil Engineering, G29

There is one lab session/tutorial per week in the CS labs at **1200 on Fridays**. This is a change from the published timetable.

# In this lecture

- SQL queries.

# The relational model

In the relational model, data is organised into *relations*. Relations are also called 'tables'. A relation usually represents a logical entity or relationship of some kind, for example, 'Employee'.

A relation has *attributes*. Attributes are also called 'fields' or 'columns'. The attributes of a relation are the 'categories' of information stored about the entity or relationship, for example, 'salary', 'date of birth').

Attributes have a *domain*, i.e. the range of values that the value is taken from. For example, 'strings of length 255 characters', integers, dates.

A relation contains *tuples* (if it is not empty). Tuples are also called 'rows', 'instances'. These are the actual 'instances' of the data in the relation (with reference to object-oriented programming, if the relation is the class then the tuples are like the objects).

# Keys

A *key* for a relation is a set of attributes that uniquely identifies the tuples in the relation. The set of all attributes of a relation is clearly a key for the relation but keys usually consist of a subset of the attributes and often a single attribute.

A *candidate key* is any key of the relation.

The *primary key* for a relation is key that has been selected from the candidate keys to act as the key for a relation.

A *foreign key* is a set of attributes in a relation that uniquely identifies a row in a *another table*.

Keys are crucial for the operation of databases.

# SQL

- Structured Query Language ('S-Q-L' or 'sequel') is a declarative language for the creation, manipulation and querying of databases.
- With SQL we do not specify *how* something is to be done but *what* is to be done.
- SQL is a flexible language in the sense that the same command can be expressed in multiple ways.
- SQL is *case insensitive* hence keywords and relation names can be typed in upper or lower case characters.
- Strings in SQL are case sensitive, however: 'John'  $\neq$  'john'.
- In order to distinguish SQL keywords from attribute and relation names, etc., we will (generally) adopt the convention that SQL keywords are written with upper case characters.
- In this lecture we will focus on querying databases using SQL. Creating and manipulating databases will be covered in a later lecture.

# SQL: basic SELECT command

The SELECT command selects data from a database. This will usually be a subset of the data in the database. You don't generally want all of the tuples.

The SELECT command has the following basic form:

```
SELECT <A1, A2, ..., An>  
FROM <R1, R2, ..., Rm>  
[WHERE condition is met]
```

Where:

$A_1, A_2, \dots, A_n$  are the attributes to be selected.

$R_1, R_2, \dots, R_m$  are the relations to query.

Any rows that meet the [optional] condition in the WHERE clause will be added to the result.

Each of the clauses (SELECT-FROM-WHERE) can be complex.



# SQL: basic SELECT command

The SELECT command returns a relation. This means that SELECT commands can be nested in a clause of another SELECT command where a relation is expected.

If a relation returned by a SELECT command contains a single attribute and a single tuple then SQL will silently convert that to a scalar value where necessary. No casting is required.

# SQL: Basic SELECT command

Consider relation  $R_1$

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22

$R_1$

# SQL: Basic SELECT command

Consider the following query on relation  $R_1$ .

```
SELECT A1, A2  
FROM R1  
WHERE A3 = 'ABB'
```

A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22

$R_1$

In this simple query the FROM clause is evaluated first to determine which relation is to be queried; the WHERE clause is then evaluated to remove tuples from the relation that do not meet the condition; finally the SELECT clause is evaluated to remove unwanted attributes.

A <sub>1</sub>	A <sub>2</sub>
1	2.5
8	0.7

*Result*

# SQL: Basic SELECT command

To get all of the attributes in the result we simply use SELECT \*.

```
SELECT *  
FROM  $R_1$   
WHERE  $A_3 = \text{'ABB'}$ 
```

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22

$R_1$

In this case all of the columns are returned in the result.

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
8	0.7	'ABB'	22

Result

# SQL: Basic SELECT command

By default the SELECT command does not remove duplicates from the result.

```
SELECT A3  
FROM R1
```

This is the equivalent query to specifying that duplicates should be included.

```
SELECT ALL A3  
FROM R1
```

Note that this query has no WHERE clause. In this case, all of the tuples will be returned.

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22

$R_1$

$A_3$
'ABB'
'CDK'
'ABB'

*Result*

# SQL: Basic SELECT command

To remove duplicates from the result we specify the DISTINCT option.

```
SELECT DISTINCT A3  
FROM R1
```

Note that the order of the tuples in the result is essentially random.

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22

$R_1$

$A_3$
'CDK'
'ABB'

*Result*

# SQL: Basic SELECT command

We can perform computation and other operations in the SELECT clause.

```
SELECT A1, A4 * 10  
FROM R1  
WHERE A3 = 'ABB'
```

A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22

*R<sub>1</sub>*

In this query, the values in the attribute A<sub>4</sub> will be multiplied by 10 before being added to the result.

A <sub>1</sub>	???
1	150
8	220

*Result*

Note that the second result column has no name. We can fix this by creating an alias.

# SQL: Basic SELECT command

We can perform computation and other operations in the SELECT clause.

```
SELECT A1, A4 * 10 AS result  
FROM R1  
WHERE A3 = 'ABB'
```

A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22

*R<sub>1</sub>*

In this query, the values in the attribute A<sub>4</sub> will be multiplied by 10 before being added to the result.

A <sub>1</sub>	result
1	150
8	220

*Result*

The second result attribute has been given an alias ('result') which will be used in the resulting relation.



# Comparison operators

A number of comparison operators are available to use in the WHERE clause. In the examples above only the equality operator '=' was used.

operator	meaning
=	equals
<>	not equals
!=	not equals
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
BETWEEN	within an inclusive range
LIKE	containing a pattern
IN	in a specified set

# Comparison operators

Consider (the expanded) relation  $R_1$

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22
2	3.4	'CDK'	50
5	0.2	'DEF'	87
9	9.4	'ECG'	120

$R_1$

# Comparison operators: 'not equal'

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22
2	3.4	'CDK'	50
5	0.2	'DEF'	87
9	9.4	'ECG'	120

$R_1$

```
SELECT *  
FROM  $R_1$   
WHERE  $A_3$  <> 'CDK'
```

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
8	0.7	'ABB'	22
5	0.2	'DEF'	87
9	9.4	'ECG'	120

# Comparison operators: 'less than'

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22
2	3.4	'CDK'	50
5	0.2	'DEF'	87
9	9.4	'ECG'	120

$R_1$

```
SELECT *  
FROM  $R_1$   
WHERE  $A_4 < 75$ 
```

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22
2	3.4	'CDK'	50

# Comparison operators: BETWEEN

```
SELECT *  
FROM  $R_1$   
WHERE  $A_2$  BETWEEN 3.4  
AND 8.5
```

Note that 3.4 is included in the result. Whether the boundary cases are included in the result or not is implementation dependent. (On *Postgres* the boundaries are included).

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22
2	3.4	'CDK'	50
5	0.2	'DEF'	87
9	9.4	'ECG'	120

$R_1$

$A_1$	$A_2$	$A_3$	$A_4$
4	4.5	'CDK'	30
2	3.4	'CDK'	50

Result

# Comparison operators: NOT BETWEEN

```
SELECT *  
FROM  $R_1$   
WHERE  $A_2$  NOT BETWEEN  
3.4 AND 8.5
```

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22
2	3.4	'CDK'	50
5	0.2	'DEF'	87
9	9.4	'ECG'	120

$R_1$

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
8	0.7	'ABB'	22
5	0.2	'DEF'	87
9	9.4	'ECG'	120

# Comparison operators: IN

```
SELECT *  
FROM R1  
WHERE A3 IN ('ABB', 'DEF')
```

Tuples will be selected where  
the attribute has *any* of the  
values in the specified set.

A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22
2	3.4	'CDK'	50
5	0.2	'DEF'	87
9	9.4	'ECG'	120

*R<sub>1</sub>*

A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>
1	2.5	'ABB'	15
8	0.7	'ABB'	22
5	0.2	'DEF'	87

*Result*

# Comparison operators: LIKE

The LIKE operator enables comparison with a pattern. This is achieved using 'wildcards'. The following wildcards can be used.

wildcard	meaning
%	matches 0 or more characters
_	(underscore) matches any single character



# Comparison operators: LIKE

```
SELECT *  
FROM  $R_1$   
WHERE  $A_3$  LIKE '%BB'
```

Note that the following query  
(for example) gives the same  
result):

```
SELECT *  
FROM  $R_1$   
WHERE  $A_3$  LIKE  
'%%B%B%%'
```

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
8	0.7	'ABB'	22
2	3.4	'CDK'	50
5	0.2	'DEF'	87
9	9.4	'ECG'	120

$R_1$

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
8	0.7	'ABB'	22

*Result*

# Comparison operators: LIKE

```
SELECT *  
FROM  $R_1$   
WHERE  $A_3$  LIKE 'E_G'
```

Note that the following query  
(for example) has an empty  
result.

```
SELECT *  
FROM  $R_1$   
WHERE  $A_3$  LIKE 'E_G_'
```

Whilst this query (for example)  
gives the same result.

```
SELECT *  
FROM  $R_1$   
WHERE  $A_3$  LIKE 'E_G%'
```

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
8	0.7	'ABB'	22
2	3.4	'CDK'	50
5	0.2	'DEF'	87
9	9.4	'ECG'	120

$R_1$

$A_1$	$A_2$	$A_3$	$A_4$
9	9.4	'ECG'	120

Result

# Boolean connectives

The binary connectives AND and OR, and the unary operator NOT, can be used to form more complex conditions.

# Boolean connectives: AND

```
SELECT *  
FROM  $R_1$   
WHERE  
 $A_2 < 3.0$  AND  $A_4 > 15$ 
```

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22
2	3.4	'CDK'	50
5	0.2	'DEF'	87
9	9.4	'ECG'	120

$R_1$

$A_1$	$A_2$	$A_3$	$A_4$
8	0.7	'ABB'	22
5	0.2	'DEF'	87

*Result*

# Boolean connectives: OR

```
SELECT *  
FROM R1  
WHERE  
A2 < 3.0 OR A3 = 'ABB'
```

A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22
2	3.4	'CDK'	50
5	0.2	'DEF'	87
9	9.4	'ECG'	120

*R<sub>1</sub>*

A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>
1	2.5	'ABB'	15
8	0.7	'ABB'	22
5	0.2	'DEF'	87

*Result*

# Boolean connectives: multiple

```
SELECT *  
FROM  $R_1$   
WHERE  
 $A_3 \neq \text{'ABB'}$  AND  
( $A_1 = 2$  OR  $A_1 \neq 9$ )
```

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
4	4.5	'CDK'	30
8	0.7	'ABB'	22
2	3.4	'CDK'	50
5	0.2	'DEF'	87
9	9.4	'ECG'	120

$R_1$

Note the alternative form of the 'not equals' operator.

$A_1$	$A_2$	$A_3$	$A_4$
4	4.5	'CDK'	30
2	3.4	'CDK'	50
5	0.2	'DEF'	87

Result

# Ordering the result

There is no order to the results given by an SQL query unless the result is specifically ordered. In practice the same query can produce a different ordering each time it is run.

# ORDER BY, one column

The ORDER BY clause sorts the tuples in the result of an SQL query. The result can be sorted by one or more columns.

```
SELECT *  
FROM  $R_1$   
ORDER BY  $A_2$ 
```

$A_1$	$A_2$	$A_3$	$A_4$
5	0.2	'DEF'	87
8	0.7	'ABB'	22
1	2.5	'ABB'	15
2	3.4	'CDK'	50
4	4.5	'CDK'	30
9	9.4	'ECG'	120

*Result*



# ORDER BY, multiple columns

The ORDER BY clause sorts the tuples in the result of an SQL query. The result can be sorted by one or more columns.

```
SELECT *  
FROM  $R_1$   
ORDER BY  $A_3$ ,  $A_4$ 
```

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
8	0.7	'ABB'	22
4	4.5	'CDK'	30
2	3.4	'CDK'	50
5	0.2	'DEF'	87
9	9.4	'ECG'	120

*Result*

# Ordering the result

The previous examples used the default ordering method in SQL which is ascending order. This can also be optionally specified with the keyword `ASC`. To achieve an ordering in descending order `DESC` must be specified.

# ORDER BY DESC, one column

The ORDER BY clause sorts the tuples in the result of an SQL query. The result can be sorted by one or more columns.

```
SELECT *  
FROM  $R_1$   
ORDER BY  $A_2$  DESC
```

$A_1$	$A_2$	$A_3$	$A_4$
9	9.4	'ECG'	120
4	4.5	'CDK'	30
2	3.4	'CDK'	50
1	2.5	'ABB'	15
8	0.7	'ABB'	22
5	0.2	'DEF'	87

*Result*

# ORDER BY DESC, multiple columns

The ORDER BY clause sorts the tuples in the result of an SQL query. The result can be sorted by one or more columns. This example demonstrates that the columns can be sorted independently. DESC applies only to  $A_4$  in this example.

$A_1$	$A_2$	$A_3$	$A_4$
8	0.7	'ABB'	22
1	2.5	'ABB'	15
2	3.4	'CDK'	50
4	4.5	'CDK'	30
5	0.2	'DEF'	87
9	9.4	'ECG'	120

*Result*

```
SELECT *  
FROM  $R_1$   
ORDER BY  $A_3$ ,  $A_4$  DESC
```

# Limiting the number of tuples in the result

If relations have a large number of rows then the result can be limited to a specified number of tuples using `LIMIT`

# ORDER BY, LIMIT

The LIMIT clause does not have to be used with ORDER BY but if it is not the tuples in the result are essentially randomly selected from the relation.

```
SELECT *  
FROM  $R_1$   
ORDER BY  $A_4$  LIMIT 3
```

This query gives the tuples with the three lowest values of  $A_4$ .

$A_1$	$A_2$	$A_3$	$A_4$
1	2.5	'ABB'	15
8	0.7	'ABB'	22
4	4.5	'CDK'	30

*Result*

# Aggregation

SQL has aggregate functions that enable the computation of counts, maximum values, minimum values, average values, etc. Some examples follow.

# COUNT()

The COUNT() function counts the number of rows.

If COUNT(\*) ('count all') is used then the function counts the number of rows in the relation.

```
SELECT COUNT(*)  
FROM R1
```

count
6

*Result*



# COUNT()

If COUNT(attribute-name) is used then the function counts the number of values in that attribute. NULL values will be excluded from the count.

```
SELECT COUNT( $A_2$ )  
FROM  $R_1$ 
```

In this case this will give the same result as COUNT(\*).

count
6

*Result*

# COUNT()

If COUNT(DISTINCT  
attribute-name) is used then the  
function counts the number of  
distinct values in that attribute.

```
SELECT COUNT(DISTINCT  
A3)  
FROM R1
```

count
4

*Result*

# COUNT()

COUNT() can be used in conjunction with WHERE to restrict the result.

```
SELECT COUNT( $A_2$ )  
FROM  $R_1$   
WHERE  $A_2 > 1.0$ 
```

count
4

*Result*

# MAX()

MAX() returns the maximum value of an attribute.

```
SELECT MAX( $A_2$ )  
FROM  $R_1$ 
```

MAX() works with non-numeric data. Changing  $A_2$  to  $A_3$  in the above query will return the result 'ECG'.

count
9.4

*Result*

# MAX()

MAX() can also be used in conjunction with WHERE.

```
SELECT MAX( $A_2$ )  
FROM  $R_1$   
WHERE  $A_2 < 5$ 
```

count
4.5

*Result*

# GROUP BY

The GROUP BY clause can be used in conjunction with aggregate functions to obtain information about groups of tuples in the table.

# GROUP BY

```
SELECT A3,  
ROUND(AVG(A4))  
FROM R1  
GROUP BY A3
```

This query also demonstrates the use of the ROUND() function.

Conceptually (since we are only using one relation at present), the tuples are first grouped according to the attribute, then the aggregate function is applied to each group, then the SELECT clause is applied to select the desired attributes from the result.

A <sub>3</sub>	round
'DEF'	87
'ECG'	120
'CDK'	40
'ABB'	19

*Result*

# WHERE, GROUP BY

```
SELECT  $A_3$ ,  
ROUND(AVG( $A_4$ ))  
FROM  $R_1$   
WHERE  $A_1 \neq 9$   
GROUP BY  $A_3$ 
```

The WHERE clause is applied before the GROUP BY clause, removing tuples from  $R_1$  that don't meet the condition. The order of processing is then the same as for the previous example.

$A_3$	round
'DEF'	87
'CDK'	40
'ABB'	19

*Result*



# GROUP BY, HAVING

The HAVING clause enables us to express a condition that applies to groups rather than to individual tuples.

```
SELECT A3,  
ROUND(AVG(A4))  
FROM R1  
GROUP BY A3  
HAVING  
ROUND(AVG(A4)) < 80
```

A <sub>3</sub>	round
'CDK'	40
'ABB'	19

*Result*

The HAVING clause is evaluated after the tuples have been grouped by GROUP BY.

# Querying multiple relations

Generally databases contain multiple relations that are related to each other through *keys*. In this context a query will consist of somehow *joining* multiple tables together based on the keys and then selecting the tuples and attributes required in the result.

There are multiple ways of joining relations. We will now look at some of them.

# Some sample relations

We will use the sample relations below for the following examples.

```
projects=# select * from employee;
```

empid	lname	location	salary	manager
0	Johnson	London	30000	Khan
1	Khan	London	45000	Davis
2	Wilson	Oxford	28000	Peters
3	Cuthbert	Oxford	21000	Peters
4	Peters	St Albans	50000	Davis
5	Davis	London	75000	

(6 rows)

```
projects=# select * from project;
```

projectid	projectname	projectleader	budget
0	Vanity	Peters	100000
1	White Elephant	Davis	50000
2	Infeasible	Peters	125000

(3 rows)

```
projects=# select * from projectemps;
```

projectid	empid	contract
0	0	3
0	1	5
1	2	3
1	3	2
1	1	3
2	3	1
2	4	5

(7 rows)

# Querying multiple relations

This small database has data relating to projects, employees and employees working on projects.

# Cross Join

The Cross Join can be achieved as shown below and computes the cross product of the relations given. The 'cross product' operation is signified by the ',' placed between the relation names in the FROM clause. It can be seen that each of the tuples in the first relation is paired with each of the tuples from the second relation. This is not usually a useful result without restriction on the rows in the result.

```
SELECT *  
FROM employee,  
projectemps
```

0	Johnson	London	30000	Khan	0	0	3
1	Khan	London	45000	Davis	0	0	3
2	Wilson	Oxford	28000	Peters	0	0	3
3	Cuthbert	Oxford	21000	Peters	0	0	3
4	Peters	St Albans	50000	Davis	0	0	3
5	Davis	London	75000		0	0	3
0	Johnson	London	30000	Khan	0	1	5
1	Khan	London	45000	Davis	0	1	5
2	Wilson	Oxford	28000	Peters	0	1	5
3	Cuthbert	Oxford	21000	Peters	0	1	5
4	Peters	St Albans	50000	Davis	0	1	5
5	Davis	London	75000		0	1	5
0	Johnson	London	30000	Khan	1	2	3
1	Khan	London	45000	Davis	1	2	3
2	Wilson	Oxford	28000	Peters	1	2	3
3	Cuthbert	Oxford	21000	Peters	1	2	3
4	Peters	St Albans	50000	Davis	1	2	3
5	Davis	London	75000		1	2	3
0	Johnson	London	30000	Khan	1	3	2
1	Khan	London	45000	Davis	1	3	2
2	Wilson	Oxford	28000	Peters	1	3	2
3	Cuthbert	Oxford	21000	Peters	1	3	2
4	Peters	St Albans	50000	Davis	1	3	2
5	Davis	London	75000		1	3	2
0	Johnson	London	30000	Khan	1	1	3
1	Khan	London	45000	Davis	1	1	3
2	Wilson	Oxford	28000	Peters	1	1	3
3	Cuthbert	Oxford	21000	Peters	1	1	3
4	Peters	St Albans	50000	Davis	1	1	3
5	Davis	London	75000		1	1	3
0	Johnson	London	30000	Khan	2	3	1
1	Khan	London	45000	Davis	2	3	1
2	Wilson	Oxford	28000	Peters	2	3	1
3	Cuthbert	Oxford	21000	Peters	2	3	1
4	Peters	St Albans	50000	Davis	2	3	1
5	Davis	London	75000		2	3	1
0	Johnson	London	30000	Khan	2	4	5
1	Khan	London	45000	Davis	2	4	5
2	Wilson	Oxford	28000	Peters	2	4	5
3	Cuthbert	Oxford	21000	Peters	2	4	5
4	Peters	St Albans	50000	Davis	2	4	5
5	Davis	London	75000		2	4	5

(42 rows)

# Cross Join

The Cross Join can be explicitly requested in the FROM clause with the same result.

```
SELECT *  
FROM employee  
CROSS JOIN  
projectemps
```

0	Johnson	London	30000	Khan	0	0	3
1	Khan	London	45000	Davls	0	0	3
2	Wilson	Oxford	28000	Peters	0	0	3
3	Cuthbert	Oxford	21000	Peters	0	0	3
4	Peters	St Albans	50000	Davls	0	0	3
5	Davls	London	75000		0	1	5
0	Johnson	London	30000	Khan	0	1	5
1	Khan	London	45000	Davls	0	1	5
2	Wilson	Oxford	28000	Peters	0	1	5
3	Cuthbert	Oxford	21000	Peters	0	1	5
4	Peters	St Albans	50000	Davls	0	1	5
5	Davls	London	75000	Davls	0	1	5
0	Johnson	London	30000	Khan	1	2	3
1	Khan	London	45000	Davls	1	2	3
2	Wilson	Oxford	28000	Peters	1	2	3
3	Cuthbert	Oxford	21000	Peters	1	2	3
4	Peters	St Albans	50000	Davls	1	2	3
5	Davls	London	75000	Davls	1	2	3
0	Johnson	London	30000	Khan	1	3	2
1	Khan	London	45000	Davls	1	3	2
2	Wilson	Oxford	28000	Peters	1	3	2
3	Cuthbert	Oxford	21000	Peters	1	3	2
4	Peters	St Albans	50000	Davls	1	3	2
5	Davls	London	75000	Davls	1	3	2
0	Johnson	London	30000	Khan	1	1	3
1	Khan	London	45000	Davls	1	1	3
2	Wilson	Oxford	28000	Peters	1	1	3
3	Cuthbert	Oxford	21000	Peters	1	1	3
4	Peters	St Albans	50000	Davls	1	1	3
5	Davls	London	75000	Davls	1	1	3
0	Johnson	London	30000	Khan	2	3	1
1	Khan	London	45000	Davls	2	3	1
2	Wilson	Oxford	28000	Peters	2	3	1
3	Cuthbert	Oxford	21000	Peters	2	3	1
4	Peters	St Albans	50000	Davls	2	3	1
5	Davls	London	75000	Davls	2	3	1
0	Johnson	London	30000	Khan	2	4	5
1	Khan	London	45000	Davls	2	4	5
2	Wilson	Oxford	28000	Peters	2	4	5
3	Cuthbert	Oxford	21000	Peters	2	4	5
4	Peters	St Albans	50000	Davls	2	4	5
5	Davls	London	75000	Davls	2	4	5

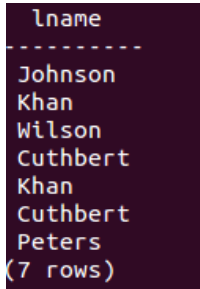
42 rows)

# Inner Join

The Cross Join isn't usually very useful. When we combine relations to produce a result it is more common to wish to restrict the tuples in the result to those that are related in some way. The Inner Join allows us to join the tables based on shared values.

```
SELECT  
employee.lname  
FROM employee,  
projectemps  
WHERE  
employee.empID =  
projectemps.empID
```

Note that we could  
use `DISTINCT` to  
remove duplicate  
tuples.



```
lname  
-----  
Johnson  
Khan  
Wilson  
Cuthbert  
Khan  
Cuthbert  
Peters  
(7 rows)
```

# Inner Join

This Inner Join works by first computing the Cross Join (which is implied by the ',' between the relation names), and then restricting the tuples in the result to only those meeting the condition in the WHERE clause. Finally, the SELECT clause excludes all of the attributes except for employee.lname.

```
SELECT employee.lname  
FROM employee, projectemps  
WHERE employee.empID = projectemps.empID
```



# Inner Join

The Inner Join also has an explicit form. This form of the Inner Join explicitly joins the tables on a specific condition, using an ON clause.

```
SELECT employee.lname  
FROM employee INNER JOIN projectemps  
ON employee.empID = projectemps.empID
```

It is considered good practice to use this syntax since it is considered more readable. Readability (as well as correctness, of course) should generally be the priority in formulating queries.

# Natural Join

In the case where the relations to be joined have a common attribute names (and types), the Natural Join can be used. However, if there are multiple attributes with the same name, only tuples that match on all of the shared attributes will be added to the result.

```
SELECT employee.lname  
employee NATURAL JOIN projectemps
```

It is considered good practice to use this syntax since it is considered more readable. Readability (as well as correctness, of course) should generally be the priority in formulating queries.

# Left Join

The joins described so far exclude tuples where no match is found. This will often be the result sought but there are other situations where it might be useful to add tuples where no match was found to the result. The Outer Joins perform this function in SQL. The Left Join (or Left Outer Join) performs a join on two relations and then adds tuples from the left relation (the one listed first) to the result if they were not already in the result.

```
SELECT project.projectName, project.projectLeader  
project LEFT JOIN projectemps  
ON project.projectID = projectemps.projectID;
```

# Left Join

```
SELECT project.projectName, project.projectLeader  
FROM project LEFT JOIN projectemps  
ON project.projectID = projectemps.projectID;
```

projectname	projectleader
Vanity	Peters
Vanity	Peters
White Elephant	Davis
White Elephant	Davis
White Elephant	Davis
Infeasible	Peters
Infeasible	Peters
Pipe Dream	
(8 rows)	

This allows us to find attributes in one relation that do not have a value in another (there can be a project with no project leader).

# Right Join

The Right Join performs the join and then adds tuples from the right (second) relation to the result if they were not already in the result.

```
SELECT project.projectName, project.projectLeader,  
projectemps.projectID  
FROM project RIGHT JOIN projectemps  
ON project.projectID = projectemps.projectID;
```

# Right Join

```
SELECT project.projectName, project.projectLeader,  
projectemps.projectID  
FROM project RIGHT JOIN projectemps  
ON project.projectID = projectemps.projectID;
```

projectname	projectleader	projectid
Vanity	Peters	0
Vanity	Peters	0
White Elephant	Davis	1
White Elephant	Davis	1
White Elephant	Davis	1
Infeasible	Peters	2
Infeasible	Peters	2
		4
(8 rows)		

Here we find the there is a project ID existing for a project that has no name or leader in the project relation.

# Full Outer Join

The Full Outer Join combines the results of the Left Join and Right Join.

```
SELECT project.projectName, project.projectLeader,  
projectemps.projectID  
FROM project FULL OUTER JOIN projectemps  
ON project.projectID = projectemps.projectID;
```

# Full Outer Join

```
SELECT project.projectName, project.projectLeader,  
projectemps.projectID  
FROM project FULL OUTER JOIN projectemps  
ON project.projectID = projectemps.projectID;
```

projectname	projectleader	projectid
Vanity	Peters	0
Vanity	Peters	0
White Elephant	Davis	1
White Elephant	Davis	1
White Elephant	Davis	1
Infeasible	Peters	2
Infeasible	Peters	2
		4
Pipe Dream		
(9 rows)		



# Self Joins

Sometimes it is necessary to join a relation with itself. This would be required when the tuples in the relation need to be compared with each other to answer a particular query. For example, with the employee relation, we might wish to ask which employees have the same manager. All of the information we need is in the employee table. Of course, in this case we must use aliases for the relations otherwise ambiguity is inevitable.

```
SELECT DISTINCT e1.lname,e2.manager
FROM employee e1, employee e2
WHERE e1.manager = e2.manager AND e1.lname <> e2.lname
```

The aliases are created in the FROM clause and then may be used in the WHERE and SELECT clauses.

lname	manager
Cuthbert	Peters
Wilson	Peters
Peters	Davis
Khan	Davis
(4 rows)	

# NULL values

A tuple may have NULL values for some attributes (provided they are not part of the primary key or don't have the NOT NULL constraint declared). NULL values must be considered when formulating queries.

NULL values cannot be compared with other values using the usual comparison operators. The result of such a comparison is 'unknown'. For example, NULL is not less than 10 or equal to 10 or greater than 10. NULL is not equal to zero or even equal to NULL.

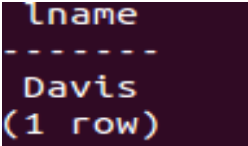
We can actively seek NULL values in SQL by using the IS NULL operator. There is also an IS NOT NULL operator.

NULL values are ignored by aggregate functions except for COUNT(\*) which will count tuples with NULL values.

NULL values are not ignored by a GROUP BY clause. All NULLs are grouped into the same group.

# NULL values

```
SELECT employee.lname  
FROM employee  
WHERE manager IS NULL
```



A terminal window with a dark background and light-colored text. It displays the output of a SQL query. The first line is 'lname', followed by a dashed line separator. The next line is 'Davis', and the final line is '(1 row)'.

```
lname  
-----  
Davis  
(1 row)
```

# NULL values

```
SELECT *  
FROM project  
WHERE projectLeader IS NOT NULL
```

projectid	projectname	projectleader	budget
0	Vanity	Peters	100000
1	White Elephant	Davis	50000
2	Infeasible	Peters	125000

(3 rows)

# Set operations

Queries can be combined using the set operations UNION, INTERSECT and EXCEPT, provided they are 'union-compatible'. This means that they must have the same number of attributes, in the same order, and with the same data types.

UNION can be used to combine results from two queries ('add' them together).

INTERSECT can be used to select only those results from the first query that are **also** in the result of the second query.

EXCEPT can be used to remove the results from the first query that are not in the second query.

# Set operations

Consider this new relation called 'CompletedProjects' in the 'projects' database. It contains data about projects that have been completed.

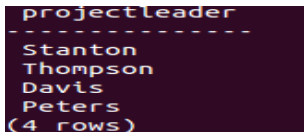
projectid	projectname	projectleader	budget
0	Gherkin	Stanton	100000
1	The Bull Ring	Thompson	50000
2	Millenium Dome	Davis	125000

(3 rows)

# Set operations: UNION

This query retrieves the names of all project leaders past and present.

```
SELECT projectleader
FROM project
WHERE projectleader IS NOT NULL
UNION
SELECT projectleader
FROM completedproject
WHERE projectleader IS NOT NULL
```



```
projectleader
-----
Stanton
Thompson
Davis
Peters
(4 rows)
```

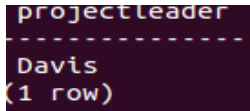
Note that by definition (since it is a set operation), UNION removes duplicates. UNION ALL can be used instead to retain them.

# Set operations: INTERSECT

This query retrieves the names of all project leaders who have led a project in the past and are leading one now.

```
SELECT projectleader
FROM project
WHERE projectleader IS NOT NULL
INTERSECT
SELECT projectleader
FROM completedproject
WHERE projectleader IS NOT NULL
```

Note that this was as simple as changing the word 'UNION' to 'INTERSECT'.



```
projectleader
-----
Davis
(1 row)
```

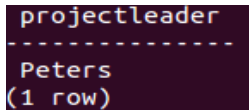


# Set operations: EXCEPT

This query retrieves the names of all project leaders who are leading a project now but have not in the past.

```
SELECT projectleader
FROM project
WHERE projectleader IS NOT NULL
EXCEPT
SELECT projectleader
FROM completedproject
WHERE projectleader IS NOT NULL
```

Again, Note that this was as simple as changing the word 'UNION' to 'EXCEPT'.



```
projectleader
-----
Peters
(1 row)
```

# Subqueries

Queries can run subqueries in order to get a result. A query that has a subquery is called a *nested query*. Subqueries can be placed in the SELECT, FROM or WHERE clause of a SELECT statement.

# Subqueries in the WHERE clause

A subquery in the WHERE clause is used to generate a value or a relation that will be used in the WHERE condition. This can be used to restrict the tuples in the result using more complex conditions than might otherwise be possible, or at least to create a more readable condition.

Common operators to use with subqueries in the WHERE clause are IN, EXISTS, ANY, ALL (combined with NOT).

# IN

As previously discussed, the IN operator enables a condition in which a value is checked against a *set* of values. In the previous example, the set was explicitly defined. We can, however, build the set dynamically from the database. This will often be more useful. IN also allows us to avoid some of the problems caused by duplicates (see below).

# IN

This query selects the project name and employee name for all employees who are based in London.

```
SELECT project.projectname, employee.lname  
FROM projectEmps, project, employee  
WHERE projectemps.projectid = project.projectid and  
employee.empid = projectemps.empid  
AND projectemps.empid IN  
(SELECT employee.empid  
FROM employee  
WHERE employee.location = 'London');
```

projectname	lname
Vanity	Johnson
Vanity	Khan
White Elephant	Khan
(3 rows)	

# IN

The query on the previous slide can be formulated easily with a JOIN without using a subquery. The advantage of using IN can be that anomalies caused by duplicate values can be more easily avoided. The IN condition simply tests if the value is in the set of results from the subquery. A JOIN may return multiple tuples for the same value which might cause an incorrect result.

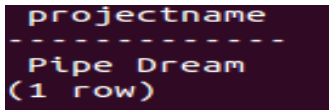
The use of IN with a subquery can also result in a more readable query, although not in the case of the query on the previous slide.

# NOT IN

More intuitive queries can be formulated to check whether a value is NOT in a set using NOT IN.

This query returns the projects that have no employees allocated to them.

```
SELECT project.projectname  
FROM project  
WHERE project.projectid NOT IN (select projectid from  
projectemps);
```



```
projectname  
-----  
Pipe Dream  
(1 row)
```

# EXISTS

The EXISTS operator allows tuples to be included based on a non-empty result of a subquery. If the subquery returns any rows at all, the tuple is included in the result.

This query finds employees who have another employee in the same location.

```
SELECT e1.lname, e1.location
FROM employee e1
WHERE EXISTS (
  SELECT *
  FROM employee e2
  WHERE e1.emplid <> e2.emplid and e1.location = e2.location)
```



# EXISTS

lname	location
Johnson	London
Khan	London
Wilson	Oxford
Cuthbert	Oxford
Davis	London
(5 rows)	

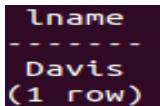
Note that, in this query, the condition in the inner query specifically excludes employees where their ID is the same as in the outer query (i.e. they are the same person).

# NOT EXISTS

NOT EXISTS is, as expected, the opposite of EXISTS and enables us to select tuples based on an empty result of a subquery. This can be used to find maximum and minimum values, for example, without using the aggregate functions

This query finds the employee with the highest salary.

```
SELECT e1.lname
FROM employee e1
WHERE NOT EXISTS
(SELECT *
FROM employee e2
WHERE e2.salary > e1.salary)
```



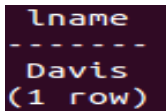
```
lname
-----
Davis
(1 row)
```

# ALL

In fact, we can formulate the query on the previous slide more naturally using another operator - the ALL operator. ALL allows a comparison to be made with all of the results in a subquery.

Here again, this query finds the employee with the highest salary.

```
SELECT lname  
FROM employee  
WHERE salary >= ALL  
(SELECT salary  
FROM employee)
```

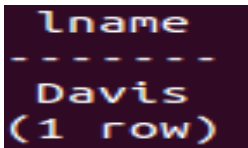


```
lname  
-----  
Davis  
(1 row)
```

# ANY

For completeness, the ANY operator enables a comparison with the result of a subquery such that a value has a relationship with at least one row of the subquery result. We can formulate the 'highest salary' query yet again but this time with ANY.

```
SELECT lname  
FROM employee  
WHERE NOT salary <= ANY  
(SELECT salary  
FROM employee)
```



```
lname  
-----  
Davis  
(1 row)
```

# Subqueries in the FROM clause

A subquery in the FROM clause creates a temporary relation that can be queried or joined to another relation. Again this effect can be achieved with other means.

The example query on the next slide demonstrates a number of things. It contains a subquery in the in the FROM clause, hence creates a temporary table. Such temporary tables must be given an alias (name) so that they can be referred to elsewhere in the query. This query also demonstrates how the results of the subquery can be joined to another relation, and how joins can be composed (i.e. the results of one join be 'passed' to another join).

# Subqueries in the FROM clause

This (artificial) query finds the name of projects that have employees working on them who have salaries greater than 30000.

```
SELECT distinct projectname  
FROM  
((SELECT empid, lname FROM employee where salary > 30000)  
AS temp NATURAL JOIN projectemps)  
NATURAL JOIN project
```

# Subqueries in the SELECT clause

A subquery in the SELECT clause can be used to generate an attribute and to perform calculations on the data.

The query on the next slide demonstrates this.

# Subqueries in the SELECT clause

This (artificial) query finds the percentage of employees who earn more than the average salary. It looks convoluted (and is).

SELECT

```
(round ((SELECT 1.0 *count(salary) FROM employee WHERE  
salary > (SELECT avg(salary) FROM employee)) /  
(1.0* (SELECT count(salary) FROM employee)) * 100))
```

Notes: the expressions involved '1.0\*' to convert the arithmetic from integer to floating point arithmetic; the form of this statement is relatively simple: ('count the number of salaries greater than the average salary'/'count the total number of salaries') \* 100.