University of Glasgow

Data Storage and Retrieval

Lecture 9

More Structured Query Language (SQL)

Dr. Graham McDonald

Graham.McDonald@glasgow.ac.uk

University of Glasgow

Basic SQL - reminder

```
SELECT attribute(s)
    FROM table(s)
        WHERE condition(s)
```

Writing SQL Queries for Questions

- "What are the names of the pets that live at house 42?"
  - Identify the projections
    - aname
  - Identify the relation(s)
    - Pets
  - Identify the constraints
    - housenum = 42
  - Decide on the **form** of the query
    - `SELECT aname FROM Pets WHERE housenum = 42`

University of Glasgow

Writing SQL Queries for Questions

- Writing the SQL
  - Identify the projections
  - Identify the relation(s)
  - Identify the constraints
  - **Decide on the form of the query**

- Different **forms** of queries are available
  - Do we use joins? What do we project? What do we select?
  - We're now going to highlight some **SQL Design Patterns**
  - So you don't need to invent your own query form each time

University of Glasgow

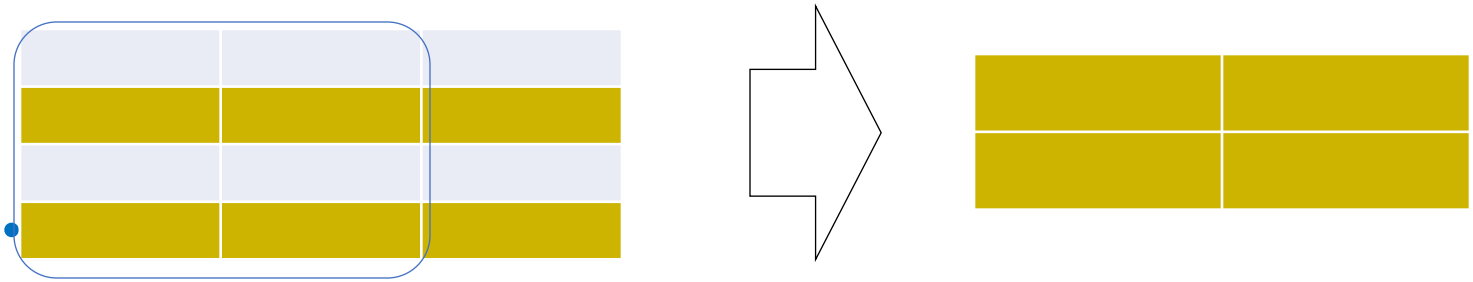
SQL Design Patterns

- **Design patterns** are commonly used in computing science
  - Unfinished but **reusable** designs for commonly occurring problem types
  - May represent good practices, e.g. for software engineering

- SQL Design Patterns help you to select the appropriate form of a query, such as:
  - Basic query
  - Equi-Join
  - Self-join

The Basic Query Pattern

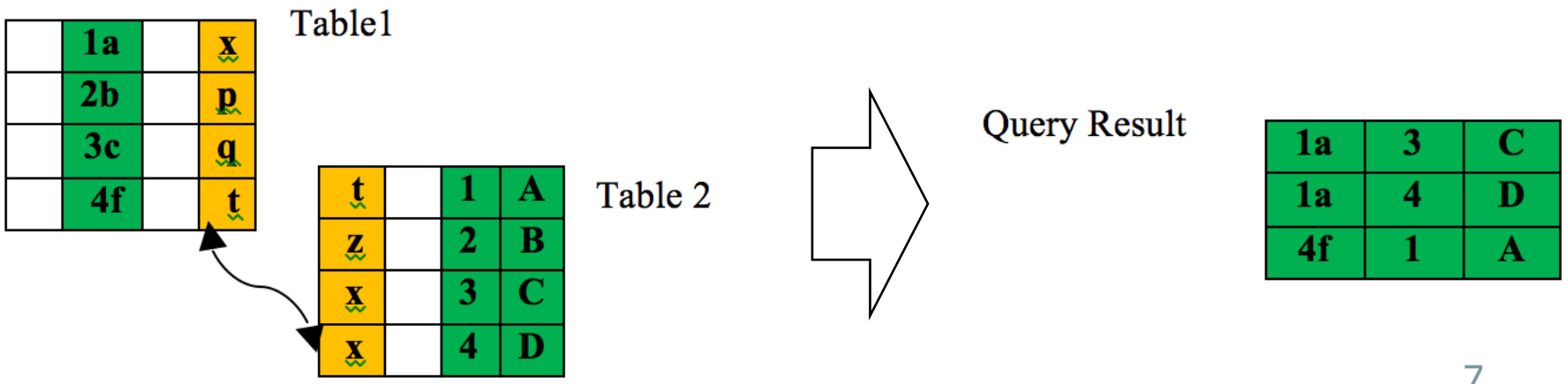
- *"What are the names of the pets at house number 42?"*
- **Basic Query** pattern is **USED WHEN** (all the data are in one table) **AND** (rows need to be filtered based on a simple static condition)



  - SELECT *column1*, *column2*
  - FROM *table*
  - [WHERE column_name operator literal_value]
- `SELECT aname FROM Pets WHERE housenum = 42`

Equi-Join Pattern

- *"List all information of employees and their department"*
- **Problem:** How can you gather information that is distributed across more than one table?
- **Equi-join Pattern** is **USED WHEN** (all the data are in more than one table) **AND** (rows need to be filtered based on data in other rows in the tables)



Equi-Join: The Syntax

- FORMAT:
  - SELECT *column1*, *column2*
  - FROM *table*1, *table*2
  - WHERE table1.column_name = table2.column_name ⬅ join condition
  - [AND condition]
- EXAMPLE: *"List the information of all employees and their department"*
  - `SELECT Employee_ID, Department_ID, Department_Name`
  - `FROM Employees as E, Departments as D`
  - `WHERE E.Department_ID = D.Department_ID`

  OR, if the attributes names are named the same in E & D…
  - `SELECT Employee_ID, Department_ID, Department_Name`
  - `FROM Employees` **`NATURAL JOIN`** `Departments`

University of Glasgow

Fundamental point:

If you have more than one relation listed in the FROM
Then you should be expressing a *join condition* equating rows from those relations,

i.e.

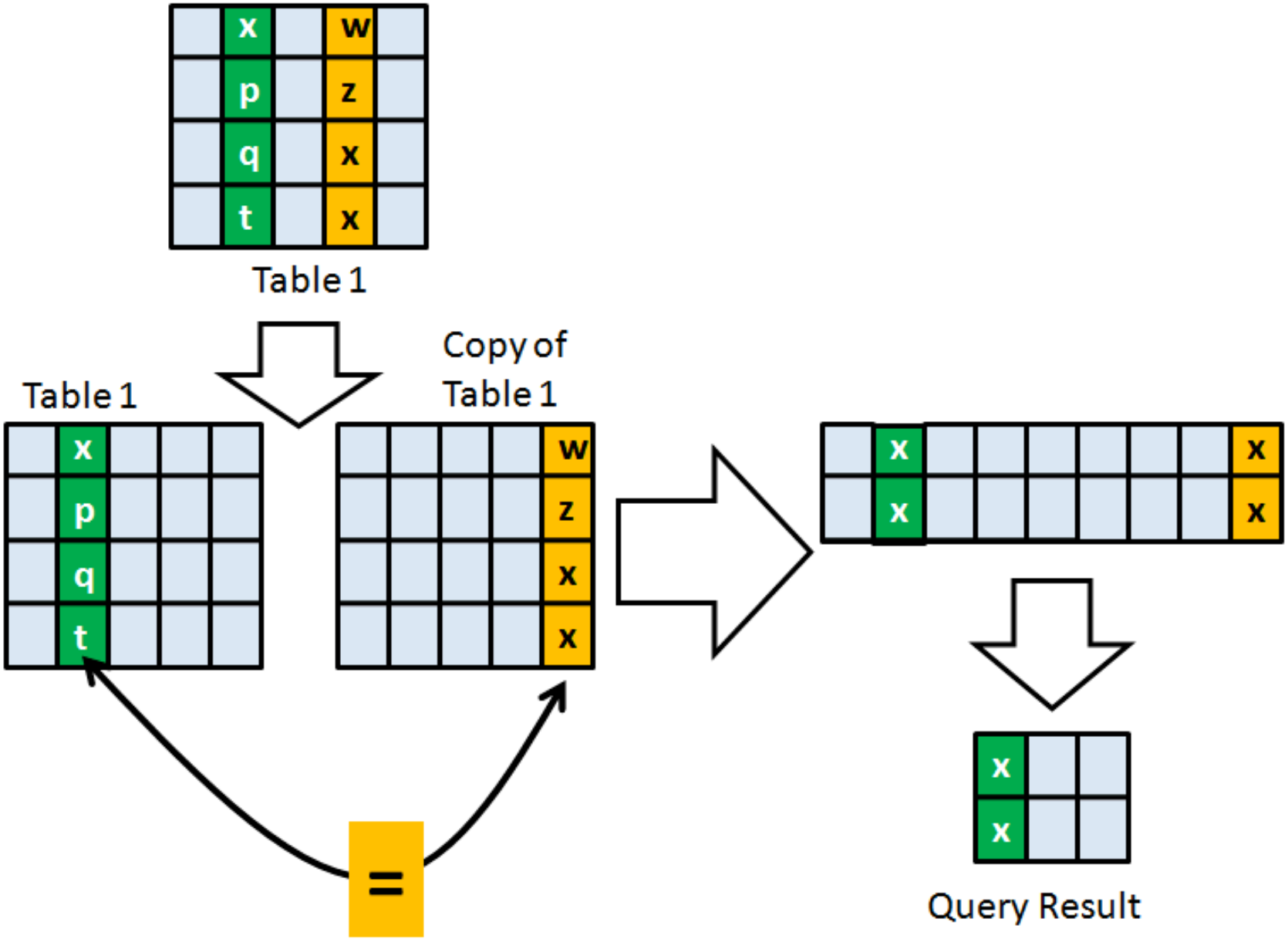
```
SELECT *
FROM A,B
WHERE A.a = B.b
```

University of Glasgow

Self-Join Pattern

- *"Get the names of the people who are supervised by John Smith"*
  - (where all employers details are stored in a single table)
- **PROBLEM:** How can you compare values from different rows in the same column?

- Self-join pattern is **USED WHEN** (all the data are in one table) **AND** (rows need to be filtered based on data in other rows in the same table)


x
w
p
z
q
x
t
x
Table 1
Table 1
Copy of
Table 1
x
p
q
t
w
z
x
x
x
x
x
x
x
x
=
Query Result

Self-Join: The Syntax

- FOCUS: Self-join is appropriate when querying across recursive relationships

- FORMAT:
  - SELECT *column1*, *column2*
  - FROM *table* AS t1, *table AS* t2 (t1, t2 are the table name *aliases*)
  - [WHERE t1.column_name = t2.column_name]

- EXAMPLE: *"Give the names of employees and their managers in dept 5"*
  - `SELECT E.name, M.name`
  - `FROM Employee as E, Employee as M`
  - `WHERE E.supervisor = M.NI# AND E.deptno = 5`

University of Glasgow

Deciding on a SQL Pattern

Data to project is	Condition is	Pattern is
… in one table, on the same row	(…select only some rows)	Basic-query
…in two tables	Anything (but remember you need a join condition)	Equi-Join
…in one table, on different rows	"	Self-join

What are the names of all of the dogs living at house 42?

What are the names of the dogs living at the same house as (person) Jim?

What are the names of the dogs that live at the same house as (animal) Red?

University of Glasgow

More SQL Design Patterns

- SQL Design Patterns help us to identify the correct form of a query
- But we don't know yet how to write queries such as
  - *What is the **average** cost of books authored by 'J K Rowling'?*
  - *What is the cost of each author's **most expensive** book?*
    - These involve projecting "aggregates" of data
- We now examine the SQL notions of Aggregation & Groupings
  - And their corresponding design pattern!

University of Glasgow

Aggregate Functions

- SELECT clause can contain expressions *calculating* data from the columns
- Examples are ***avg*** and ***count***
- In particular, we can use *aggregate* functions:
  - `SELECT AVG(Salary) FROM Employee`
- or
  - `SELECT COUNT(DISTINCT Supervisor) FROM Employee`
    - returns the number of supervisors

Aggregate Functions

- `SUM`, `MAX` and `MIN` are also available

- All of these
  - return a value derived from all the values in a column
  - resulting in a table containing a single record summarising the `Employee` table

University of Glasgow

Summary of Aggregate Functions

Specify a column, and then:

- COUNT (number of values)
- SUM (sum of values)
- AVG (average of values)
- MIN (minimum)
- MAX (maximum)

Recall - The January Census

Person

name	age	houseNum
Jim	15	34
Jo	23	38
Pete	20	38
Jenny	10	42
James	15	48
Paul	15	

Animal

aname	type	houseNum	fedBy
Fluffy	dog	34	Jim
Splodge	cat	34	Jim
Tinky	dog	38	
Robin	dog	42	Jenny
Red	dog	42	Jim
Dusty	snake		Jim

How many houses?

```
SELECT COUNT (houseNum) AS houses
FROM Person;
```

houses
5

```
SELECT COUNT (houseNum) AS houses
FROM Person
WHERE houseNum IS NOT NULL;
```

houses
5

Counts values, not unique values; NULLs are not counted

```
SELECT COUNT (DISTINCT houseNum)
AS houses
FROM Person
WHERE houseNum IS NOT NULL;
```

houses
4

```
SELECT COUNT (age)        AS teenagers
FROM Person
WHERE age >= 13 AND age <= 19;
```

teenagers
3

Count the number of teenagers

```
SELECT SUM (age) AS ageSum
FROM Person;
```

ageSum
98

```
SELECT AVG (age) AS ageAve
FROM Person
WHERE houseNum IS NOT NULL;
```

ageAve
16.6

```
SELECT MIN(age) AS min,
       MAX(age) as max,
       AVG(age) as average
FROM Person
WHERE houseNum IS NOT NULL;
```

min	max	average
10	23	16.6

Grouping

- The previous aggregations selected the rows according to the WHERE condition, and produced a single answer
  - It was said to *aggregate all selected rows*
- GROUP BY allows the aggregations to be applied to groups of rows, according to the grouping of values in a column
- It produces as many answers as there are identified groups

Example Group By Clause

```
SELECT type,  AVG (age) AS ageAve
FROM Animal
GROUP BY type;
```

type	ageAve
cat	6
dog	13.5
fish	1
snake	10

- Produces the average age for each animal type, as follows:
  - creating a group of rows, each containing all of the records with a common animal type
  - then calculating the aggregate for each group
  - the resulting table will have one row for each animal type

University of Glasgow

GROUP BY Syntax

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list;
```

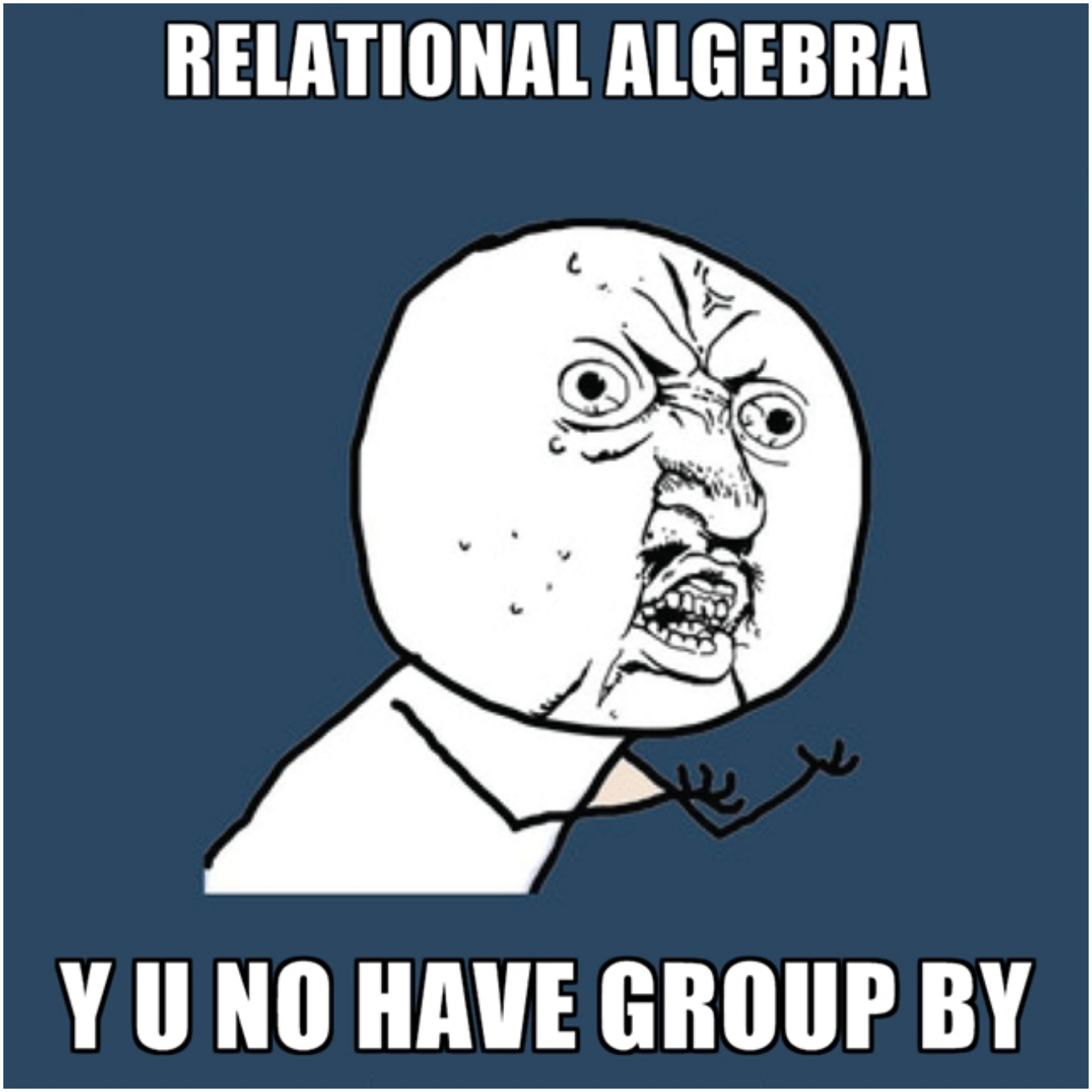
- The *target-list* contains
  1. List of column names
  2. Aggregate terms
  - NOTE: The columns in *target-list* must be either aggregated or contained in *grouping-list*

```
SELECT fedby,  AVG (age) AS ageAve
FROM Animal
GROUP BY fedby;
```

fedby	ageAve
Jim	10.5
Jenny	15
Harriet	8

What is the average age of animals fed by each person?

What we project must *either* be named in the GROUP BY, *or* aggregated


RELATIONAL ALGEBRA
Y U NO HAVE GROUP BY

University of Glasgow

A SQL Design Pattern for Grouping

- *"What is the average age of people in each house?"*
- **Problem**: How do you gather information about a group of rows?

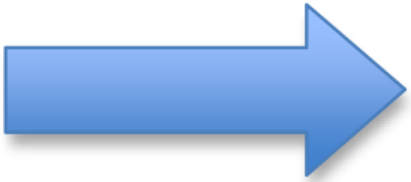
- Grouping pattern is **USED WHEN** (you are looking for a description of a *group* of data in a relation, such as a *count, maximum, minimum, average*, etc) **&** (only one value per group is required)

Seeing "each" and a statistic mentioned
is often a hint that you need a GROUP BY

Grouping Pattern Definition

Relation

A	B	D
Abc	1	20
Xyz	2	10
eee	2	15
www	1	5



Result

B	Max(D)
1	20
2	15

```
SELECT B, MAX(D)
FROM Relation
GROUP BY B
```

Calculate the maximum of D, for each group of B values

- **SELECT** grouping-*columns*, *aggregation_function(column)* **FROM** *relation(s)* [**WHERE** *condition*] **GROUP BY** (grouping-*columns*)

You can select the rows being grouped with WHERE condition(s)

Deciding on a SQL Pattern… UPDATED

Data to project is	Condition is	Pattern is
… in one table, on the same row	(…select only some rows)	Basic-query
…in two tables	Anything (but remember you need a join condition)	Equi-Join
…in one table, on different rows	"	Self-join
Compute a value across a *group* of rows	"	Grouping

What are the names of all of the dogs living at house 42?

What are the names of the dogs living at the same house as (person) Jim?

What are the names of the dogs that live at the same house as (animal) Red?

What is the average age of each pet type?

Basic Query Pattern:

Example question: Get the titles of the books costing more than £5 (where all book details are stored in a single relation)

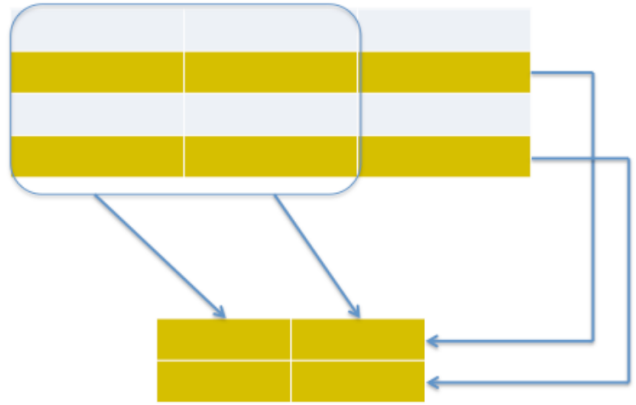
USED WHEN:

(all of the data is in one relation)

AND

(rows of the relation need to be filtered based on a simple static condition)

Identify the rows (orange) to select, then identify the columns (blue box) to project



University of Glasgow

Full Form of a Query

SELECT (**DISTINCT**) <expression> **AS**<name>,….

FROM <table> AS <name>, …

WHERE <search and join condition>

GROUP BY <column>,…

ORDER BY <column>,…

University of Glasgow

Logical Order of Execution

- Form the Cartesian Product of the tables in the **FROM** clause
- Eliminate all rows not satisfying the **WHERE** clause
- Group the remaining rows using the column(s) specified in the **GROUP BY** clause
- Evaluate the expressions in the **SELECT** clause
- Sort by the columns name(s) in the **ORDER BY** clause
- Assign column names as specified in the **AS** clause(s)
- Eliminate duplicates if the keyword **DISTINCT** has been used