Introduction to Database Systems: CS305 Advanced queries, joins and aggregates

Oliver Bonham-Carter
Hang Zhao

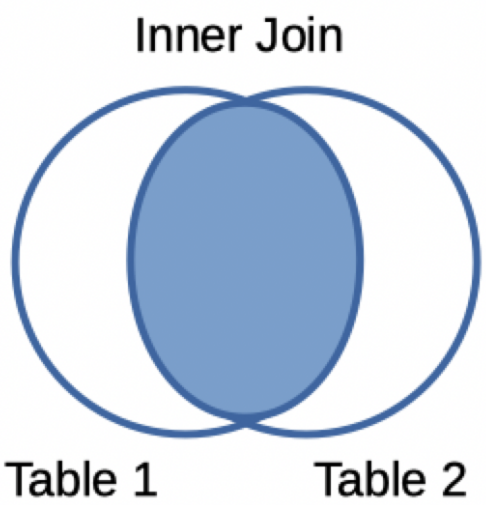
17-19 October 2023

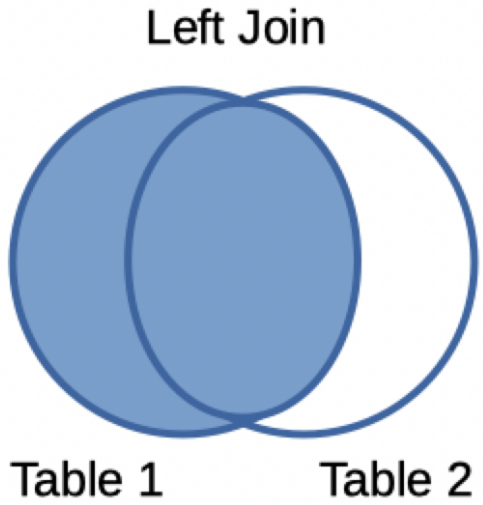
Joins: Bringing Data Together

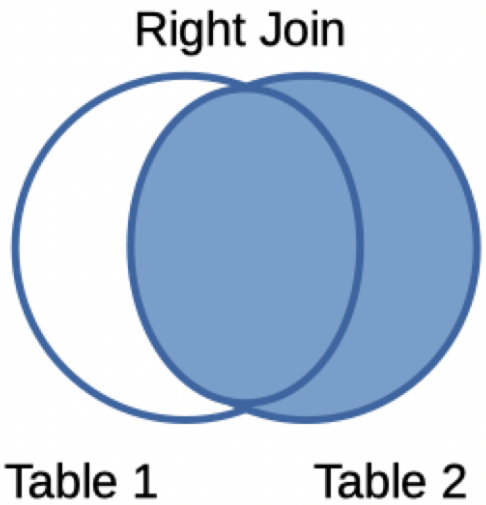
- The SQLite3 `join`-clause is used to combine records from two or more tables in a database.
- A **JOIN** is a means for combining fields from two tables by using values common to each.

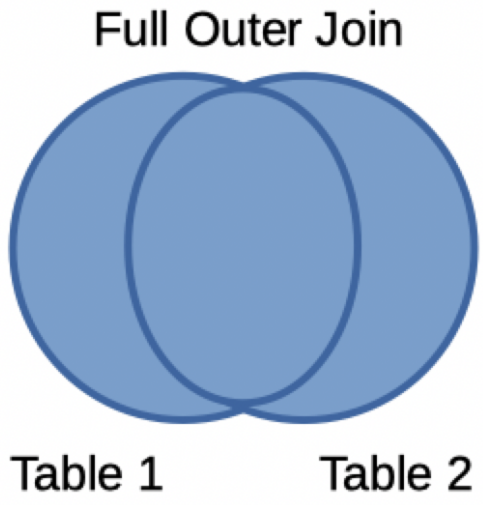
Joins: Visual Definitions

As Venn Diagrams

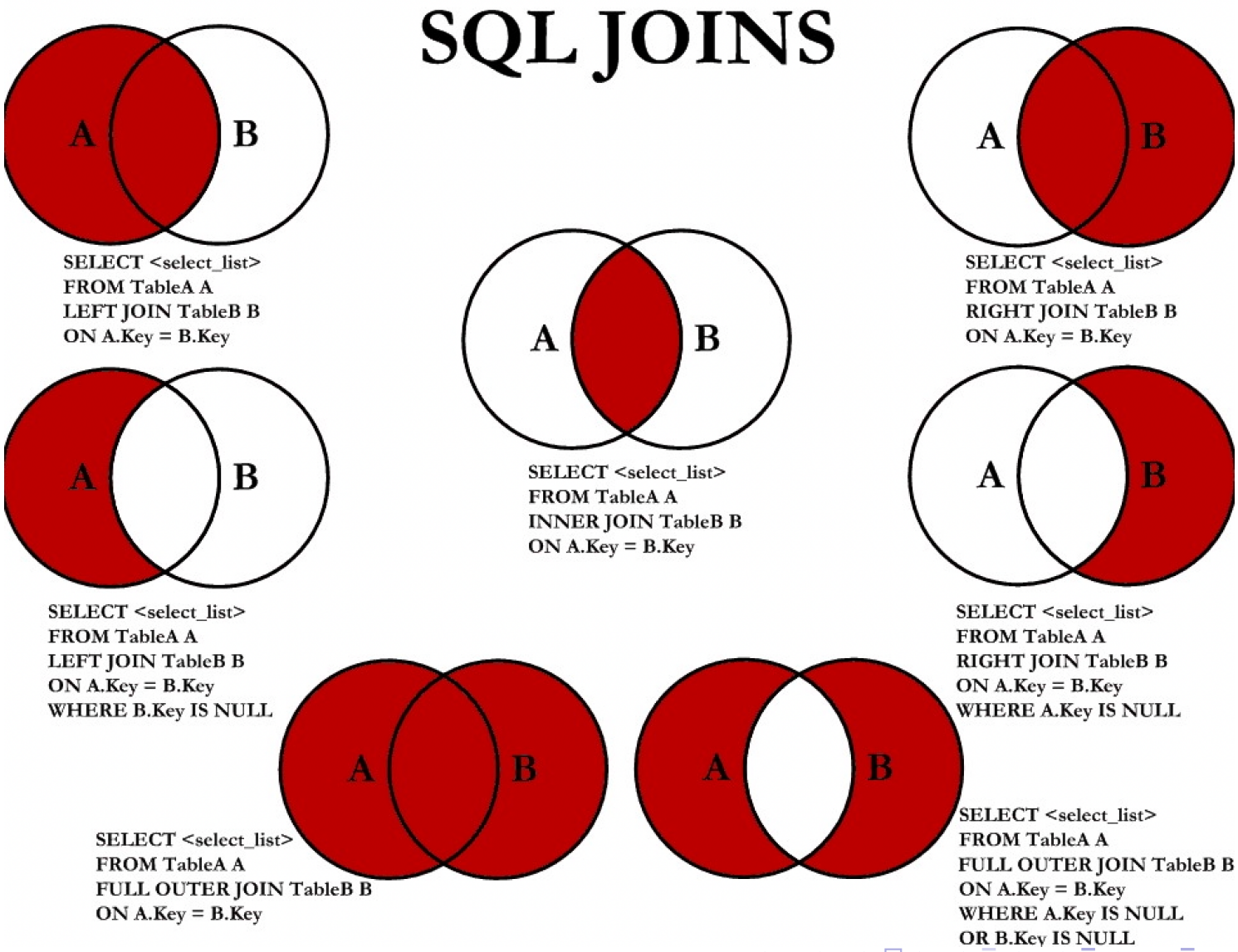








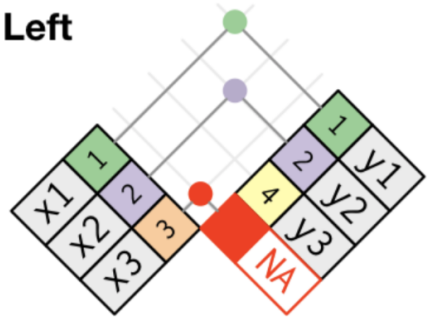
SQL Code and Venn Diagrams



Joins: Visual Definitions

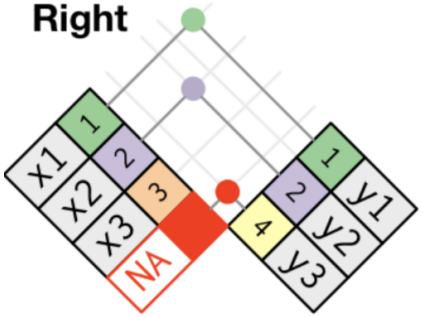
Combining Tables

Left



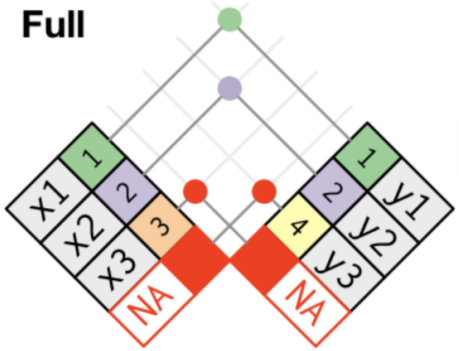
key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA

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

An Explanation of Terms

- SQL joins
  - An **inner join** will return records that have matching values in *both* tables.
  - A left **outer join** will return all records from the *left* table and the matched records from the *right* table.
  - A right **outer join** will return all records from the *right* table and the matched records from the *left* table.
  - A **full outer join** will return all records when there is a match from *either* table.

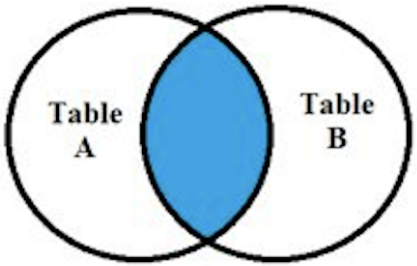
Inner Joins

Student ID	Name
1001	A
1002	B
1003	C
1004	D

Table A

Student ID	Department
1004	Mathematics
1005	Mathematics
1006	History
1007	Physics
1008	Computer Science

Table B



Student ID	Name	Department
1004	D	Mathematics

Inner joins

File: /sandbox/fruitJoin.txt

Create two tables

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

DROP TABLE IF EXISTS TableB;
CREATE TABLE TableB (
fruit VARCHAR,
colour VARCHAR);
```

Inner joins

File: /sandbox/fruitJoin.txt

Populate the tables

```
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");
```

Inner joins

File: /sandbox/fruit – innerJoin.txt

Use INNER JOIN to query

```
.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
Output

Output

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

Left Join

Matches entries from LEFT table to the other table

Setup Tables

```
/*Drop the table if it already exists*/
DROP TABLE IF EXISTS Employees;

/*Create the Employees table*/
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,   FirstName VARCHAR,
    LastName VARCHAR,   DepartmentID INT
);

DROP TABLE IF EXISTS Departments;

CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY,   DepartmentName VARCHAR
);
```

Left Join
example

Populate

```
/*Insert sample data into the Employees table*/
INSERT INTO Employees (
  EmployeeID, FirstName,
  LastName, DepartmentID)
VALUES
    (1, 'John', 'Doe', 1),
    (2, 'Jane', 'Smith', 2),
    (3, 'Bob', 'Johnson', 1),
    (4, 'Alice', 'Williams', NULL);

/*Insert sample data into the Departments table*/
INSERT INTO Departments (DepartmentID, DepartmentName)
VALUES
    (1, 'HR'),
    (2, 'IT'),
    (3, 'Finance');
```

Left Join
example

Query

```
/*Perform a LEFT JOIN to retrieve a list
  of all employees and their departments*/
SELECT
   e.EmployeeID, e.FirstName, e.LastName, d.DepartmentName
FROM
   Employees e
LEFT JOIN
   Departments d
ON
   e.DepartmentID = d.DepartmentID;
```

```
EmployeeID | FirstName | LastName | DepartmentName
-------------------------------------------------
1          | John      | Doe      | HR
2          | Jane      | Smith    | IT
3          | Bob       | Johnson  | HR
4          | Alice     | Williams | NULL
```

Right Join

Matches entries from RIGHT table to the other table

Setup Tables

```
/*Perform a LEFT JOIN to retrieve a list
  of all employees and their departments*/
SELECT
   e.EmployeeID, e.FirstName, e.LastName, d.DepartmentName
FROM
   Employees e
RIGHT JOIN
   Departments d
ON
   e.DepartmentID = d.DepartmentID;
```

```
EmployeeID | FirstName | LastName | DepartmentName
-------------------------------------------------
1          |     John  |     Doe  |     HR
2          |     Jane  |     Smith|     IT
3          |     Bob   |     Johnson|   HR
NULL       |     NULL  |     NULL |     Finance
```

Cross joins

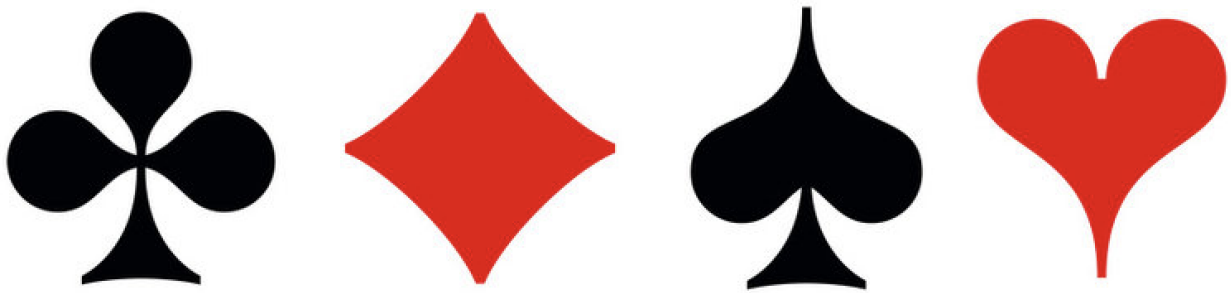
Cross product derivied from both tables

```
 DROP TABLE IF EXISTS ranks;
 CREATE TABLE ranks (
    rank TEXT NOT NULL
);
DROP TABLE IF EXISTS suits;
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;
```

Cross joins: All Card Pairs

Cross join: output

```
2|Clubs
...
J|Clubs
Q|Clubs
K|Clubs
A|Clubs
...
A|Spades
```



New Database

(A New Database!)

New Database

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

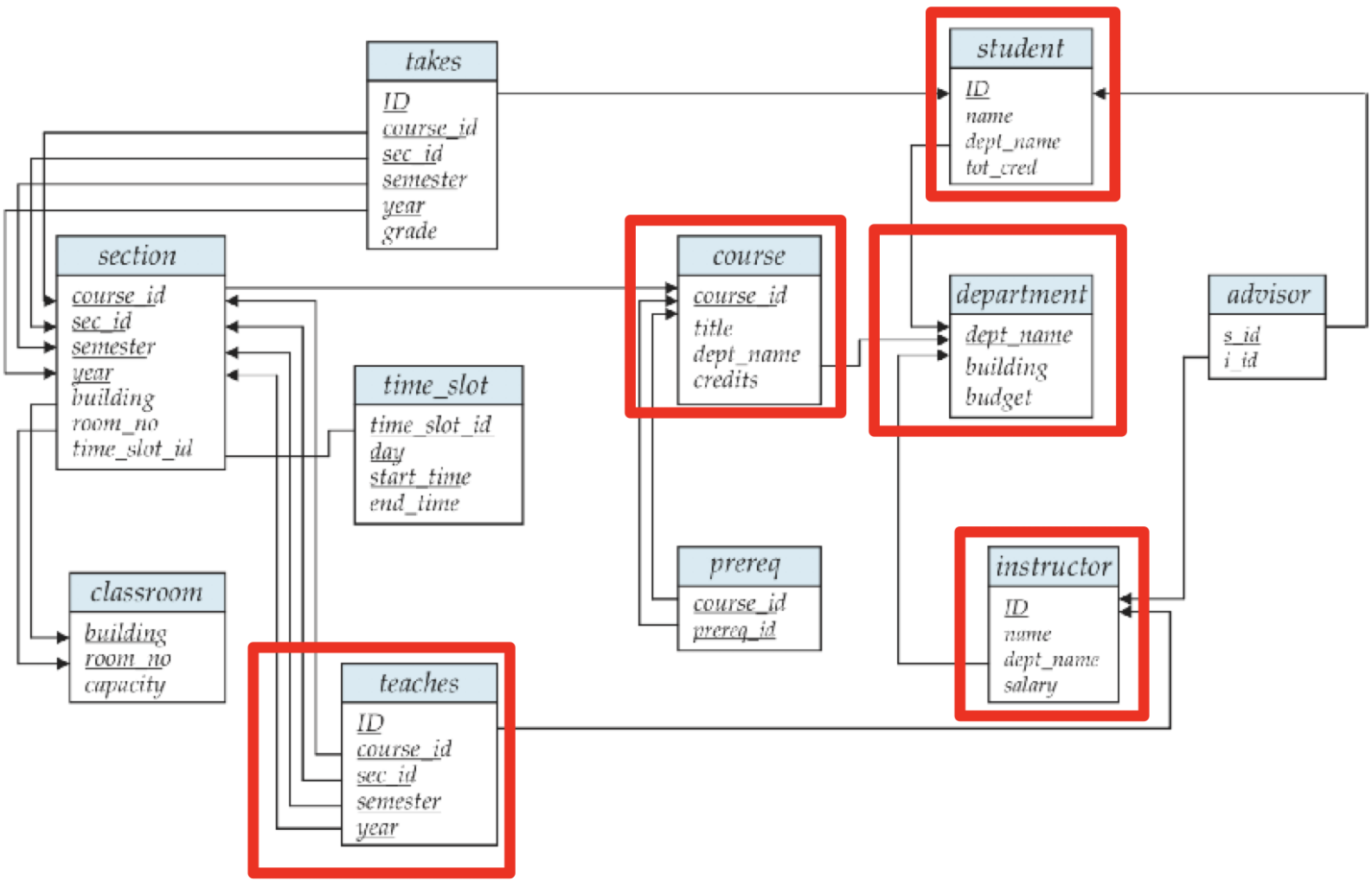


Figure 2.8 Schema diagram for the university database.

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

- **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

- 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

- 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

- 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

- 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

- 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

- 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**

- 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

- 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

- 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

- `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

- 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.

Consider this!

- Can you create a `JOIN` between two tables?
- Can you use `EXCEPT` and `AS` to add fine tune your queries?
- Can you write SQL code to be more precise numerically using `BETWEEN`, `AVG`, and greater-than and less-than?