

1 Introduction

SQL is an example of a declarative programming language. Statements do not describe computations directly, but instead describe the desired result of some computation. It is the role of the query interpreter of the database system to plan and perform a computational process to produce such a result.

In SQL, data is organized into *tables*. A table has a fixed number of named **columns**. A **row** of the table represents a single data record and has one **value** for each column. For example, we have a table named `records` that stores information about the employees at a small company¹. Each of the eight rows represents an employee.

Name	Division	records Title	Salary	Supervisor
Ben Bitdiddle	Computer	Wizard	60000	Oliver Warbucks
Alyssa P Hacker	Computer	Programmer	40000	Ben Bitdiddle
Cy D Fect	Computer	Programmer	35000	Ben Bitdiddle
Lem E Tweakit	Computer	Technician	25000	Ben Bitdiddle
Louis Reasoner	Computer	Programmer Trainee	30000	Alyssa P Hacker
Oliver Warbucks	Administration	Big Wheel	150000	Oliver Warbucks
Eben Scrooge	Accounting	Chief Accountant	75000	Oliver Warbucks
Robert Cratchet	Accounting	Scrivener	18000	Eben Scrooge

2 Creating Tables

We can use a `SELECT` statement to create tables. The following statement creates a table with a single row, with columns named “first” and “last”:

```
sqlite> SELECT "Ben" AS first, "Bitdiddle" AS last;  
BenBitdiddle
```

Given two tables with the same number of columns, we can combine their rows into a larger table with `UNION`:

```
sqlite> SELECT "Ben" AS first, "Bitdiddle" AS last UNION  
...> SELECT "Louis", "Reasoner";  
BenBitdiddleLouisReasoner
```

¹Example adapted from Structure and Interpretation of Computer Programs

2 SQL

To save a table, use `CREATE TABLE` and a name. Here we're going to create the table of employees from the previous section and assign it to the name `records`:

```
sqlite> CREATE TABLE records AS
...> SELECT "Ben Bitdiddle" AS name, "Computer" AS division,
...>    "Wizard" AS title, 60000 AS salary,
...>    "Oliver Warbucks" AS supervisor UNION
...> SELECT "Alyssa P Hacker", "Computer",
...>    "Programmer", 40000, "Ben Bitdiddle" UNION ... ;
```

We can `SELECT` specific values from an existing table using a `FROM` clause. This query creates a table with two columns, with a row for each row in the `records` table:

```
sqlite> SELECT name, division FROM records;
Alyssa P HackerComputerBen BitdiddleComputer
Cy D FectComputerEben ScroogeAccounting
Lem E TweakitComputerLouis ReasonerComputer
Oliver WarbucksAdministrationRobert CratchetAccounting
```

The special syntax `SELECT *` will select all columns from a table. It's an easy way to print the contents of a table.

```
sqlite> SELECT * FROM records;
Alyssa P HackerComputerProgrammer40000Ben Bitdiddle
Ben BitdiddleComputerWizard60000Oliver Warbucks
Cy D FectComputerProgrammer35000Ben Bitdiddle
Eben ScroogeAccountingChief Accountant75000Oliver Warbucks
Lem E TweakitComputerTechnician25000Ben Bitdiddle
Louis ReasonerComputerProgrammer Trainee30000Alyssa P Hacker
Oliver WarbucksAdministrationBig Wheel15000Oliver Warbucks
Robert CratchetAccountingScrivener18000Eben Scrooge
```

We can choose which columns to show in the first part of the `SELECT`, we can filter out rows using a `WHERE` clause, and sort the resulting rows with an `ORDER BY` clause. In general the syntax is:

```
SELECT [columns] FROM [tables]
WHERE [condition] ORDER BY [criteria];
```

For instance, the following statement lists all information about employees with the "Programmer" title.

```
sqlite> SELECT * FROM records WHERE title = "Programmer";
Alyssa P HackerComputerProgrammer40000Ben Bitdiddle
Cy D FectComputerProgrammer35000Ben Bitdiddle
```

The following statement lists the names and salaries of each employee under the accounting division, sorted in **descending** order by their salaries.

```
sqlite> SELECT name, salary FROM records
...> WHERE division = "Accounting" ORDER BY -salary;
Eben Scrooge75000Robert Cratchet18000
```

Note that all valid SQL statements must be terminated by a semicolon (;). Additionally, you can split up your statement over many lines and add as much whitespace as you want, much like Scheme. But keep in mind that having consistent indentation and line breaking does make your code a lot more readable to others (and your future self)!

Questions

Our tables:

records: **Name Division Title Salary Supervisor**

- 2.1 Write a query that outputs the names of employees that Oliver Warbucks directly supervises.
- 2.2 Write a query that outputs all information about employees that supervise themselves.
- 2.3 Write a query that outputs the names of all employees with salary greater than 50,000 in alphabetical order.

3 Joins

Suppose we have another table `meetings` which records the divisional meetings.

meetings		
Division	Day	Time
Accounting	Monday	9am
Computer	Wednesday	4pm
Administration	Monday	11am
Administration	Wednesday	4pm

Data are combined by joining multiple tables together into one, a fundamental operation in database systems. There are many methods of joining, all closely related, but we will focus on just one method (the inner join) in this class.

When tables are joined, the resulting table contains a new row for each combination of rows in the input tables. If two tables are joined and the left table has m rows and the right table has n rows, then the joined table will have mn rows. Joins are expressed in SQL by separating table names by commas in the FROM clause of a SELECT statement.

```
sqlite> SELECT name, day FROM records, meetings;
Ben BitdiddleMondayBen BitdiddleWednesday
...
Alyssa P HackerMonday...
```


- 3.3 Write a query that outputs the names of all pairs of employees that have a meeting at the same time. Make sure that if A|B appears in your output, B|A does not appear as well (A|A and B|B should additionally not appear).
- 3.4 (Extra question) Will the statement above filter out all redundant output in all cases? Why or why not?

4 Aggregation

So far, we have joined and manipulated individual rows using `SELECT` statements. But we can also perform aggregation operations over multiple rows with the same `SELECT` statements.

We can use the `MAX`, `MIN`, `COUNT`, and `SUM` functions to retrieve more information from our initial tables.

If we wanted to find the name and salary of the employee who makes the most money, we might say

```
sqlite> SELECT name, MAX(salary) FROM records;
Oliver Warbucks150000
```

Using the special `COUNT(*)` syntax, we can count the number of rows in our table to see the number of employees at the company.

```
sqlite> SELECT COUNT(*) from RECORDS;
9
```

These commands can be performed on specific sets of rows in our table by using the `GROUP BY [column name]` clause. This clause takes all of the rows that have the same value in `column name` and groups them together.

We can find the minimum salary earned in each division of the company.

```
sqlite> SELECT division, MIN(salary) FROM records GROUP BY division;
Computer25000Administration25000
Accounting18000
```

These groupings can be additionally filtered by the `HAVING` clause. In contrast to the `WHERE` clause, which filters out rows, the `HAVING` clause filters out entire groups.

5 Modifying Tables

Note: you may not yet be familiar with the syntax in this section. We will cover it more formally by the end of the class, but for now it can serve as a demo of some additional things you can do in SQL.

Tables don't need to begin fully formed, it's possible to update them after creation! We'll also introduce this alternative syntax for creating a table, which creates an empty table with the given columns:

```
CREATE TABLE [table]([column1], [column2] DEFAULT [val], ...);
```

The **optional** **DEFAULT** keyword denotes default values for a given column if they're not specified. This will be relevant when we insert new elements into our table. To add a new table entries, use the **INSERT INTO** statement:

```
INSERT INTO [table] ([column1], [column2], ...)
VALUES ([value1], [value2], ...), ([value1], [value2], ...);
```

A couple of notes:

- If a value is specified for each column of the table, you don't need to specify column names. This is because each value matches up with a column, so there's no ambiguity.
- For columns where a value is not specified, the default value will be used if available. If not a default value was not provided, that column in the new row will be left empty!

Here's an example of insertion into an empty table:

```
sqlite> CREATE TABLE dogs(name, age, phrase DEFAULT "woof");
sqlite> INSERT INTO dogs(name, age) VALUES ("Fido", 1), ("Sparky", 2);
sqlite> INSERT INTO dogs VALUES ("Lassie", 2, "I'll save you!"), ("Floofy", 3);
Error: all VALUES must have the same number of terms
sqlite> INSERT INTO dogs VALUES ("Lassie", 2, "I'll save you!"), ("Floofy", 3, "Much doge");
sqlite> SELECT * FROM dogs;
Fido1woof
Sparky2woof
Lassie2I'll save you!
Floofy3Much doge
```

The **INSERT INTO** statement can also insert a table returned by a **SELECT** statement, in which case the syntax is:

```
INSERT INTO [table] ([column1], [column2], ...)
SELECT ...;
```

We can update certain existing entries in a table using **UPDATE**:

```
UPDATE [table] SET [column1] = [value1], [column2] = [value2], ... WHERE [condition];
```

All rows matching the condition will have their columns updated. If no condition is specified, **all** rows will be updated! We can also remove certain entries in a table using **DELETE**:

```
DELETE FROM [table] WHERE [condition];
```

Just like with UPDATE, if no condition is specified, **all** rows will be deleted! Here's an example using all of the above:

```
sqlite> UPDATE dogs SET age=age+1; -- If condition isn't specified, every row is updated
```

```
sqlite> SELECT * FROM dogs;
```

```
Fido2woof
```

```
Sparky3woof
```

```
Lassie3I'll save you!
```

```
Floofy4Much doge
```

```
sqlite> UPDATE dogs SET phrase = "bark" WHERE age=2;
```

```
sqlite> SELECT * FROM dogs WHERE age=2;
```

```
Fido2bark
```

```
sqlite> DELETE FROM dogs WHERE age=3;
```

```
sqlite> SELECT * FROM dogs;
```

```
Fido2bark
```

```
Floofy4Much doge
```

Finally, we can delete an entire table using the DROP TABLE [table] statement. In this example, the .schema statement shows us a list of the current tables, along with their column names.

```
sqlite> .schema
```

```
CREATE TABLE dogs(name, age, phrase DEFAULT "woof");
```

```
sqlite> DROP TABLE dogs;
```

```
sqlite> .schema
```

```
sqlite> -- Nothing displayed above
```

Questions

Our tables:

```
dogs:  Name  Age  Phrase, DEFAULT="woof"
```

- 5.1 What would SQL display? **Keep track of the contents of the table after every statement below.** Write Error if you think a statement would cause an error.

```
sqlite> SELECT * FROM dogs;
```

```
Fido1woof
```

```
Sparky2woof
```

```
Lassie2I'll save you!
```

```
Floofy3Much doge
```

```
sqlite> INSERT INTO dogs(age, name) VALUES ("Rover", 3);
```

```
sqlite> SELECT * FROM dogs;
```

```
sqlite> UPDATE dogs SET name=age, age=name WHERE name=3;
sqlite> SELECT * FROM dogs;
```

```
sqlite> UPDATE dogs SET phrase="Hi there!" WHERE name LIKE "F%";
sqlite> SELECT * FROM dogs;
```

```
sqlite> DELETE FROM dogs WHERE age < 3;
sqlite> SELECT * FROM dogs;
```

```
sqlite> INSERT INTO dogs VALUES ("Spot", 2), ("Buster", 4);
```

```
sqlite> INSERT INTO dogs(name, phrase) VALUES ("Spot", "bark"), ("Buster", "barkbark");
sqlite> SELECT * FROM dogs;
```

```
sqlite> INSERT INTO dogs(name, age) SELECT name, phrase from dogs where age = 3;
sqlite> DELETE FROM dogs WHERE phrase != "woof";
sqlite> SELECT * FROM dogs;
```

6 Extra Questions

Use the following table called `courses` for the questions below:

courses		
Professor	Course	Semester
Dan Garcia	CS 61C	Sp19
John DeNero	CS 61A	Fa18
Dan Garcia	CS 10	Fa18
Josh Hug	CS 61B	Sp18
John DeNero	CS 61A	Sp18
John DeNero	CS 61A	Fa17
Paul Hilfinger	CS 61A	Fa17
Paul Hilfinger	CS 61A	Sp17
John DeNero	Data 8	Sp17
Josh Hug	CS 61B	Sp17
Satish Rao	CS 70	Sp17
Nicholas Weaver	CS 61C	Sp17
Gerald Friedland	CS 61C	Sp17
⋮	⋮	⋮

- 6.1 Create a table called `num_taught` that contains three columns: `professor`, the course they taught, and the number of times they taught each course.
- 6.2 Write a query that outputs two professors and a course if they have taught that course the same number of times. You may use the `num_taught` table you created in the previous question.
- 6.3 Write a query that outputs two professors if they co-taught (taught the same course at the same time) the same course more than once.