Introduction to Database Systems: CS312
Advanced queries, joins and aggregates

Oliver BONHAM-CARTER

31 March 2022

Exam 1: Monday 4^{th} April 3:00pm

Exam1
Joins
New Database
Sets
AS Clauses
Strings
Ordering

Study your SLIDES

- By Honor code: you cannot talk to your colleagues about exam questions
- Released online: Monday, 4^{th} April by 3:00pm
- You will have 24 hours to take the exam,
- Submit to GitHub by 5^{th} April by 3:00pm
- Builder files to create SQLite3 databases
- Creating a database from a datasets: Adding tables and data
- Writing queries for multiple tables
- Populating, updating tables and data
- Using Primary and foreign keys
- Integrity constraints

Joins: Bringing Data Together

Exam1
Joins
Terms
Cross Joins
Inner Join
Cartesian Products
Left, Right
New Database
Sets
AS Clauses
Strings
Ordering

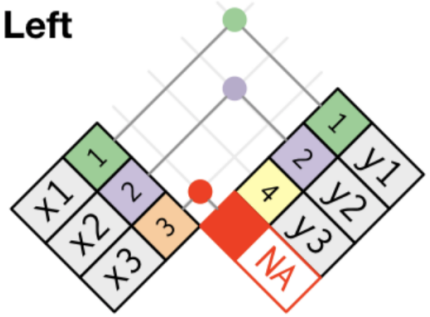
- The SQLite3 `join`-clause is used to combine records from two or more tables in a database.
- A **JOIN** is a means for combining fields from two tables by using values common to each.

Joins: Visual Definitions

Combining Tables

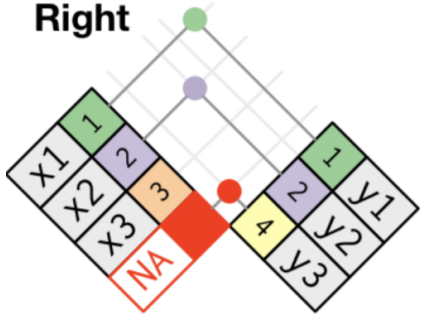
Exam1
Joins
Terms
Cross Joins
Inner Join
Cartesian Products
Left, Right
New Database
Sets
AS Clauses
Strings
Ordering

Left



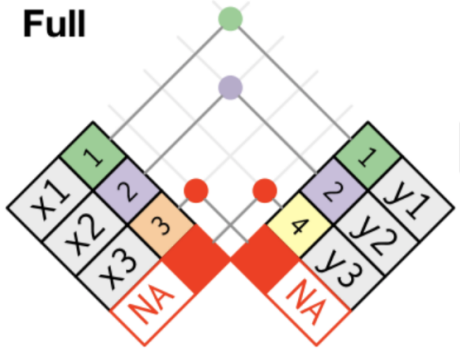
key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA

Right



key	val_x	val_y
1	x1	y1
2	x2	y2
4	NA	y3

Full

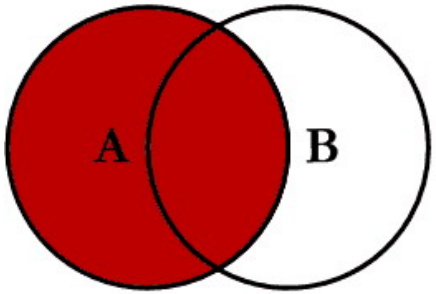


key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA
4	NA	y3

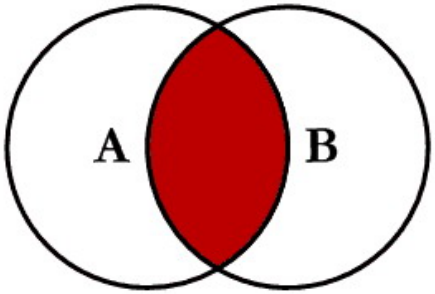
SQL Code and Venn Diagrams

Exam1
Joins
Terms
Cross Joins
Inner Join
Cartesian Products
Left, Right
New Database
Sets
AS Clauses
Strings
Ordering

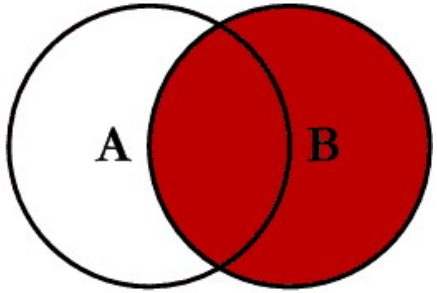
SQL JOINS



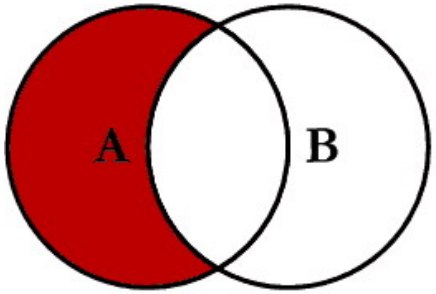

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



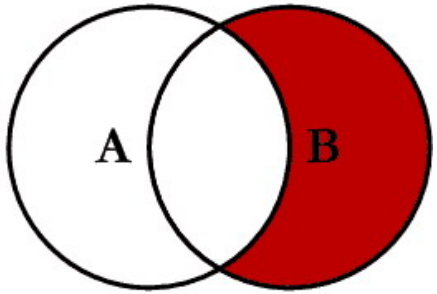

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



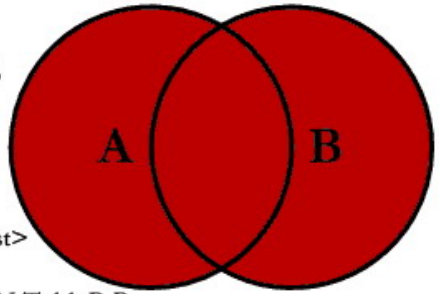

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



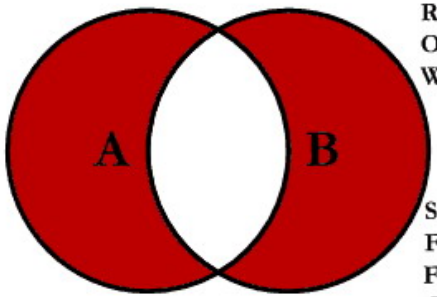

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

© C.L. Moffatt, 2008

An explanation of terms

Exam1
Joins
Terms
Cross Joins
Inner Join
Cartesian Products
Left, Right
New Database
Sets
AS Clauses
Strings
Ordering

- SQL joins
  - The **CROSS JOIN**: Matches every row of the first table with every row of the second table. If the input tables have x and y columns, respectively, the resulting table will have $x * y$ columns.
  - The **INNER JOIN**: Creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate.

Cross joins

Matches every row of the first table with every row of the second table.

Exam1
Joins
Terms
Cross Joins
Inner Join
Cartesian Products
Left, Right
New Database
Sets
AS Clauses
Strings
Ordering

- Cross join: *SELECT ... FROM table1 CROSS JOIN table2 ...*
- Automatically testing for equality between the values of every column that exists in both tables

File: /sandbox/crossJoin.txt

```
 CREATE TABLE ranks (
    rank TEXT NOT NULL
);

CREATE TABLE suits (
    suit TEXT NOT NULL
);

INSERT INTO ranks(rank)
VALUES('2'),('3'),('4'),('5'),('6'),('7'),('8'),('9'),('10'),('J'),('Q'),('K'),('A');

INSERT INTO suits(suit) VALUES('Clubs'),('Diamonds'),('Hearts'),('Spades');

SELECT rank, suit
  FROM ranks
       CROSS JOIN
       suits
ORDER BY suit;
```

Inner joins

Joins two tables where values are equal and disregards the rest.

Exam1
Joins
Terms
Cross Joins
Inner Join
Cartesian Products
Left, Right
New Database
Sets
AS Clauses
Strings
Ordering

File: /sandbox/fruitJoin.txt

```
DROP TABLE TableA;
CREATE TABLE TableA (
fruit VARCHAR,
colour VARCHAR);

DROP TABLE TableB;
CREATE TABLE TableB (
fruit VARCHAR,
colour VARCHAR);

INSERT INTO TableA VALUES ("Lemons_A","Yellow");
INSERT INTO TableA VALUES ("Apples_A","Red");
INSERT INTO TableA VALUES ("Grapes_A","Purple");

INSERT INTO TableB VALUES ("Lemons_B","Yellow");
INSERT INTO TableB VALUES ("Apples_B","Red");
INSERT INTO TableB VALUES ("Oranges_B", "Orange");
INSERT INTO TableB VALUES ("Grapes_B","Purple");


.tables

SELECT * from TableA;
SELECT* from TableB;

SELECT TableA.fruit, TableA.colour, TableB.colour, TableB.fruit
FROM TableA
INNER JOIN
TableB ON TableB.colour == TableA.colour;
```

Inner joins

Joins two tables where values are equal and disregards the rest.

Exam1
Joins
Terms
Cross Joins
Inner Join
Cartesian Products
Left, Right
New Database
Sets
AS Clauses
Strings
Ordering

File: /sandbox/fruitJoin.txt

```
SELECT
  TableA.fruit, TableA.colour,
  TableB.colour, TableB.fruit
FROM
  TableA
INNER JOIN
  TableB ON TableB.colour == TableA.colour;
```

Output

```
Lemons_A|Yellow|Yellow|Lemons_B
Apples_A|Red|Red|Apples_B
Grapes_A|Purple|Purple|Grapes_B
```

Cross Join (Cartesian Product Demo)

Exam1
Joins
Terms
Cross Joins
Inner Join
Cartesian Products
Left, Right
New Database
Sets
AS Clauses
Strings
Ordering

- Inner join: *SELECT ... FROM table1 [INNER] JOIN table2 ON conditional_expression ...*
- Combines column values of two tables (table1 and table2) based upon the join-predicate

Create TableA and TableB

```
drop table tableA;
create table tableA (
 num VARCHAR);

drop table tableB;
create table tableB (
 num VARCHAR);
```

Cross Joins (Cartesian Product Demo)

Exam1
Joins
Terms
Cross Joins
Inner Join
Cartesian Products
Left, Right
New Database
Sets
AS Clauses
Strings
Ordering

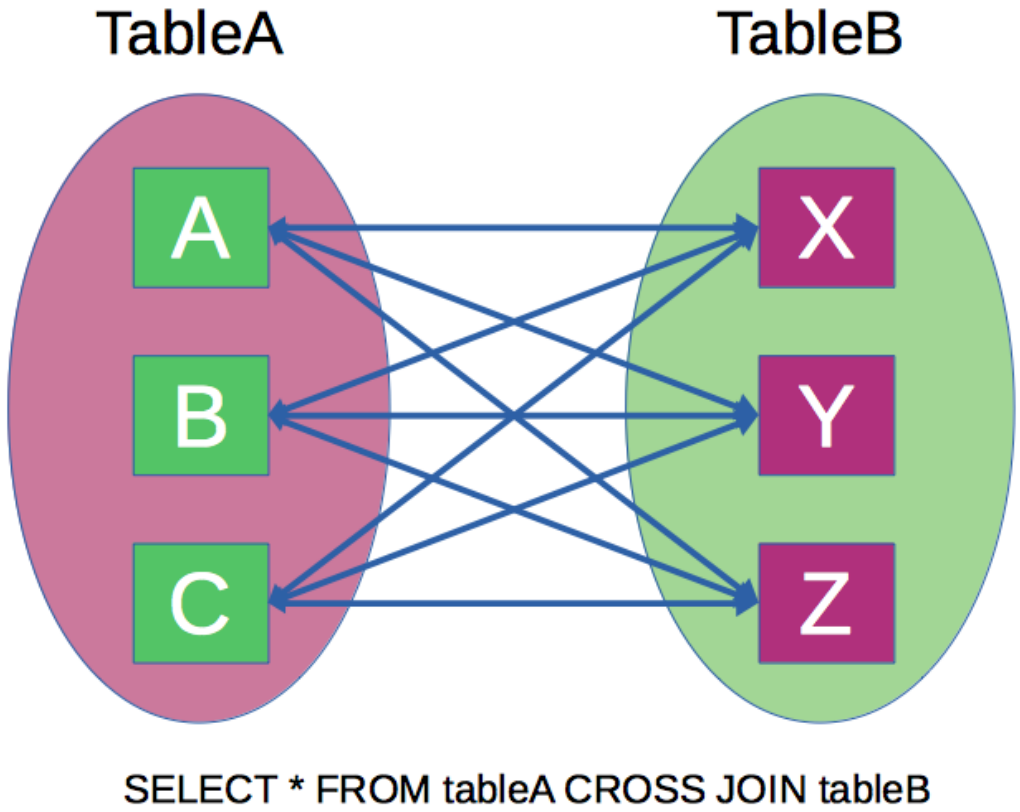
File: crossJoin2.txt

```
INSERT INTO tableA VALUES (1);
INSERT INTO tableA VALUES (2);
INSERT INTO tableA VALUES (3);
INSERT INTO tableA VALUES (4);

INSERT INTO tableB VALUES (1);
INSERT INTO tableB VALUES (2);
INSERT INTO tableB VALUES (3);
INSERT INTO tableB VALUES (4);
INSERT INTO tableB VALUES (5);
INSERT INTO tableB VALUES (6);
INSERT INTO tableB VALUES (7);
INSERT INTO tableB VALUES (8);
INSERT INTO tableB VALUES (9);
```

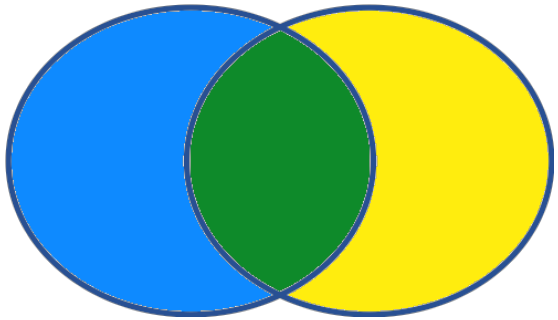
Joins
CROSS JOIN: Cartesian Products

Exam1
Joins
Terms
Cross Joins
Inner Join
Cartesian Products
Left, Right
New Database
Sets
AS Clauses
Strings
Ordering




```
SELECT * from TableA CROSS JOIN TableB;
SELECT * from tableA, TableB;
```

Inner Joins



Left: Blue and Green in Venn Diagram, Above

```
/* inner (left) join */
SELECT TableA.num  FROM TableB  LEFT JOIN TableA  ON TableA.num == TableB.num;
SELECT count(TableA.num)  FROM TableB  LEFT JOIN TableA  ON TableA.num == TableB.num;
```

Right: Yellow and Green in Venn Diagram, Above

```
/* inner (right) join */
 SELECT TableB.num  FROM TableA  LEFT JOIN TableB  ON TableA.num == TableB.num;
 SELECT count(TableB.num)  FROM TableA  LEFT JOIN TableB  ON TableA.num == TableB.num;
```

- How many spaces did you count from each query?
- What do the spaces tell you?

New Database

Exam1
Joins
New Database
Sets
AS Clauses
Strings
Ordering

(A New Database!)

New Database

Schema: Red boxes are the tables of today's database study

Exam1
Joins
New Database
Sets
AS Clauses
Strings
Ordering

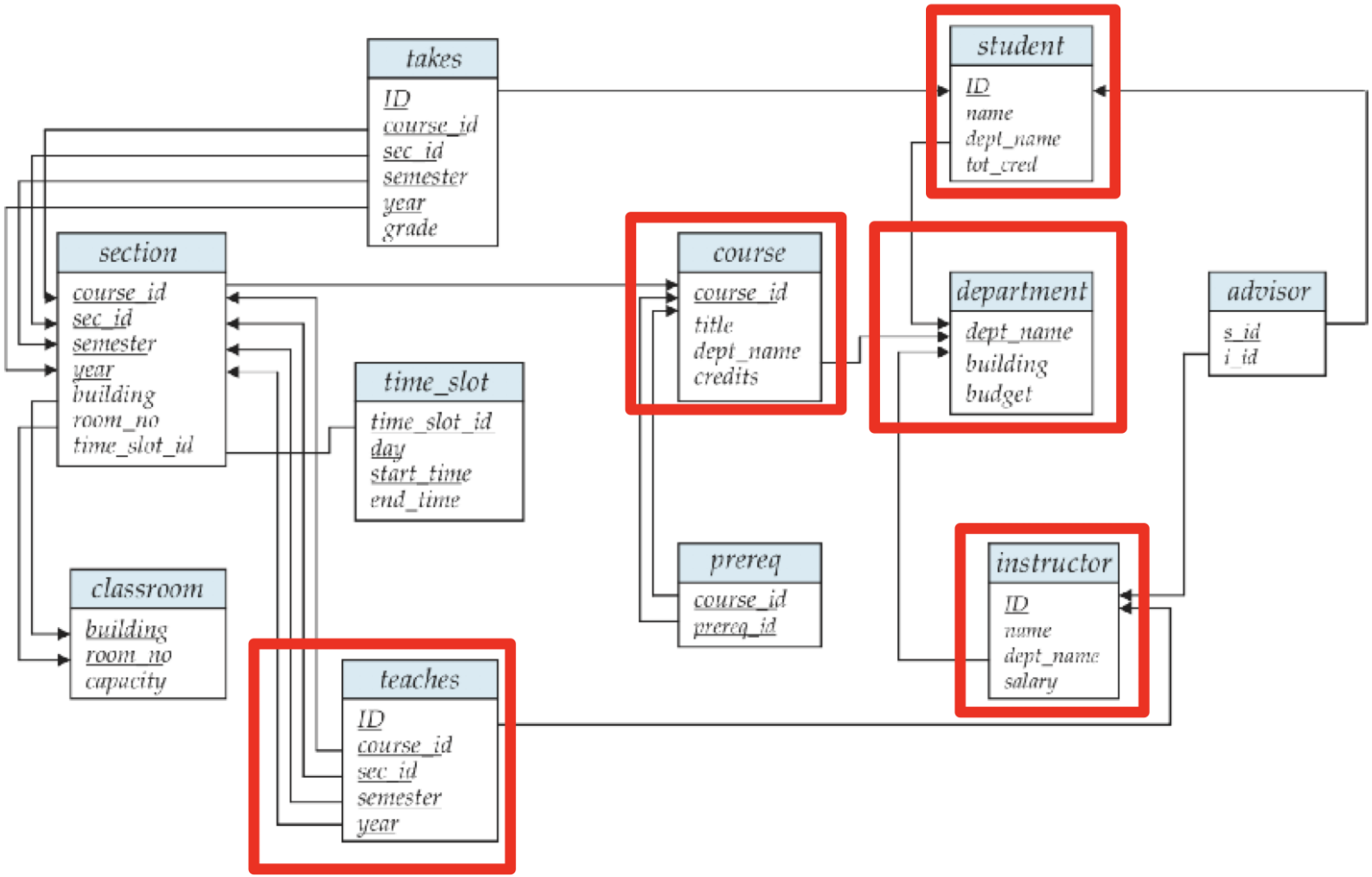


Figure 2.8 Schema diagram for the university database.

New Database

- Find the database maker file, *campusDB_build.txt*, in your sandbox directory

```
cat campusDB_build.txt | sqlite3 myCampusDB.sqlite3
```

Set Operations
OR & AND

Exam1
Joins
New Database
Sets
AS Clauses
Strings
Ordering

- **OR**: Find all `deptNames` in the **UNION** of `Instructor` and `Course`

```
select deptName from Instructor UNION select deptName from course;
```

```
select distinct(deptName) from Instructor;
```

- **AND**: Find all `deptNames` in the **INTERSECT** of `Instructor` and `Course`

```
select deptName from Instructor INTERSECT select deptName from Course;
```

```
select distinct(Instructor.deptName) from Instructor, Course where Instructor.deptName == Course.deptName;
```

Set Operations

Exam1
Joins
New Database
Sets
AS Clauses
Strings
Ordering

- select distinct(deptName) from Instructor;
- select distinct(deptName) from Course;

- The EXCEPT operator compares the result sets of two queries and returns distinct rows from the left query that are not in the output by the right query.
- Find all deptNames different to both the *Instructor* and *Course*
- Check these two queries below. Why is the output different?

- `select deptName from Instructor EXCEPT select deptName from Course;`
- `select deptName from Course EXCEPT select deptName from Instructor;`

AS clauses

Exam1
Joins
New Database
Sets
AS Clauses
Strings
Ordering

- The **AS** clause is used to rename relations; useful for reducing necessary code in queries
- Ex: *For all instructors in the university who have taught some course, find their names and the course ID of all their taught courses*
  - Select I.name, T.courseID
    FROM `Instructor` AS I , `Teaches` AS T
    WHERE I.ID= T.ID;
- On the second line:
  - the `Instructor` table is renamed to `I`
  - the `Teaches` table is renamed to `T`.

AS clauses

- Another reason to rename a relation is a case where we wish to compare tuples in the same relation.
- We then need to take the Cartesian product of a relation with itself and, without renaming, it becomes impossible to distinguish one tuple from the other.
- Suppose that we want to write the query, *find the names of all instructors whose salary is greater than at least one instructor in the Math department.*
  - SELECT DISTINCT T.name
    FROM Instructor as T ,
    Instructor AS S
    WHERE T.salary> S.salary and S.deptName == “Math”

AS clauses

Exam1
Joins
New Database
Sets
AS Clauses
Strings
Ordering

- Find all names of common teachers in `Instructor` and `Teaches` tables

Use AS to implement variables attributes to hold places

- `select distinct(Instructor.name) as newName from Instructor, teaches where Instructor.ID = teaches.ID and newName == "Thompson";`

AS clauses

Exam1
Joins
New Database
Sets
AS Clauses
Strings
Ordering

- Find the names of all Instructors whose salary is greater than at least one Instructor in the Math department.
- `select distinct(T.name) from Instructor as T, Instructor as S where T.salary > S.salary and S.deptName == "Math";`
- `select distinct T.name, T.salary from Instructor as T, Instructor as S where T.salary > S.salary and S.deptName == "Math";`
- Reference: `select * from Instructor;`

Regular Expression-ish

- Textual *wildcards* to recover information from partial knowledge.
- Finding substrings using the % and _ operators.

- `select name from Instructor where name like "%ille%";`
  - Selects *Miller* from a substring
- `select name from Instructor where name like "%son";`
  - Selects all names followed by "son" substring
- Compare to: *Select * from Instructor;*
- `select name from Instructor where name like "__ll__";`
- `select name from Instructor where name like "__ll___";`
  - Selects "Miller" or "William" from the number of spaces after the "ll";.

Regular Expression-ish

Exam1
Joins
New Database
Sets
AS Clauses
Strings
Ordering

- Find special pattern characters (i.e., "%" and "_") in strings
- SQL even allows the specification of an escape character.

- **like** 'ab\%cd%' **escape** '\' matches all strings beginning with "ab%cd".
- **like** 'ab\\cd%' **escape** '\' matches all strings beginning with "ab\cd".

Ordering Results

Exam1
Joins
New Database
Sets
AS Clauses
Strings
Ordering
Having

- SQL allows for sorting the output.
- Output is sorted alphabetically

- `select name from Instructor order by name;`
- `select name,salary from Instructor order by salary;`
  - Provides numerical values in an interval

"Intermediate" Results Using **HAVING**

Exam1
Joins
New Database
Sets
AS Clauses
Strings
Ordering
Having

- The **HAVING** clause enables you to specify conditions that filter which group results appear in the final results.
- The **HAVING** clause must follow the GROUP BY clause in a query and must also precede the ORDER BY clause if used.

Pseudo-code

```
SELECT column1, column2
FROM table1, table2
WHERE [ conditions ]
GROUP BY column1, column2
HAVING [ conditions ]
ORDER BY column1, column2
```

Group By

Exam1
Joins
New Database
Sets
AS Clauses
Strings
Ordering
Having

- Give the number of names, and names of all members of departments who make less than 100000.

- `select count(name), deptName from Instructor GROUP BY deptName HAVING salary < 100000;`

- Give the deptNames and the average salaries for departments that begin with the letter 'C'.

- `select deptName, avg(salary) from Instructor group by deptName HAVING deptName LIKE "C%";`

Group By

Exam1
Joins
New Database
Sets
AS Clauses
Strings
Ordering
Having

- Give the department names and salaries from the Instructor group for whose members make between 97K and 100K.

- `select deptName, salary from Instructor group by deptName HAVING salary < 100000 and salary > 97000;`

- Compare to: Give me deptName and salary information where the salary is between 97K and 100K.

- `select deptName, salary from Instructor where salary < 100000 and salary > 97000 group by deptName;`

Use avg to Query

Exam1
Joins
New Database
Sets
AS Clauses
Strings
Ordering
Having

- `select deptName, avg(salary) from Instructor group by deptName;`
  - Report average salaries for departments
- `select deptName, avgSalary FROM (select deptName, avg(salary) as avgSalary from Instructor group by deptName) where avgSalary > 97000;`
  - Report average salaries larger than $97k. This query is similar to one using the **HAVING** clause. Here we use the **FROM** clause.

Ordering Result Using BETWEEN

Exam1
Joins
New Database
Sets
AS Clauses
Strings
Ordering
Having

- SQL allows for sorting the output by criteria
- Output is sorted for values in an interval

- `select name, salary from Instructor where salary <= 100000 and salary >= 90000;`
- `select name, salary from Instructor where salary between 70000 and 100000;`
  - Query values in their intervals.