Some Notes on Databases

Preface

These are notes I wrote up for my database class. They are fairly brief and informal. These cover basic SQL (using MySQL), some concepts on database design, and a little about NoSQL, including Redis and MongoDB.

If you see anything wrong (including typos), please send me a note at `heinold@msmary.edu`.

Last updated: December 29, 2025.

1 Getting Started

There are several widely used SQL databases, including MySQL, PostgreSQL, Microsoft SQL Server, Oracle Database, and SQLite. The syntax of SQL each uses is largely the same, but they all differ from each other in various places. In these notes, we will be using MySQL. It is free, full-featured, widely used, and easy to get started with.

Download MySQL from here: `https://dev.mysql.com/downloads/mysql/`. You'll have to download the MySQL Workbench separately here: `https://dev.mysql.com/downloads/workbench/`. When installing it, follow all the defaults. It will ask you to create a password for the root account. Create whatever password you like, but be sure to remember what it is. If the installer asks you to include the `world` database, do so, since many of the examples in these notes will use it.

We'll be using the MySQL Workbench. A nice, quick intro to using it is here: `https://www.youtube.com/watch?v=2mbHyB2VLYY`.[1]

To create a new database in MySQL Workbench, in the area on the left, choose "Schemas" and then right-click and choose "Create Schema". Give it a name and follow the default instructions. Once it's done, double click on its name to make sure it's highlighted in bold. Make sure there's a blank query tab open in which to run commands. If there's not, under the File menu there is an option to open one. You could also use control+T. Try typing `SELECT NOW()` and clicking the lightning bolt icon to run the command. If everything is working, you should see the current date and time pop up.[2]

To run commands in MySQL Workbench, enter them into the query area. You can run all of the commands in that area at once by clicking on the lightning bolt run button. You can run a portion of the commands by highlighting them and clicking that button. You can run a single command by using the lightning bolt icon that has an I on it. This will run the command for the line the cursor is on. You can also do this by typing control+enter on the line with the command.

You can save your SQL query tabs as .sql files using the file menu.

A little about databases A database is a place to store data. The databases we use SQL with are called relational databases. The database is broken up into *tables*, and the tables are related to each other in various ways. Each table has various properties called *columns*, and it has various entries called *rows*.

For example, suppose we want a database for storing stuff to do with a school. We might have a table for students, a table for courses, a table for faculty, and a table for classrooms, among many other possible tables. The student table might have a column for the student's ID number, a column for the student's name, a column for their class (freshman, sophomore, etc.), and columns for various other things. The rows in the student table correspond to individual students.

[1] There is also a command-line interface. You might be able to find it in your OS by searching for MySQL. Otherwise, find where MySQL was installed, then go into the MySQL Server directory and then the bin directory and run `mysql -u root -p`. Replace `root` with the name of the user using the database if you're not accessing it as root.

[2] If not, make sure you are connected to the database. You might need to rerun the installation configuration. On Windows, you could try running Windows's Services program and scroll down to make sure the MySQL service has been started.

2 Creating tables and working with tables

Here is some syntax for creating a simple table to represent a student.

```
CREATE TABLE students (
    student_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    birthday DATE
);
```

The first column, `student_id` is something called a primary key. Most tables should have a primary key. A simple way to create a primary key is to use the syntax above, which guarantees that each row of the database will have a unique primary key. The `AUTO_INCREMENT` gives the first row a key of 1, the second row a key of 2, etc. This is important to help us tell rows apart in case all their other columns end up equal. The second and third columns are for the student's birthday and name. When declaring columns, we have to say what data type they hold. More on data types in a little while. The various columns and their data types form what is called the table's *schema*.

When you run this command in MySQL workbench, the table will show up in the schemas area. If you don't see it, click on the refresh icon in the upper right corner of the schemas area.

Inserting info into a table Below is a line to insert a row into the database.

```
INSERT INTO students (name, birthday) VALUES ('Brian', '1999-01-30');
```

Note that we don't have to specify the ID. MySQL automatically generates it for us. Here is how we could insert multiple rows at once.

```
INSERT INTO students (name, birthday) VALUES
('Sally', '1992-02-29'),
('Jim', '1987-12-25');
```

We don't have to enter values for every field. For things that are not given, MySQL will insert a special value called NULL, which indicates that there is no value for that column. For instance, we can do this:

```
INSERT INTO students (name) VALUES ('Maria');
```

Finally, to show all the information in the table, use the following:

```
SELECT * FROM students;
```

The MySQL workbench will display the results. It will look something like this:

student_id	name	birthday
1	Brian	1999-01-30
2	Sally	1992-02-29
3	Jim	1987-12-25
4	Maria	NULL

Note also that you don't need to remember the order you created things in the database. For instance, the following two lines have the same effect.

```
INSERT INTO students (name, birthday) VALUES ('Tom', '2000-11-30');
INSERT INTO students (birthday, name) VALUES ('2000-11-30', 'Tom');
```

Changing info in a table Use the `UPDATE` statement to change a row. Here are two examples that change info for the first student.

```
UPDATE students SET birthday='1999-02-01' WHERE student_id=1;
UPDATE students SET name='Bryan', birthday='1999-02-02' WHERE student_id=1;
```

As you enter all commands into the MySQL Workbench and run them, notice at the bottom it will tell you if the command was successful or if there was an error.[1]

[1] You can use various things besides ID in the WHERE clause. For instance, if you want to update the birthday of the student whose name is Brian, instead of using the ID in the WHERE clause, you could use `WHERE name='Brian'`. However, that's usually not a good idea since if there are multiple students with that name, then all of their birthdays will be updated. In fact, your installation of SQL might have

Changing table structure If you want to change part of a table's schema, use the `ALTER TABLE` command. Here is an example.

```
ALTER TABLE students ADD phone_number VARCHAR(20);
```

This adds a new column. It is also possible to change the table in other ways, such as renaming a column. Below, we rename the `phone_number` column and also slightly change its data type.

```
ALTER TABLE students CHANGE phone_number phone VARCHAR(30);
```

Deleting things If you want to delete a table, use the `DROP TABLE` command. To delete a row, use `DELETE FROM`. Examples are below.

```
DROP TABLE students;
DELETE FROM students WHERE student_id = 1;
```

Comments In standard SQL, `--` is used for a single line comment. MySQL also allows `#`. For multi-line comments, use `/* */`, like in C++ or Java.

Spacing Extra whitespace is ignored, so you can space things out however you like.

Case The convention is to enter all SQL commands in uppercase. It's not required, however. MySQL is case sensitive on some things, but not others. It is not case sensitive on keywords and column names. It is case sensitive for table names on Linux but not on Windows.

Data types and other things

Here are some of the most important data types:

- `VARCHAR` – Use this for text/string data. You specify a maximum size. For instance, `name VARCHAR(20)` allocates up to 20 characters for a name. It will only store as many characters as you insert, and it will cut it off if you insert something greater than the maximum size. The max length you can use is 65535.
- `INTEGER` – Use this for whole numbers. The `INTEGER` data type is 32-bit signed integer, which means it holds values from roughly −2 billion to +2 billion.
- `NUMERIC` – Use this for decimal numbers. To store a price of an item, you might use `price NUMERIC(10, 2)`. This says that it will hold up to 10 total digits, 2 of which come after the decimal point. That is, there can be up to 8 before the decimal point and up to 2 after it. In general, `NUMERIC(n, k)` says to use n digits total, with k of them coming after the decimal point.

For dates and times, here are some important types:

- `DATE` – This stores dates in a `YYYY-MM-DD` format.
- `TIME` – This stores times in a `HH:MM:SS` format.
- `DATETIME` – This is like a combo of the `DATE` and `TIME` types. Its format is `'YYYY-MM-DD HH:MM:SS'`.

There are various other data types that we will cover later.

"safe references" enabled by default, which will prevent you from doing this. If you really want to do something like this, you can disable that setting.

3 Querying the database with SELECT

The `SELECT` statement is the most important in all of SQL. It is how we get info from the database. To demonstrate some of its features, here is a table we will be working with:

```
CREATE TABLE people(
    person_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    last_name VARCHAR(100),
    first_name VARCHAR(100),
    birthday DATE,
    salary INTEGER,
    major VARCHAR(100)
);
```

If you're working along with this, insert in a few people into this table, like below:

```
INSERT INTO people (last_name, first_name, birthday, salary, major) VALUES
('Smith', 'John', '1990-05-15', 55000, 'Math'),
('Johnson', 'Emily', '1985-08-22', 60000, 'CS'),
('Brown', 'Michael', '1995-02-10', 48000, 'CS'),
('Davis', 'Sarah', '1988-11-30', 62000, 'Cyber'),
('Wilson', 'Christopher', '1992-07-03', 58000, 'Data'),
('Martinez', 'Linda', '1998-04-18', 52000, 'Data'),
('Harris', 'Daniel', '1987-09-25', 67000, 'CS'),
('Clark', 'Amanda', '1993-01-08', 54000, 'Cyber'),
('White', 'Matthew', '1989-06-14', 59000, 'Math'),
('Lee', 'Jennifer', '1997-03-22', 51000, 'CS');
```

Very simple queries Run this query to return the entire contents of the database.

```
SELECT * FROM people;
```

In SQL and much of computer science in general, `*` is a wildcard that stands for everything. If we just want certain columns, we can do the following:

```
SELECT last_name FROM people;
SELECT first_name, last_name FROM people;
```

WHERE clause Most `SELECT` statements have a WHERE clause that allows us to add conditions. Here is one example:

```
SELECT last_name, salary FROM people WHERE salary > 50000;
```

You can have multiple conditions like below:

```
SELECT last_name, salary FROM people WHERE birthday >= '1990-01-01' AND salary >= 50000;
```

Use `AND`, `OR`, and `NOT` as logical operators. To check for equality, use a single equals (== won't work in MySQL). To check if something is not equal, you can use either `!=` or `<>`. Here is an example

```
SELECT last_name FROM people WHERE salary = 54000 AND last_name != 'Smith';
```

Ranges and sets If you want to select things from a range, you can either do it with logical operations or using the `BETWEEN` statement, like below:

```
SELECT last_name, salary FROM people WHERE salary >= 50000 AND salary <= 60000;
SELECT last_name, salary FROM people WHERE salary BETWEEN 50000 and 60000;
```

Note that `BETWEEN` is inclusive. For instance, the query above includes 50000 and 60000. You can also check to see if something is or isn't in a set of values by using IN or NOT IN, like in the examples below.

```
SELECT last_name, salary FROM people WHERE first_name IN ('John', 'Amanda');
SELECT last_name, salary FROM people WHERE first_name NOT IN ('John', 'Amanda');
```

ORDER BY clause There is an optional `ORDER BY` clause that allows us to sort things.

```
SELECT last_name, salary FROM people WHERE salary > 50000 ORDER BY salary;
SELECT last_name, salary FROM people WHERE salary > 50000 ORDER BY salary DESC;
```

By default it sorts things from smallest to largest. To switch the order, add `DESC`, like in the second query above. The opposite of `DESC` is `ASC` (descending vs. ascending); it is the default, so you usually don't have to specify it.

You can use commas to order by multiple things. For instance, the query below will sort things first by salary, and for those people with the same salaries, they will be sorted by birthday from oldest to youngest.

```
SELECT last_name, salary FROM people ORDER BY salary, birthday DESC;
```

Operations and aliases We can perform various operations on the results. For instance, if we want to see what salaries would look like after a 20% raise, we could do the following:

```
SELECT last_name, salary*1.2 FROM people;
```

When MySQL shows you the results, it will name the second thing as `salary*1.2`. If you want to rename it to something nicer, do the following:

```
SELECT last_name, salary*1.2 AS `Raised Salary` FROM people;
```

We used backticks for the name `Raised Salary` because it has a space. Backticks are only needed if the alias has spaces, special characters, or if it is the same as an SQL keyword.

You can use `AS` to rename any columns in the output. For instance, you could do the following:

```
SELECT last_name AS Name FROM people;
```

We can actually leave off the word `AS`, but it's usually more readable to leave it in. Here is another example using a few features. It uses SQL's built-in `CONCAT` function to concatenate the first and last names with a space. It also sorts by name, sorting by last name and then by first name (if the last names are equal).

```
SELECT CONCAT(first_name, ' ', last_name) AS Name, salary*1.2 as `Raised Salary`
FROM people ORDER BY last_name, first_name;
```

DISTINCT Suppose we have 10 people in the database and some of them have the same majors as other. The following query shows the majors for all 10 people, and there will be duplicates.

```
SELECT major FROM people;
```

If we want to remove the duplicates, use the `DISTINCT` keyword, like below.

```
SELECT DISTINCT major FROM people;
```

LIMIT and OFFSET MySQL allows you to limit the number of results returned.[1] Here is an example that returns the three highest salaries and who has them.

```
SELECT last_name, salary FROM people ORDER BY salary DESC LIMIT 3;
```

The `OFFSET` keyword allows you to skip some of the results. For instance, the query below skips the first 4 results and starts with the fifth.

```
SELECT last_name, salary FROM people ORDER BY salary DESC OFFSET 5;
```

Functions There are a huge number of functions built into SQL. Some of the most useful ones are `COUNT`, `MAX`, `MIN`, `AVG`, and `SUM`. Here are a few examples:

```
SELECT MAX(salary) FROM people;
SELECT AVG(salary) FROM people;
```

The first line finds the largest salary and the second finds the average of the salaries. Here are a few examples of `COUNT`:

```
SELECT COUNT(*) FROM people;
SELECT COUNT(salary) FROM people;
SELECT COUNT(*) FROM people WHERE salary > 50000;
SELECT COUNT(DISTINCT major) FROM people;
```

[1] The LIMIT and OFFSET commands given here are not part of the SQL standard. Some SQL vendors use these, while others use different statements to accomplish the same things.

The first example returns how many rows are in the database. The second returns how many rows have an entry in the `salary` column. NULLs are not counted. The third line counts how many people have a salary over 50000. The last line returns how many different majors there are represented in the table. If we just did `COUNT(major)` without the DISTINCT keyword, the number returned would be the total number of rows in the table. We need DISTINCT to make it ignore repeats.

These functions are used all the time. There are also many functions for working with strings and dates, and many other types of functions as well. We'll cover some of them later.

NULL If a row doesn't have an entry for a specific column, the value stored there is a special value called NULL. If we want to check if a column is null, we can't use `=`. We need to use the `IS` keyword, like below.

```
SELECT last_name, salary FROM people WHERE salary IS NULL;
SELECT last_name, salary FROM people WHERE salary IS NOT NULL;
```

LIKE The `LIKE` keyword is used to find text matching certain patterns. For instance, the line below finds people whose names start with the letter "J":

```
SELECT first_name FROM people WHERE first_name LIKE 'J%';
```

The `J%` pattern says to look for a J followed by any number of other characters. The two special characters in `LIKE` are `%`, which stands for any number of characters, and _, which stands for a single character. Here are a few example patterns and what they find.

```
name LIKE '%x'   -- all names ending in x
name LIKE 'J%n'  -- all names starting with J and ending in n
name LIKE 'J__n' -- all names starting with J, followed by any two characters and then n
```

MySQL has a similar command `RLIKE` that allows more sophisticated patterns using regular expressions.[1] We won't cover regular expressions are here, but if you are familiar with them from other programming languages, they also work in SQL. For example, the example below looks for people with user names that start with A, B, or C, followed by at least one lowercase letter, followed by at least one digit.

```
SELECT username FROM people WHERE username RLIKE '[ABC][a-z]+\d+';
```

4 Grouping things

Quite often in SQL, we want to group things by a specific category. SQL's `GROUP BY` statement is used for this. For example, suppose we have a table called `people` that has information about their salary and college major. Maybe we want to list the average salaries by major. Here is what that would look like:

```
SELECT major, AVG(salary) AS Average FROM people GROUP BY major;
```

For this, SQL will find all the people with that major and average their salaries. The results might look like this:

major	Average
Business	61400
CS	94220
Math	85402

As another example, if we just wanted to know how many people there are with each major, we could do the following:

```
SELECT major, COUNT(*) AS Number FROM people GROUP BY major;
```

Note that the alias is not required. It just makes the column have a nicer name than `COUNT(*)`. The functions `SUM`, `MAX`, and `MIN` are also commonly used with `GROUP BY`.

Suppose instead of doing a `GROUP BY` on the whole table, we just want a portion of it, say just those who graduated before 2020. We could add a WHERE statement, like below.

[1] `RLIKE` is not standard SQL, but most vendors include some regex functionality in one way or another.

```
SELECT major, AVG(salary) AS Average
FROM people
WHERE grad_year < 2020
GROUP BY major;
```

Note that as queries get longer and have more parts, they can be more readable if you break them across several lines.

HAVING If you want to restrict the results based on the grouped data, `WHERE` can't be used. Instead we have to use `HAVING`. For example, suppose from the previous query we just want to display those majors where the average is at least 80,000. We can do the following:

```
SELECT major, AVG(salary) AS Average
FROM people
WHERE grad_year < 2020
GROUP BY major
HAVING AVG(salary) >= 80000;
```

In general, `WHERE` is used when the restrictions made before the grouping happens and `HAVING` is used for restrictions after the grouping, which are often on the things calculated with SUM, COUNT, etc.

Here is another example using the `country` table in the `world` database built into MySQL. It shows all the regions of the world and their average life expectancies. But it restricts it to regions whose names start with the letter N, and it only shows those whose average life expectancy is over 70. Finally, it orders things from highest to lowest life expectancy.

```
SELECT region, AVG(LifeExpectancy) AS average
FROM country
WHERE region LIKE 'N%'
GROUP BY region
HAVING average > 70
ORDER BY average DESC;
```

Note that order is important. The WHERE clause must come before GROUP BY, and HAVING must come after GROUP BY.

Grouping by multiple things It is sometimes helpful to group by multiple things. For instance, the following will group by both continent and region.

```
SELECT continent, region, SUM(population)
FROM country
GROUP BY continent, region
ORDER BY continent;
```

5 Subqueries

A subquery is a query within a query. Here is an example of where one would be useful. Using MySQL's world database, suppose we want to show all the countries that have a population larger than France's. One approach would be to get France's population and then hardcode it into another query, like below:

```
SELECT population FROM country WHERE name = 'France';  -- returns 59225700
SELECT name FROM country WHERE population > 59225700;
```

A better approach is to use a subquery, like this:

```
SELECT name FROM country WHERE population >
    (SELECT population FROM country WHERE name = 'France');
```

Subqueries must always be enclosed in parentheses, like we see above. As a second example, let's list all the countries and their largest cities by population. To start, suppose we just want the largest city in the US. We could do that as follows:

```
SELECT name
FROM city
WHERE CountryCode = 'USA'
ORDER BY population DESC
LIMIT 1;
```

Now let's use that as a subquery, like this:

```
SELECT name,
       (SELECT name FROM city
        WHERE CountryCode = 'USA'
        ORDER BY population DESC
        LIMIT 1) AS `largest city`
FROM country;
```

In the results, each country name is followed by the string "New York", the largest city in the U.S. according to the database. To get it to list the largest city in each country, replace `USA` with `C.code`, referring to the country from the outer query. This is called a *correlated subquery*, a subquery that refers to fields from outside query.

```
SELECT name,
       (SELECT name FROM city
        WHERE CountryCode = C.code
        ORDER BY population DESC
        LIMIT 1) AS `largest city`
FROM country C
ORDER BY name;
```

6 Joins

SQL is a relational database. Data is stored in various tables that are related to each other via foreign keys. When you need data from multiple tables at once, a join is used.

For example, suppose we have a table of courses offered at a college. To keep things simple, assume each course has just a title, number of students enrolled, and a room. Each room itself has its own information, say a room number and how many people it can hold. It's usually considered good practice in SQL in a situation like this to break things into two tables, one for courses and one for rooms. Here is how those two tables might be created:

```
CREATE TABLE rooms(
    room_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    room_number INTEGER,
    capacity INTEGER
);

CREATE TABLE courses(
    course_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(100),
    enrollment INTEGER,
    room_id INTEGER,
    FOREIGN KEY (room_id) REFERENCES rooms(room_id)
);
```

When we break things into two tables like this, we need a way of connecting them. That's what the foreign key line is above. It links the courses and rooms tables together. When adding data to the `courses` table, we might do something like this:

```
INSERT INTO courses (title, enrollment, room_id) VALUES ('Database', 20, 1);
```

We specify the room's ID when doing the insert. Now suppose we want a listing of the titles of all the classes in Room 101. We can do that as follows:

```
SELECT courses.title
FROM courses
INNER JOIN rooms USING(room_id)
WHERE room_number = 101;
```

This uses `INNER JOIN` to put the two tables together.[1] The reason for the join is that we need the course title information from the `courses` table and the room information from the rooms table. When using a join, we need to tell what column to do the join based on. In this case, the shared column is `room_id`.

Here is a query that returns the titles of all the courses whose enrollment exceeds the room capacity.

```
SELECT title
FROM courses
INNER JOIN rooms USING (room_id)
WHERE enrollment > capacity;
```

Here is another example, this time using MySQL's `world` database. There is a table called `city` that contains

[1] Besides inner joins, there are also outer joins, which will be covered later.

info about various cities and there is a table called `country` that has info about countries. The `city` table has a field called `countrycode` that is the link between it and the `country` table. The `country` table doesn't have a field called `countrycode`, but it does have a field called `code` that is the same thing.

Suppose we want to list all the cities, their population, and the name of the country they are in. We will need a join here since the country names are only in the `country` table. Here is what that will look like:

```
SELECT city.name, city.population, country.name
FROM city
INNER JOIN country ON city.countrycode = country.code;
```

There are two things here that are a bit different from the first example of joins we looked at. First, note that we are using the names of the different tables in `city.name` and `country.name`. This is because both tables have a column called "name" and we need a way to distinguish them. Second, instead of `USING`, we use `ON` to tie the tables together. The `ON` syntax always works. The `USING` syntax works only if the columns being joined have the same name. Here they don't since one is called `code` and the other is called `countrycode`.

Note that in joins, we can give the tables aliases. This is sometimes nice if the table name is long and we need to refer to it a bunch. For instance, we could rewrite the query above like this:

```
SELECT ci.name, ci.population, co.name
FROM city ci
INNER JOIN country co ON ci.countrycode = co.code;
```

As another example, suppose a school has a database for students and courses. The `students` table has information on students like `student_id`, `name`, and year. The `courses` table has information on courses, like `course_id`, `title`, and `credits`.

To keep track of which students are taking which courses, we introduce a third table `student_courses`. It will be used to keep track of which students are taking which courses. It will only have a field for the student ID and one for the course ID. This is how it could be defined:

```
CREATE TABLE student_courses(
    student_id INTEGER,
    course_id INTEGER,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES students(student_id),
    FOREIGN KEY (course_id) REFERENCES courses(course_id)
    );
```

Notice that we have foreign keys for both the student and the course. The primary key for this table is an ordered pair of both the student ID and course ID. This sort of construction is very common in databases. It is sometimes called a *junction table*. Below is some hypothetical data in these tables.

student_id	name	year
1	Smith	soph
2	Roberts	fresh
3	Johnson	senior

course_id	title	credits
1	Database	3
2	Java	3
3	Calculus	4

student_id	course_id
1	2
1	3
2	3
3	1

The first row of the `student_courses` table says that Student #1 is taking Course #2, the second row says that Student #1 is taking Course #3, etc. Suppose we wanted to know the titles of all the courses Smith is taking. We will have to join three tables to get the information we need: the `student_courses` table since it keeps track of who is taking what, the `students` table since that will tell us what Smith's name is, and the `courses` table since it has the course titles. Here is the query:

```
SELECT title
FROM student_courses
INNER JOIN students USING(student_id)
INNER JOIN courses USING (course_id)
WHERE name = 'Smith';
```

One thing to note about this is the order of the joins in this case doesn't matter. It usually doesn't matter too much except that it is usually faster for MySQL if smaller tables come before larger ones in the join.

7 Other useful SQL topics

IN, EXISTS

Subqueries are useful when combined with the `IN` and `EXISTS` statements. Recall that the `IN` operator tells if something is in a list of things, like below:

```
SELECT * FROM country WHERE region IN ('Caribbean', 'Central Africa', 'Middle East');
```

Here is an example with a subquery. This sums up the population of all the countries where Dutch is spoken.

```
SELECT SUM(population) FROM country WHERE code IN
    (SELECT countryCode
     FROM countryLanguage
     WHERE language="Dutch");
```

We can also use `NOT IN` to check if something is not in a list. For instance, suppose we want all the countries in Africa that have a government form that is different from all the government forms in Asia. We use a subquery to get all the government forms in Asia and then use `NOT IN`.

```
SELECT name, governmentForm
FROM country
WHERE continent = 'Africa'
 AND governmentForm NOT IN
  (SELECT governmentForm FROM country WHERE continent='Asia');
```

The `EXISTS` keyword tells if a subquery returns any results or not. For example, the query below returns all continents in which Spanish is spoken.

```
SELECT DISTINCT continent FROM country c1 WHERE EXISTS
    (SELECT 1
     FROM country c2
     INNER JOIN countrylanguage ON c2.code = countrylanguage.countrycode
     WHERE language='Spanish' AND c1.code = c2.code);
```

Note that we just selected the number 1 in the subquery. We could have also selected names or IDs. It doesn't really matter since we are just looking to see if anything at all is returned. Note also that this is a correlated subquery. The subquery needs use the country from the outer query.

We can also do `NOT EXISTS`. For example, the query below returns all the continents that don't have a country whose name starts with D.

```
SELECT DISTINCT continent
FROM country c1
WHERE NOT EXISTS
    (SELECT name FROM country c2
     WHERE name LIKE "D%" AND c1.continent=c2.continent);
```

`NOT IN` might not work if the subquery's results could contain nulls. For instance, suppose we want to list all the countries that don't share an independence year with an African country. We might try the following.

```
SELECT name, indepyear
FROM country
WHERE continent != 'Africa' AND indepyear NOT IN
    (SELECT indepyear FROM country WHERE continent='Africa');
```

It would work except that some countries don't have independence years. This messes with the `NOT IN` operator. The fix for this is to use `NOT EXISTS` and change the subquery around a bit.

```
SELECT name, indepyear
FROM country c1
WHERE continent != 'Africa' AND NOT EXISTS
    (SELECT 1 FROM country c2 WHERE continent='Africa' AND c1.indepyear = c2.indepyear);
```

CASE statements

SQL has a `CASE` statement that is useful for conditions. For example, suppose we have a table of employee data with a name and start year. Depending on the start year, employees are put into different groups. We could do the following:

```
SELECT employee_name,
```

```
CASE
    WHEN start_year >= 2024 THEN 'newbie'
    WHEN start_year BETWEEN 2010 AND 2023 THEN 'associate'
    ELSE 'longtime employee'
END AS `employee group`
FROM employees;
```

The syntax takes a little getting used to, but it behaves a lot like if statements in other languages. If the cases are all just equalities based on a single column, then something like the following can be done.

```
SELECT employee_name,
CASE state
    WHEN 'Pennsylvania' THEN 'PA'
    WHEN 'Maryland' THEN 'MD'
END AS `state abbrev.`
FROM employees;
```

Case statements can be used in many other places, like in WHERE conditions, updates, etc.

UNION

Sometimes, you need the same information from two tables returned in a single query. For instance, maybe there is a table called `pa_cities` that lists all the cities in Pennsylvania and their populations, and there is a similar table for Maryland. To return all the cities in either state with a population over 100,000, we could use the following:

```
SELECT name, population FROM pa_cities WHERE population > 100000
UNION SELECT name, population FROM md_cities WHERE population > 100000;
```

The UNION keyword puts the two queries together into a single query. For UNION to work, both queries must return the same number of columns.

By default, if there are duplicate entries between the two columns, UNION will only show them once. It sort of applies a DISTINCT to the results. If you want duplicates shown, use UNION ALL in place of UNION. For instance, the following query lists all the countries in MySQL's world database twice in a row each.

```
SELECT name FROM country
UNION ALL
SELECT name FROM country
ORDER BY name;
```

ANY, ALL

You might sometimes see the ALL and ANY operators applied to subqueries. Here is an example that lists the countries in Europe that have more people than any country in the Northern African region.

```
SELECT name, population
FROM country
WHERE continent = 'Europe' AND population > ANY
   (SELECT population FROM country WHERE region = 'Northern Africa');
```

If we replace ANY with ALL, then the query would list the countries in Europe that have more people than every country in the Northern Africa region.

Note that the ANY query could instead be done by checking if `population` is greater than or equal to the minimum population in the Northern Africa region. The ALL query could be done using the maximum population.

8 More about joins

Understanding joins

To help understand joins, here are two simple tables to represent employees and offices. Each office has a room number and a building. Each employee has a name and an office, which is a foreign key into the `offices` table.

```
CREATE TABLE offices(
    office_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    building VARCHAR(100),
    room_number INTEGER
);

CREATE TABLE employees(
    employee_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    office_id INTEGER,
    FOREIGN KEY (office_id) REFERENCES offices(office_id)
);
```

We'll insert the following data into the tables.

```
INSERT INTO offices (building, room_number) VALUES
 ('Business Center', 102),
 ('Business Center', 302),
 ('Physical Plant', 201);

INSERT INTO employees (name, office_id) VALUES
 ('Alice', 1),
 ('Bob', 2),
 ('Carol', 3),
 ('Dewey', 2),
 ('Eric', NULL);
```

Below is a basic inner join that lists each employee and the room number of their office. Specifically, it will list Alice 102; Bob 302; Dewey 302; Carol 201.

```
SELECT name, room_number
FROM employees
INNER JOIN offices USING(office_id);
```

This looks through both the `employees` and the `offices` table. The `USING(office_id)` looks for *matches*. That is, we are looking for where an entry in the `employees` table has an `office_id` that matches an `office_id` in the `office` table, and when it finds one, it ties those two together.

There is another type of join, called an *outer join*, that is often useful. Notice in the example above, Eric is not in the results. The reason is that he doesn't have an office, so there is no way for a match to be made on `office_id` between the two tables. If we want all the employees to be listed, even if they don't have an office, we use a left outer join.

```
SELECT name, room_number
FROM employees
LEFT JOIN offices USING(office_id);
```

This returns all the same results as the previous query and for Eric it lists a NULL as his office number. Note that we just say LEFT JOIN. We could put the word OUTER in there, but it's usually left out.

In general, left joins are useful if you are pulling data from a table and want to add extra info from another table, but you don't want the lack of extra info for a specific row to prevent that row from being shown.

For instance, suppose we have a table of users of an app and another table that lists contact info of those users. Maybe we want to list info about the users and also their contact info, if available. We use a left join so that the users are all displayed even if they don't have any contact info. An inner join would only display those users with contact info.

There is also a right join, but it's rarely used since you can accomplish the same thing as a right join by using a left join and reversing the order of the tables in the join.

If you do a join without a USING or ON statement, it will return all ordered pairs with the first part of the pair coming from the first table and the second part coming from the second table. Here is an example:

```
SELECT name, room_number
```

```
FROM employees
INNER JOIN offices;
```

It returns Alice 201; Alice 302; Alice 102; Bob 201; Bob 302; Bob 102; ..., all 15 possible pairs of a name and a room number. Mathematically, this is the *Cartesian product* of the names and room numbers. This type of join is occasionally useful. A lot of times, however, it is an error where the query writer forgot to put in the USING/ON statement. If we really do want this, to make our intent clear, we should say CROSS JOIN, like below.

```
SELECT name, room_number
FROM employees
CROSS JOIN offices;
```

One example where a cross join is useful is if we had a table of employees and a table of managers and we want to pair off each employee and manager for a competition. We could do a simple cross join to get all the pairs.

It's possible to join on other conditions besides just on some foreign key matching up. Here is a bit of a goofy example using the `employees` and `offices` tables. It returns all pairs of employees and office numbers where the first letter of an employee's name matches the first letter of a building's name

```
SELECT name, room_number
FROM employees
INNER JOIN offices ON LEFT(name, 1) = LEFT(building, 1);
```

In particular, this returns Bob 302 and Bob 102 since offices with those room numbers are both in the Business Center, which starts with the same letter as Bob.

Here is a more practical example. Suppose we want a tournament where every employee plays every other employee in one game, with no one playing themselves. We can do this as follows:

```
SELECT a.name, b.name
FROM employees a
INNER JOIN employees b ON a.employee_id < b.employee_id;
```

This is sometimes called a *self join* since we are joining a table with itself. The `ON` condition here makes it so the pairs are limited to those where the employee ID of the first half of the pair is less than the employee ID of the second half. This eliminates cases where the employee IDs are equal, so no one is matched up with themselves, and it also makes it so that we'll get a pair such as (Alice, Bob), but not also (Bob, Alice).

Joining to subqueries

One useful query technique is that subqueries can go into FROM and JOIN statements.

Using MySQL's world database, suppose we want all the countries that have more people than the entire population of South America. Here is one way to approach the problem:

```
SELECT name
FROM country
WHERE population >
    (SELECT SUM(population)
     FROM country
     WHERE continent='South America');
```

Here is a second approach that involves joining to the subquery:

```
SELECT name
FROM country
INNER JOIN
   (SELECT SUM(population) AS total
    FROM country
    WHERE continent='South America') AS sa_total
WHERE population > total;
```

Both work, but the second one might run faster. In the first approach, the subquery is recomputed for every row of the `country` table. In the second approach, the subquery is just computed once. However, under the hood MySQL might optimize the first one and cache the result of the subquery.

Note that when joining to a subquery, MySQL requires that you give the subquery an alias, even if you don't use the alias.

Here is another example that lists all the countries and what percentage of the world's population they have:

```
SELECT name, population / total
FROM country
INNER JOIN
 (SELECT SUM(population) AS total
  FROM country) AS world_total
ORDER BY population / total DESC;
```

Below is an example that lists all the countries ranked by their most populous city. Because of the use of the `RANK` function (covered a little later in these notes), it really helps to join to a subquery here.

```
SELECT name, m, RANK() OVER (ORDER BY m DESC)
FROM country
INNER JOIN
 (SELECT countrycode, MAX(population) AS m
  FROM city
  GROUP BY countrycode) AS x
ON country.code = x.countrycode;
```

Here is another example that shows all the countries that have at least one city in the `city` table.

```
SELECT name
FROM country
WHERE EXISTS
  (SELECT 1 FROM city WHERE city.CountryCode = country.Code);
```

Here is how to do that by joining to a subquery:

```
SELECT DISTINCT country.name
FROM country
INNER JOIN city ON city.CountryCode = country.Code;
```

Here is one more example that is tricky to do without joining to a subquery. It lists all the cities of the `city` database, ranked in order by population, and it lists the country they are in.

```
SELECT RANK() OVER (ORDER BY city_pop DESC) AS ranking, city_name, name, city_pop
FROM country c
INNER JOIN LATERAL
 (SELECT name AS city_name, population AS city_pop
  FROM city
  WHERE countrycode = c.code
  ORDER BY population DESC
  LIMIT 1) AS x;
```

Note the keyword LATERAL here. We need it any time we do a join to a subquery and we want the subquery to refer to the other table in the join (in this case the `country` table).

A few useful functions

This section covers some of the more useful functions built into MySQL. The full documentation for functions is here: https://dev.mysql.com/doc/refman/8.0/en/functions.html.

String functions Below are a few useful string functions. There are many more.

- `CONCAT` for string concatenation. It takes a variable number of arguments and concatenates them all together. For example:

  ```
  SELECT CONCAT('Dr.', first_name, ' ', last_name) FROM people;
  ```

- A related function is `GROUP_CONCAT`. Here are two examples:

  ```
  SELECT GROUP_CONCAT(last_name) FROM people;
  SELECT GROUP_CONCAT(last_name SEPARATOR ';') FROM people;
  ```

  By default, `GROUP_CONCAT` returns all the results as a single string with each item separated by commas. You can change the comma to something else, like in the second line above. The `GROUP_CONCAT` function is useful in queries using GROUP BY. For instance, the query below returns the continents and a list (as a string) of the countries in them:

  ```
  SELECT continent, GROUP_CONCAT(name) AS `country list`
  FROM country
  GROUP BY continent;
  ```

- `LCASE, UCASE` – converts something to all lowercase or uppercase.
- `LEFT` — Returns the substring consisting of the leftmost so many characters. There is also `RIGHT` for the rightmost characters, and `SUBSTRING` for more general substrings.
- `POSITION` — Returns the first location of a substring in a string. For instance, `POSITION('-' IN name)` returns the location of the first hyphen in a field called `name`.

  To get all the characters up to the first space in a column called `name`, we could use `LEFT(name, POSITION(' ' IN name))`.
- `REPLACE` — Replaces occurrences of a string with another.
- `FORMAT` – This can be used turn a number into a nicely formatted value, rounded to a specific number of decimal places and with commas. For instance, `FORMAT(mileage, 1)` would take a mileage of 42301.396 and display it as 42,301.4. The second argument of the function is how many decimal places to use. If you just want commas inserted into a whole number, use 0.

Date and time functions It's tricky to try to do your own math on dates and times, especially since months have varying numbers of days, the rules for leap years are a little weird, and leap seconds are unpredictable. Instead, it's good to rely on the ones built into SQL. Below are some useful functions. There are many more.

- `NOW` — Returns the current date and time.
- There are various functions, like `YEAR`, `DAY`, `DAYNAME`, `MONTH`, `HOUR`, etc. that let you extract parts of a date or time. Here is one example that just returns the years people are born.

```
SELECT YEAR(birthday) from people;
```

- `TIMESTAMPDIFF` – This returns how much time has passed between two times/dates. The function's first argument is the type of unit. The second and third arguments are the `DATETIME` or `TIMESTAMP` items. Here is an example to find someone's age.

```
SELECT TIMESTAMPDIFF(YEAR, birthday, NOW()) AS age FROM people;
```

- `DATE_ADD` – This allows you to add a specified amount of time to various date objects. Below is a simple example. For the exact syntax, consult the MySQL documentation.

```
SELECT DATE_ADD(NOW(), INTERVAL 3 DAY);
```

- `DATE_FORMAT` – The `DATETIME` format is YYYY-MM-DD. The `DATE_FORMAT` function can be used to put things into a nicer format. Here is an example that returns birthdays in a format that looks like `January 1, 2024`.

```
SELECT last_name, DATE_FORMAT(birthday, '%M %d, %Y') AS birthday
FROM people WHERE birthday>'1990-01-01' ORDER BY birthday;
```

  The `%M` formatting code is for a full month name, `%d` is for the day as a number, and `%Y` is for a four-digit year. There are a few dozen different codes. See the MySQL documentation for all the possible formatting codes.
- `STR_TO_DATE` – This is the opposite of `DATE_FORMAT`, for if you have a date in a different format and want to convert it to the YYYY-MM-DD format. Here is an example.

```
UPDATE people SET birthday=STR_TO_DATE('May 1, 1999','%M %d,%Y') WHERE person_id=1;
```

A few others Here are a few miscellaneous functions.

- `ROUND` – Rounds a number to a specified number of places, like `ROUND(salary, 0)` to round a salary to the nearest whole number.
- `CAST` – Sometimes, we'll have data in one format that we'll need to convert into another. For instance, maybe we have a database that is storing times as strings instead of using the `TIME` data type. If we want to use MySQL's time functions, we would first have to convert the string into a `TIME`. If the field storing the time is called `start_time`, we could do that with `CAST(start_time AS TIME)`.

Window functions

SQL has functions `RANK`, `DENSE_RANK`, `PERCENT_RANK`, and `NTILE` that are useful for getting rankings. Here is an example:

```
SELECT name, population,
 RANK() OVER (ORDER BY population DESC) AS ranking
FROM country;
```

The ranking part shows each country's numerical rank in population (#1, #2, etc.). Note the syntax uses the keyword OVER followed by an ORDER BY clause in parentheses that says what to rank on.

The `DENSE_RANK` function differs from `RANK` in how ties are handled. If we have two things tied for first, `RANK()` will assign #3 for the next country, while `DENSE_RANK` will assign #2 for that country. The `PERCENT_RANK` function gives the percentile of each entry. The `NTILE` function breaks things up into percentile groups. For instance, `NTILE(4)` breaks things up into quartiles.

The `OVER` keyword take an optional `PARTITION BY` clause that specifies something to group things by. Here is an example that gives each country's rank in terms of population in the continent it belongs to.

```
SELECT name, continent,
  RANK() OVER (PARTITION BY continent ORDER BY population DESC) as num
FROM country;
```

Sometimes `PARTITION BY` is useful on its own with `AVG`, `SUM`, or `COUNT`. For instance, the following query shows each country and the average population of the continent it is on:

```
SELECT name,
  AVG(population) OVER (PARTITION BY continent) AS `avg pop`
FROM country;
```

9 Connecting a database to a software project

We'll cover connecting MySQL to a few popular languages. The same concepts often work for other languages and other databases, though the exact imports and function names will be different.

Python

You'll want to install the `mysql-connector-python` package. If Python is in your path, you can try the following line at a command prompt:

```
pip3 install mysql-connector-python
```

If you're lucky, that will work perfectly. If you're on Windows and you get an error saying the `pip3` command can't be found, try opening up a *fresh* Python shell. Then type the following:

```
import os
print(os.getcwd())
```

This will tell you where Python is installed. Copy that location and use the `cd` command at the OS command prompt to change to that directory. Your command will look something like `cd c:\users\...`, though the exact directory will vary based on your OS and setup. Then type `cd scripts` to change to the `scripts` subdirectory. Now run the `pip3` command given earlier. Hopefully that should work.

Below is a very basic Python program to talk to a MySQL database.

```
import mysql.connector

connection = mysql.connector.connect(host='localhost',
                                     database='world',
                                     user='root',
                                     password='your_mysql_password_here',
                                     port = 3306)
```

```
cursor = connection.cursor()
cursor.execute('SELECT Name, Population FROM country')

for x in cursor.fetchall():
    print(f'Name: {x[0]}    Population: {x[1]}')

connection.close()
```

After the import line, we connect to the database. The host will be `localhost`, which refers to your own computer. Once you decide to move your project to the internet somewhere, you'll likely need to change this. The `database` field is for the database schema you are connecting to (shows up in the left tab in the MySQL Workbench). In this case, we are just using MySQL's built in `world` database. For user, we have `root`, which is the default user that is typically set up with the MySQL installation. Change this if you have a different user name. The port number, 3306, is the standard MySQL port. You can leave this off usually. You only need to specify a port number if you're running MySQL at something other than port 3306.

The username and password should not be stored in the code, like we've done here, for any serious project, especially something where the source code would be visible to anyone. It's better to read them in from a private configuration file, store them in environment variables on the system, or something else. But for simplicity, we've just put them in the code, which is fine for testing things out on your own device.

After connecting to the database, we create a cursor object. With a cursor object, we can run a query. We can use `cursor.fetchall()` to get the query's results back as a list of tuples.

If you want to plug in data from the program into the query, like maybe something user-entered, do something like this:

```
continent = input('Enter continent:')
query = '''SELECT Name, Population FROM country WHERE continent=%s AND population>=%s'''
cursor.execute(query, (continent, 20000000))
```

While we could use string concatenation to paste the user input directly into the query, don't, as that could lead to SQL injection, a dangerous vulnerability covered a little later in these notes.

You can also do queries that change a database, say by inserting data, updating data, etc. You will need to run `connection.commit()` after the query in order for the changes to be saved in the database.

The final line of the code above is `connection.close()`. For a short script like this, it's not necessary as Python will close the connection when the program ends. However, in a real system, it's good to close a connection when you are done with it to avoid chewing up unnecessary resources. Databases do have a limited number of connections that can be open at a time.

Java

The same concepts we saw with the Python code apply to the Java code below, so we won't explain it in as much depth. To run this, you'll need to download the MySQL Connector driver from https://dev.mysql.com/downloads/connector/j/. You'll probably want to download the "platform independent" option. It's a zip file. Locate the executable jar file in it, and put it in a folder somewhere on your computer. Then use your IDE to include it as a project library. If you're not sure how to do that, search online or ask an AI for instructions.

```
import java.sql.*;

public class MySQLJava {
    public static void main(String[] args) {
        Connection conn = null;
        PreparedStatement stmt = null;
        ResultSet results = null;

        try {
            conn = DriverManager.getConnection(
                    "jdbc:mysql://localhost:3306/world",
                    "root",
                    "your_mysql_password_here"
            );

            stmt = conn.prepareStatement("SELECT Name, Population FROM country WHERE continent = ?
                                         AND population > ?");
```

```
            stmt.setString(1, "South America");
            stmt.setInt(2, 20000000);

            results = stmt.executeQuery();
            while (results.next()) {
                System.out.println(results.getString("name") + " " + results.getInt("population"));
            }

        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                if (results != null) results.close();
                if (stmt != null) stmt.close();
                if (conn != null) conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

R

For R, it is recommended to install the `RMariaDB` package. That package also needs `Rcpp`. The code below installs them and demonstrates a simple way to work with them.

```
install.packages("RMariaDB")
install.packages("Rcpp")

library(RMariaDB)

con <- dbConnect(
  RMariaDB::MariaDB(),
  user = "root",
  password = "your_mysql_password_here",
  host = "localhost",
  port = 3306,
  dbname = "world"
)

print(dbGetQuery(con, "SELECT name FROM country LIMIT 10"))

query <- "SELECT name, population FROM country WHERE continent=? AND population > ?"
stmt <- dbSendQuery(con, query)
dbBind(stmt, list('South America', 20000000))
print(dbFetch(stmt))
```

PHP

Probably the most common place you'll see a database used is on the internet. Many websites work like this: Users enter data on the site. That data is sent to the server. Code on the server queries the database, formats the results, and returns them to the user. Almost any reasonable programming language can be used on the server to talk to the database. Some of the most common are PHP, JavaScript, Python, and Java. We will look at PHP here since it is one of the easier ones to get set up and working.

To test it out on your local machine, one option is to install XAMPP. Once it's installed, open the XAMPP control panel. Just start Apache, but don't start their version of MySQL, since you already have one running. Make sure to save your HTML and PHP files in XAMPP's htdocs directory. On Windows, it's `c:\xampp\htdocs`. To test out your installation, put `<?php echo "hello world"; ?>` into a file, save it as `hello.php` in the `htdocs` directory, and in your browser, go to `localhost/hello.php`.

Below is a combination of PHP and HTML code that creates a simple form, processes that form, queries the database, and displays the results.

```
<?php
$servername = "localhost";
$username = "root";
$password = "your_mysql_password_here";
$database = "world";
$port = 3306;

$conn = new mysqli($servername, $username, $password, $database, $port);
```

```
if ($conn->connect_error)
        die("Connection failed: " . $conn->connect_error);

?>
<!DOCTYPE html>
<html>
    <head>
        <title>Testing</title>
    </head>
    <body>
        <form method="post">
            Continent name <input type="text" name="continent"><br>
                        Population cutoff <input type="text" name="cutoff"><br>
            <input type="submit">
        </form>
        <?php
        if (isset($_POST["continent"]) && isset($_POST["cutoff"])) {
            $stmt = $conn->prepare("SELECT name, population FROM Country WHERE continent = ?
                                    AND population > ?");
            $stmt->bind_param("si", $_POST["continent"], intval($_POST["cutoff"]));
            $stmt->execute();
            $result = $stmt->get_result();

            if ($result->num_rows > 0)
                while($row = $result->fetch_assoc())
                    echo "Name: " . $row["name"]. " Population " . $row["population"]. "<br>";
            else
                echo "No results";
        }

        ?>
    </body>
</html>
```

Many of the concepts here are similar to the ones we've seen with other languages. The main difference is that this file contains both client (web browser) and server code. The server code is contained within the `<?php` and `?>`. That code is only seen by or run by the server. On the client, there is the HTML form. When the user clicks the submit button, the data is sent to the server. It accesses the user's data via the `$_POST[]` associative array (dictionary).

The page above is self-contained. Sometimes, you want a separate PHP file to handle processing the form. To do so, add something like `action="some_other_file.php"` into the HTML form tag.

Here is another example of a database query that is used to create a drop-down menu with each country name appearing in it.

```
<form method="post" action="basic_form4.php">
    <select name="country">
        <?php

        $result = $conn->query("SELECT Name FROM country");
        while($row = $result->fetch_assoc())
            echo "<option value='" . $row["Name"] . "'>" . $row["Name"] . "</option>\n";

        ?>
    </select>
    <input type="submit">
</form>
```

SQL Injection

One of the most serious vulnerabilities in websites is SQL injection. It involves an attacker putting SQL code into a web form to trick a server's database into running the attacker's query.

Here is an example of some PHP code that takes some user input and inserts it into an SQL query.

```
$result = $conn->query("SELECT * FROM products WHERE name='" . $_POST["name"] . "'");
```

If the person enters the name "chocolate", then it generates this query:

```
SELECT * FROM products WHERE name='chocolate'
```

This says to take all results from the products table that have a name of chocolate. Here is something an attacker could enter into the form: `'OR 1=1#`. It generates the SQL query below.

```
SELECT * FROM products WHERE name='' OR 1=1#
```

The initial quote of what the attacker enters closes off the quote from the PHP code. Then we have `OR 1=1`. Remember that logical OR evaluates to true if either of its two operands are true. Since `1=1` is guaranteed to be true, this means the condition will evaluate to true. The end result is that it will display all the items in the database. The `#` at the end is a comment which makes sure the closing quote from the PHP code doesn't cause a syntax error with the attacker's code. Here is another input an attacker could enter. You can probably guess what it will accomplish:

```
' and 1=2 union select user, password,0,0 from mysql.user #
```

Here is another example. Below is a line of PHP code that takes a user-entered name and password and attempts to pull information about the user from the system.

```
$result = $conn->query("SELECT displayname FROM people WHERE user='" .
                       $_POST["user"] . "' AND password='" . $_POST["password"] . "'");
```

Here are a few things we could enter into the user field:

- `' OR 1=1#` — Similar to what we saw above, this will show the value of the `displayname` for every user in the system.
- `' AND 1=2 UNION SELECT user FROM people#` — This will give us a list of all the user names in the system.
- `root'#` — Assuming there is a user called root, this will log them in without a password. This works by commenting out the password part of the query so that the system doesn't bother to check the password.

We could do even worse things like insert new users into the system or delete the whole database via a `DROP TABLE` command.

Avoiding SQL injection The root cause of the weakness in the two code samples above is that we are pasting user-entered data directly into the query. So if someone enters things into the form that look like SQL code, the SQL program running on the server will execute them just as if they are code. To avoid this, we use something called *prepared statements* or *parameterized queries*. In these, we rely on whatever programming language we're using to do the pasting for us, trusting it to make sure that whatever is entered will not be interpreted as SQL code. So if someone enters `' OR 1=1#` into a product search, that text will be interpreted as searching for a product called `' OR 1=1#` and not as SQL code. Here is how this might work in PHP:

```
$stmt = $conn->prepare("SELECT * FROM people WHERE user = ? AND password = ?");
$stmt->bind_param("ss", $_POST["user"], $_POST["password"]);
$stmt->execute();
$result = $stmt->get_result();
```

The functions replace the placeholder question marks with the text that the user entered, and the `mysqli_stmt_bind_param` function takes care to make sure that replacement is done safely. In summary, never paste user input directly into an SQL query via concatenation. Use a prepared statement.

10 Managing a database

Table creation and data types

We have already seen how to create a simple table. Here is an example:

```
CREATE TABLE books(
    book_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    book_name VARCHAR(100),
    num_pages INTEGER
);
```

We had looked at some of the basic data types. Here we will look at more of them.

Integer types

For whole numbers, we have used `INTEGER`. This is a 32-bit integer, which means 32 bits or 4 bytes of memory are used to store it. This limited size means there is a limit to how large the integer can be. In particular, for a signed integer, the range is from around −2 billion to around +2 billion. For an unsigned integer, the range is from 0 to around 4 billion.[1] Note that if you want an unsigned integer, which gives a slightly larger range at the cost of not allowing negatives, it is declared as `INTEGER UNSIGNED`.

If you need something larger, MySQL has a 64-bit `BIGINT` data type (like a `long` in many programming languages). This will allow integers up to around 10^{18}. If you still need something bigger, you could declare the data type as something like `NUMERIC(30, 0)`, which would be a 30-digit number. However, this won't be as fast as an ordinary integer since it is represented in software, rather than hardware.

MySQL also has `TINYINT`, `SMALLINT`, and `MEDIUMINT` data types, that are 8, 16, and 24 bits, respectively. The main reason for these is to save space. For instance, if you have a database with a billion rows, and one of the fields is an integer you know will never get very large, you could use a `TINYINT` instead of an `INTEGER`. This will save around 3 bytes of space per row, which is around 3 gigabytes for the whole table. On the other hand, if your table is relatively small, it's probably not worth the trouble of micromanaging the exact sizes of every integer field.

Note also that MySQL does have a boolean data type (called `BOOLEAN`), but under the hood it's actually just a `TINYINT` with `TRUE` standing for 1 and `FALSE` standing for 0.

Decimal numbers

For decimal numbers, `NUMERIC` is the type we learned. MySQL also has `FLOAT` and `DOUBLE`. For most situations, `NUMERIC` is probably better, though `FLOAT` and `DOUBLE` are useful, especially for scientific data.

The big difference is that `NUMERIC` stores values exactly, while `FLOAT` and `DOUBLE` store them approximately. However, `FLOAT` and `DOUBLE` use can use CPU operations directly to do their calculations, so they are faster.

As you might have learned elsewhere, floating point numbers are stored using a particular binary format. Many numbers, when converted to binary, have an infinite (binary) decimal expansion. For instance, the binary expansion of .3 is .01001100110011..., that repeats 0011 forever. When this is stored, we have to cut it off somewhere, and that leads to inaccuracy. Because of this, .3 is actually stored as approximately 0.299999999999999989. This roundoff error can lead to surprising results if you're not aware of it. For instance, consider the following:

```
INSERT INTO products (price) VALUES (.2);
UPDATE products SET price = price + .1 WHERE id=1;
SELECT * FROM products WHERE price=.3;
```

The product whose price we changed will not be listed by the final SELECT statement. Roundoff error means that $.2 + .1$ actually gets stored as around 0.30000000000000004, which is different from the representation we saw for .3. There are ways around this, but usually it's best to use `NUMERIC` instead, especially when dealing with money amounts. `NUMERIC` stores values in more of a string-like format, and the math it uses is guarantees that a check such as $.1 + .2 = .3$ won't have to worry about roundoff error.

String data types

The `VARCHAR` data type is pretty flexible and the easiest to use for strings. However, there are a few other string data types to be aware of. One of them is `CHAR`. The difference between `VARCHAR(10)` and `CHAR(10)` is that the former stores a variable number of characters with a limit of 10, whereas the latter always stores exactly 10 characters. It will pad the string out with spaces to make sure it fills out 10 characters. Except for very small strings, `CHAR` will typically take up more space than `VARCHAR` since `VARCHAR` only uses as much space as needed

[1] Specifically, a signed integer ranges from -2^{31} to $2^{31}-1$ and an unsigned integer ranges from 0 to $2^{32}-1$.

to hold the string.[1].

You tend to see `CHAR` used in some older databases. It is useful in modern databases when you know your string must always be a particular length, such as for state abbreviations (MD, PA, NJ, etc.). In these cases, the database internally will handle the `CHAR` type a little more efficiently than `VARCHAR`. But if your string lengths vary, it's better to use `VARCHAR`.

MySQL also has `TEXT`, `MEDIUMTEXT`, and `LONGTEXT`. These are like `VARCHAR` in that they can hold text of varying lengths. However, they are specifically for larger blocks of text. `TEXT` has a maximum size of 64 KB, `MEDIUM TEXT` has a maximum size of 16 MB, and `LONGTEXT` has a maximum size of 4 GB. MySQL will often store the strings associated with these data types outside the main database, just storing a pointer to where they are located. This means it will be slower to access them. These data types are usually used for storing things like articles or blog posts.

Other data types

MySQL[2] has a special type called `ENUM` that gives a predefined set of values that the column can hold. For instance, we might have something like `class ENUM('Fr', 'So', 'Ju', 'Se')` that declares a column called `class` that can be one of those four values. If you try to assign a different value, MySQL will throw an error. Thus, enums can be useful to prevent data entry errors.

For dates and times, the data types are `DATE`, `TIME`, and `DATETIME`. There are a few others.

For storing binary data, like images or other files, there are `BLOB`, `MEDIUMBLOB`, and `LARGEBLOB`, which can store up to 64 KB, 16 MB, and 4 GB, respectively. An alternative to storing files themselves in the database is to store paths to the files (i.e. the file locations). There are pros and cons to each approach.

MySQL has a few other data types that are sometimes useful. One is `JSON`, for text in the JSON (JavaScript Object Notation) format. MySQL has functions for parsing the JSON. Another useful type is `GEOMETRY`, for storing geospatial data. MySQL has functions for working with this data, such as returning the distance between two points.

Constraints

When declaring a table, MySQL allows us to set various constraints on the data. Whenever data is added to the table or modified, MySQL will check to make sure the constraint is satisfied. When writing a program to interact with a database, we could also do the checks ourselves in the code, but if we set up the database with constraints, then MySQL will do the checking for us, saving us the trouble (and also saving us trouble if we forget to do a check or miscode something).

Below is an example table with several different constraints:

```
CREATE TABLE students(
    id INTEGER PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    school VARCHAR(100) DEFAULT "Mount St. Mary's",
    nickname VARCHAR(100) UNIQUE,
    age INTEGER CHECK (age >= 18)
);
```

For the `name` column, we see the `NOT NULL` constraint, which makes it so that the `name` field must always be included. We are not allowed to insert data without specifying the name.

For the `school` column, we use `DEFAULT`. When inserting data, if we don't specify the school, then default value is used instead of `NULL`.

For the `nickname` column, we use the `UNIQUE` constraint. This makes it so that no two students can ever have the same nickname in the system.

[1] `VARCHAR` also stores the length of the string, so `VARCHAR(2)` will actually use a little more space than `CHAR(2)`
[2] `ENUM` is not a standard data type in SQL. Each database vendor does them a little differently.

For the `age` column, we have a `CHECK` constraint. This example won't allow us to add anyone with age under 18. It also won't allow us to change anyone's age to be under 18.

You can use multiple constraints on any column. You can also use a `CHECK` constraint that references multiple columns. You could add a line in the `CREATE TABLE` statement like this: `CHECK name != 'Tom' AND age != 20`. This would mean the table could not have any 20-year-olds named Tom.

One useful default value is shown below. Whenever a row is added, this will record the moment it was added.

```
created DATETIME DEFAULT CURRENT_TIMESTAMP()
```

One other constraint-like feature is generated columns. These are columns that are calculated based on other columns. Here is an example:

```
CREATE TABLE people(
    id INTEGER PRIMARY KEY AUTO_INCREMENT,
    first_name VARCHAR(100),
    last_name VARCHAR(100),
    full_name VARCHAR(200) GENERATED ALWAYS AS (CONCAT(first_name, ' ', last_name)) STORED
    );
```

This example stores a person's full name based on their first and last names. While we could do this using SELECT when we query the database, what's nice about this approach is it doesn't have to be recomputed each time we run a new SELECT query. This can save time in some cases. The drawback is the extra space. Instead of `STORED`, we can use `VIRTUAL`, which means the column won't be stored in the database. In that case, we don't get much of a speedup, but if this is a frequent calculation, it's nice to have it once in the table definition instead of having to recode it in several queries.

11 Database management operations

This section is about a few common things to do with managing tables.

A database system contains various schemas, and each schema is further broken into tables. For instance, with MySQL, the built-in `world` database or schema has a `country` table, a `countryLanguage` table, and several others. You can use the MySQL Workbench to create new schemas, make certain schemas active, and do other things. However, it's nice to know that you can also do that directly via commands to the database. This is useful if you are working at a command prompt or if you connecting a program to a database and need the program, for instance, to create a schema. Below are four common commands and what they do:

```
CREATE SCHEMA bookstore;  -- creates a new database/schema
USE bookstore;            -- switches the active database
SHOW TABLES;              -- shows a list of all the tables in the active database
DESC books;               -- shows all the columns and their data types in the books table
```

Here are a few commands useful for deleting stuff:

```
DROP SCHEMA bookstore;
DROP TABLE books;
```

Sometimes in a script, you might want to create a table, but you might not be sure if it already exists. For this, you can add `IF NOT EXISTS` into the command. Something similar can be used when deleting a table.

```
CREATE TABLE IF NOT EXISTS books(...);
DROP TABLE IF EXISTS books;
```

This deletes a row from a table.

```
DELETE FROM books WHERE book_id=1;
```

The following deletes possibly many rows from the table, any book whose name starts with J.

```
DELETE FROM books WHERE book_name LIKE 'J%';
```

The MySQL workbench might give you an error if you try to do this. This happens any time you try to delete or update data using a WHERE condition that isn't just a simple check of a key value. This is to avoid common errors where you could accidentally delete or change a large part of a table. There is a setting to change if you

want to disable this protection. It's under Edit->Preferences->SQL Editor. All the way at the bottom is a "safe updates" checkbox you would have to uncheck.

If you want to delete all the data in a table but still keep the table, use DELETE FROM without a WHERE clause, like this:

```
DELETE FROM books;
```

To change the structure of a table, use the ALTER TABLE command. Here are a few common operations:

```
ALTER TABLE books ADD author VARCHAR(100);
ALTER TABLE books RENAME COLUMN author TO book_author;
ALTER TABLE books RENAME TO computer_books;
ALTER TABLE books CHANGE author book_author VARCHAR(100);
ALTER TABLE books DROP COLUMN pub_date;
ALTER TABLE books ADD CONSTRAINT page_check CHECK (num_pages <= 10000);
```

We have already learned how to insert data into a table. Here is an example:

```
INSERT INTO books (book_name, num_pages) VALUES ('database', 1000), ('java', 500);
```

We can also insert data from one table into another.

```
INSERT INTO prog_books (book_name, num_pages)
   SELECT book_name, num_pages FROM books;
```

To change an existing row or rows, use the UPDATE statement. We have already briefly covered it. Here is an example that adds 10 pages to all the math books and also changes the subject name slightly.

```
UPDATE books SET subject='Math', num_pages = num_pages + 10 WHERE subject = 'MATH';
```

Just like with inserting, you can update based on another query, with subqueries, joins, or other things. As a quick example, this query gives a raise to all the employees in room 101:

```
UPDATE employees SET salary = salary * 1.1 WHERE office_id =
  (SELECT office_id FROM offices WHERE office_num = 101);
```

If you want to back up a table, use something like the following:

```
CREATE TABLE books_backup AS SELECT * FROM books;
```

This will store the data from the table, but not any other details such as what the primary key is, default values, etc. You could also use a similar structure to create a new table from another that isn't an exact copy.

```
CREATE TABLE books2 AS SELECT id, name FROM books WHERE subject = 'Math';
```

Sometimes when deleting or updating, you could make a serious mistake. One option to recover from this is to make a backup of the table before making changes. Another option is *transactions*. Below is an example.

```
START TRANSACTION;
DELETE FROM books WHERE book_name LIKE 'J%';
SELECT * FROM books;
ROLLBACK;
```

In the above example, we start the transaction and then try to delete data. In the third line, we check to see what is now in the table. If we don't like what we see, we can use `ROLLBACK` to undo the change. If we are happy with the change and want it to be permanent, then we would use `COMMIT` instead of `ROLLBACK`.

Suppose we have the following two tables .

```
CREATE TABLE offices(
    office_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    building VARCHAR(100),
    room INTEGER
);

CREATE TABLE profs(
    prof_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    prof_name VARCHAR(100),
    office_id INTEGER,
    FOREIGN KEY (office_id) REFERENCES offices(office_id)
);
```

If we try to something like `DELETE FROM offices WHERE office_id=4`, MySQL will raise an error if there is any professor in that office (i.e., if there is a row in the `profs` table with that `office_id`). The reason for the error is MySQL wouldn't know what the right thing would be to do with that `office_id` for professors in the

deleted office. If you want to avoid the error, there are a couple of things you could do.

The first is to use the following when declaring the foreign key:

```
FOREIGN KEY (office_id) REFERENCES offices(office_id)
ON DELETE CASCADE
```

`ON DELETE CASCADE` causes the deletions to "cascade", that is, to cause deletions in the other table. In our example, when an office is deleted, all the professors in that office will be deleted. Here is a second option:

```
FOREIGN KEY (office_id) REFERENCES offices(office_id)
ON DELETE SET NULL
```

In this case, when the office is deleted, the `office_id` field for all the professors in that office will be set to NULL.

12 Useful database features

Views

A view is a way to create a sort-of mini table that has parts of another table or tables. Here is a simple example:

```
CREATE VIEW maryland AS SELECT * FROM counties WHERE state_name='Maryland';
SELECT * FROM maryland;
```

For this, we assume that `counties` is a table that contains data on all the counties in the U.S. This view is a reduced table that contains only the Maryland data. It is not really a table in that the data is not stored but is instead generated as needed. One use for this view is that it gives us a simplified table that we can use without having to worry about all the other stuff in the table.

Views are often useful for making other queries simpler. To demonstrate this further, suppose we have a database tracking student enrollments in courses. There are tables for the courses, the students, and a table called `student_courses` for who is taking which course. This last table has foreign keys to the other tables, so we always have to do joins to get what we want. We can create a view to simplify things.

```
CREATE VIEW course_info AS
SELECT student_name, course_title
FROM student_courses
INNER JOIN students USING(student_id)
INNER JOIN courses USING (course_id);
```

Then if we want the names of all the students taking Database, we could do `SELECT student_name FROM course_info WHERE course_title = 'Database'`. One other common use of views is to restrict access to sensitive information. Suppose we have a table of student info that contains social security numbers, GPA, credits taken, and the student's major. We want to give access to data on the number of credits and majors to various departments, but the social security number and GPA are sensitive info that most departments shouldn't have access to. We can create a view with just the safe information, like below:

```
CREATE VIEW restricted_data AS SELECT id, credits, major1 FROM students;
```

Functions and stored procedures

SQL has two related features, functions and stored procedures. Functions are for repetitive tasks that can call from inside a SELECT statement. Stored procedures are for sequences of operations that often will modify or change info in the database. They are usually for things you would do repeatedly. There are a lot of things to know to write functions and stored procedures. We will just hit a few highlights here. For more details, the MySQL tutorial at https://www.mysqltutorial.org/ is pretty good.

Here is a simple function to do a temperature conversion.

```
DELIMITER $$
CREATE FUNCTION convert_temp(temp NUMERIC(4,1))
RETURNS NUMERIC(4,1)
```

```
DETERMINISTIC
BEGIN
    RETURN temp * 9 / 5 + 32;
END $$
DELIMITER ;
```

Here is how it might be used.

```
SELECT city, date, convert_temp(high_temp) FROM temperature_data;
```

There are a few things to note about the function syntax. First, the parameters and their data types as well the return type are specified. Next comes the keyword `DETERMINISTIC`. This means there is no randomness in the function. The same inputs always return the same things. If the function uses random numbers or something like the `NOW()` function, then use `NOT DETERMINISTIC`. SQL uses `BEGIN` and `END` to indicate the start and end of the function's code. One unusual thing is the `DELIMITER` stuff. It is used to change the end-of-statement delimiter. Specifically, semicolons usually end statements. However, in functions and stored procedures, we will often have multiple statements to run, and we also need to mark the end of the function. It confuses MySQL if we use the semicolon for all those purposes, so we temporarily change the delimiter to `$$`, use that to end the function and then change the delimiter back to a semicolon. To create the function in the MySQL workbench, highlight the whole thing (from the first `DELIMITER` statement to the last) and use the lightning bolt button.

Here is a simple stored procedure.

```
DELIMITER $$
CREATE PROCEDURE shift_temperatures(IN shift INTEGER)
BEGIN
    DROP TABLE IF EXISTS temperature_archive;
    CREATE TABLE temperature_archive LIKE temperature_data;
    INSERT INTO temperature_archive SELECT * FROM temperature_data;
    UPDATE temperature_data SET high = high + shift;
END $$
DELIMITER ;
```

We would call it with something like `CALL shift_temperatures(10);`. The procedure shifts all the high temperatures in the `temperature_data` table by a specified amount. Before doing so, it archives the data, deleting any previously archived data.

Functions and stored procedures can use if statements and loops, and they can declare variables. Here is an example using an if statement and a variable. It declares a variable that will store how many data points are in the table. If there are less than 10, then we will raise an error. Otherwise, we do the same temperature shift as in the last example.

```
DELIMITER $$
CREATE PROCEDURE shift_temperatures(IN shift INTEGER)
BEGIN
    DECLARE temp_count INT;
    SELECT COUNT(*) INTO temp_count FROM temperature_data;

    IF temp_count < 10 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Not enough records.';
    ELSE
        CREATE TABLE IF NOT EXISTS temperature_archive LIKE temperature_data;
        INSERT INTO temperature_archive SELECT * FROM temperature_data;
        UPDATE temperature_data SET high = high + shift;
    END IF;
END $$
DELIMITER ;
```

Although we won't do any examples with them, we'll note that it's common to use transactions inside of stored procedures. An if/then statement can be used to check if something went wrong with an action, and then we can rollback or commit as needed.

This is as far as we will go with stored procedures. Like previously mentioned, there is a lot more to them. Stored procedures are useful for tasks that need to be done often, like end-of-the day processing of accounts for a bank. For complicated tasks, it might be better to use a programming language to interact with the database. Stored procedures can be a pain to debug.

Triggers

Triggers are a little like a stored procedure that runs before or after insertions, deletions, and updates. Here is an example:

```
DELIMITER $$
CREATE TRIGGER log_deleted_records
AFTER DELETE ON temperature_data
FOR EACH ROW
BEGIN
    INSERT INTO archive_temperature_data (city_name, date, high, low)
    VALUES (OLD.city_name, OLD.date, OLD.high, OLD.low);
END $$
DELIMITER ;
```

Right after something is deleted, this trigger runs to put a copy of all the deleted rows into an archive table. The `FOR EACH ROW` in the code makes sure that if multiple rows are deleted, then they are all archived. The `OLD` in the code refers to the row being deleted. For insertions and updates, there is also `NEW`. With inserts, it refers to the row being inserted. For updates, `OLD` refers to the old value of the row and `NEW` refers to its value after being updated.

Here is another example that records a timestamp each time a row is updated (it assumes that the table has a field called `last_update`).

```
DELIMITER $$
CREATE TRIGGER last_change
BEFORE UPDATE ON temperature_data
FOR EACH ROW
BEGIN
    SET NEW.last_update = NOW();
END $$
DELIMITER ;
```

Here is another silly example to show what you can do with triggers.

```
DELIMITER $$
CREATE TRIGGER prevent_new_york_delete
BEFORE DELETE ON temperature_data
FOR EACH ROW
BEGIN
    IF OLD.city_name = 'New York' THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Deletion of New York data is not allowed!';
    END IF;
END $$
DELIMITER ;
```

Indexes

If we're looking for a row in a database of 10 million rows, and it happens to be the last row in the database, then SQL will need to perform a slow linear search through all 10 million rows before it finds the one we want. We can speed this process up considerably by creating an index. Below are three ways to create an index on a field called `name` in a table called `employees`.

1. Add the line `INDEX name_index(name);` into the `CREATE TABLE` statement.
2. Use the command `CREATE INDEX name_index ON employees(name);`.
3. Use the command `ALTER TABLE employees ADD INDEX name_index(name);`

Once data is indexed, that changes how SQL stores the data. It will often use a data structure called a B-tree, which an efficient tree structure related to binary search trees. Now instead of taking linear time, it takes logarithmic time. Very roughly speaking, this means instead of taking a worst case of 10,000,000 operations, it would take a worst case of around $\log_2(10000000) \approx 23$ operations, a huge change.

Note that it is not necessary to create an index on a table's primary key. MySQL does that automatically. Further, if the `UNIQUE` keyword is used on a column, then an index is automatically created for that column.

Though they can give a huge speedup, be careful about overusing indexes. The speedup indexes give us comes at a cost. First, the B-tree data structure takes up more space than the regular approach. Second, while SELECT queries are faster, insertions and deletions will be slower because they involve rebuilding parts of the B-tree. Also, if you do a lot of insertions and deletions, the B-tree data structure can become a little inefficient. Running the `OPTIMIZE TABLE` command will rebuild the B-tree to make it more efficient.

As a final note, MySQL has fulltext indexes, which speed up searches of string data types when we're looking to see if they contain specific frequently requested substrings.

13 A little on database design

Here we'll cover a little about how to break a database up into tables and how to design the columns of each table. Time spent on designing a database can save a lot of trouble down the road.

Example 1 Let's look at a simple example of a database storing course information for a school. To keep it simple, suppose the `courses` table just keeps track of the course name, start time, and room. That room might be something like "Academic Center 203". This contains both a building number and a room number. It's often better to put a multipart column like this into its own separate table. This is how it would look.

```
CREATE TABLE rooms(
    room_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    building VARCHAR(100),
    room_number VARCHAR(100)
);

CREATE TABLE courses(
    course_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    start_time TIME,
    room_id INTEGER,
    FOREIGN KEY (room_id) REFERENCES rooms(room_id)
);
```

The last two lines of the `courses` table tie the `courses` and `rooms` tables together. The `FOREIGN KEY` statement is a reference to the primary key of the `rooms` table. It is what relates the two tables to each other. Here is one way to create a room and two courses in that room.

```
INSERT INTO rooms (building, room_number) VALUES ('Academic Center', '212');
INSERT INTO courses (name, start_time, room_id) VALUES
('Calculus', '11:00:00', 1),
('Database', '14:00:00', 1);
```

One weakness of this is that we need to know the ID of the room, but since joins are a key feature of SQL, this is not a problem.

It would be simpler to just have a single `courses` table and have the room name as a column. However, a benefit of breaking things up is we can easily store more info about a room. For instance, we might also want to store info on which campus the room is at, what its seating capacity is, etc. Another benefit is that when there is a room table with all the columns separate from each other, we can easily run queries to find all the classes in a specific room or building. When the rooms are just names like "Academic Center 212" or "Science 120B', we end up having to do some potentially tricky string operations to do those types of queries.

Example 2 Suppose we have a table for students, and we want to be able to track which courses a student is taking. When I was first learning database stuff, I wanted to do this by having a list of courses as part of the students table. However, that is just not how things are done in SQL. Instead, we make separate `students` and `courses` tables, and then a new table called `student_courses` that ties the two tables together. This is called an *intersection table* or *junction table*. Here is what this might look like.

```
CREATE TABLE students(
    student_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    student_name VARCHAR(100)
);

CREATE TABLE courses(
```

```
    course_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    course_name VARCHAR(100)
);

CREATE TABLE student_courses(
    student_id INTEGER,
    course_id INTEGER,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES students(student_id),
    FOREIGN KEY (course_id) REFERENCES courses(course_id)
);
```

Note that we've kept the students and courses tables very simple, though in the real world they would each have many more columns. The intersection table consists of the IDs from both of the individual tables and its primary key is a combination of the two. Both of those keys are used as foreign keys into the intersection table.

Example 3 Suppose we want to track attendance of students in courses. We want to be able to answer questions such as which students were absent from a particular course on April 4, how many sophomores missed more than 5 classes, which computer science course had the best attendance, etc. We can accomplish this by using three tables: a table for students that will keep info about the students (id, name, grade level etc.); a table for info about courses (id, title, department, etc.); and a table to track attendance. This last table will have a student ID and a course ID that are foreign keys into the respective tables. It will also have a date field for which day of class we are looking at, and it will have a field to track if the student was there or not. Here is SQL code to create those tables.

```
CREATE TABLE students (
    student_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    grade_level ENUM('Freshman', 'Sophomore', 'Junior', 'Senior')
);

CREATE TABLE courses (
    course_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(100),
    department VARCHAR(100)
);

CREATE TABLE attendance (
    student_id INTEGER,
    course_id INTEGER,
    class_date DATE,
    present BOOLEAN,
    PRIMARY KEY (student_id, course_id, class_date),
    FOREIGN KEY (student_id) REFERENCES students(student_id),
    FOREIGN KEY (course_id) REFERENCES courses(course_id)
);
```

Relationships and Entity relationship diagrams

When tables relate to each other, there are three typical types of relationships: one-to-one, one-to-many, and many-to-many. Here is an example.

One example is a database with an employees table and an offices table, related by an `office_id` foreign key in the employees table. If we assume that each employee has only one office and each office holds only one person, then it's a one-to-one relationship. On the other hand, if several employees are packed into an office, with each employee in only one office, then it's a one-to-many relationship. Finally, if we can pack several employees in an office and some employees can have multiple offices, then it's a many-to-many relationship.

In general, a one-to-one relationship between tables A and B is where each row in A is related to only one row in B and vice-versa. In a one-to-many relationship, each row in A is related to potentially multiple rows in B, while each row of B is only related to one row of A. In a many-to-many relationship, each row of A is potentially related to multiple rows of B and each row of B is potentially related to multiple rows of A.

In a many-to-many relationship, usually a third table, a junction table, is used to see the full relationship.

Here are a few examples:

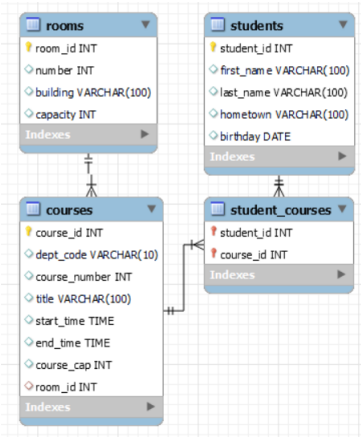
1. Table of states; table of state capitals: This is a one-to-one relationship as each state has only one capital

and each capital is the capital of only one state.

2. Table of professors; table of courses. If we ignore co-taught courses, then this is a one-to-many relationship as each professor will teach multiple courses, but courses will have only one professor.

3. Table of students; table of courses. This is a many-to-many relationship as each student can be taking multiple courses and each course can have multiple students. This many-to-many relationship would typically be implemented using a student-courses junction table.

When building complicated databases, people often use an *entity relationship diagram* (ERD) to graphically show the tables, their columns, and how the tables relate to each other. Below is an example generated by MySQL (using database → reverse engineer) for the student-courses example given earlier.



The arrows between tables indicate foreign-key relationships. The exact shape on the arrows represents the type of relationship (one-to-one, one-to-many, or many-to-many). We won't cover any more about ERDs here. If you want to know more, many database books go into great detail about them.

Names

Here are a few notes about naming things.

- Names for databases, tables, and columns can contain letters, numbers, and underscores. You can use spaces, but it's best to avoid them.
- It's conventional to use lowercase for database, table, and column names. Probably more people use snake case, like `date_created`, though some people use camel case, like `dateCreated`, or even Pascal case, like `DateCreated`.
- Usually table names are plural (like `CREATE TABLE students` instead of `CREATE TABLE student`).
- Primary key names often have the name of the table in them. For instance, if the table is called `students`, the primary key would be called `student_id`.
- It's considered best to avoid using the same column name in different tables. For instance, instead of having a column called `name` in both the students and courses tables, we could call the student one `student_name` and the course one `course_name`. This helps when using joins.
- It's best to avoid giving things the same name as an SQL keyword. For instance, a table called "table" would be a bad idea. But if you really want to or need to, you can, but to do that, you must surround the name with backticks, like this: `` `table` ``. Backticks are also useful if you want to name something with a space in it.[1]

[1] Some SQL vendors other than MySQL use double quotes or square brackets instead of backticks.

Columns

Here are some notes about designing columns. These are not hard-and-fast rules, but it's usually a good idea to follow them.

- *Avoid multi-valued columns.* For instance, if you wanted to track the courses a student is taking, you could use a column with values like `"CMSCI 254, CMSCI 359, MATH 228"`. However, doing this means you'll need to do string operations to pull apart the column data, and that can sometimes get tricky. Instead, usually people store course information by creating separate tables for students and courses, and use a junction table `student-courses` to track what courses students are taking.

  It's also good to avoid compound-valued columns, like a column for a course code like `CMSCI 359`. It's better to have separate columns for the department code and course number. That makes it easier to sort by course number or code. Similarly, instead of a single column for a full postal address, we could have separate columns for the first line of the address, the second line, the city, the state, and the zip code. That way, if we need to do queries on things such as cities or states, it's much easier than trying to pull apart an entire address string with string operations to search for the city or state.

- *Columns in a table should all be directly relevant to the table.* If not, consider making new tables. For instance, if a courses table had columns for the course professor's first name, last name, and number of credits they are teaching, that info would best be put into a separate table called `professors`, and then a foreign key to that table should be in the courses table.

- *Put some thought into what to use as columns.* For instance, instead of having a column to track someone's age, have a field for their birthday. A simple date calculation can be used to work out their age from their birthday. This avoids having to remember to update ages every year in the database.

  As another example, instead of having a GPA column in a students table, we can use SQL queries to directly compute it. This saves space, and it also avoids us having to remember to update it whenever a new course grade comes in. A downside of this, however, is we may end up wasting more CPU cycles constantly recalculating the GPA whenever someone needs to see it. If GPA is needed often, then it can be more efficient to store it directly, but if it isn't needed very often, then it might be better to compute it when needed since it avoids us having to remember to update it.

- *Avoid repeating identical information in different tables.* For instance, suppose we have tables for courses and faculty offices, with the courses table having a field for the name of the professor teaching it and the offices table having a field for the name of the professor whose office it is. Suppose the professor gets married and changes their name. Then we have to remember to update the name in both places. This is a really easy thing to forget to do, and the result will be inconsistent information in the database. It would be better to have a separate faculty or staff table and then have foreign keys to that in the courses and offices tables.

Choosing keys

The purpose of a primary key is so that every row can be uniquely identified. For instance, if the only columns in a database of employees were name and year started, it's quite possible there would be two John Smiths who both started in 2025. Adding an ID to each row would allow us to distinguish them.

A good key should be unique, immutable (i.e., not changing), and never null. The two options for a primary key are a *natural key* and a *surrogate key*. A natural key is something in the actual data that can serve as a key, while a surrogate key is usually an auto-incrementing integer value we add to each row to ensure uniqueness. In MySQL, we get this by something like `table_id INTEGER PRIMARY KEY AUTO_INCREMENT`.

Some people recommend always using a surrogate key. Others recommend using a natural key when it's possible. A surrogate key will always be unique, immutable, and never null, so you never have to worry about problems with that. The drawback is that it uses extra space.

A good example of a natural key would be a state's two letter abbreviation in a table of information about U.S. states. Another good example would be a book's ISBN number in a table about books. The ISBN is likely to already be in the table, and it satisfies all the properties of a key that we want.

Here are a few examples of things that wouldn't be good keys. First, suppose we have a database of employees. Using names as a key would be a bad idea for a couple of reasons. First, if there are two people with the same name, then the uniqueness property of keys would be broken. Second, people sometimes change their names, especially when getting married. We don't want keys to ever change. As another example, suppose we decide to use U.S. Social Security numbers (SSNs) as keys. These are supposed to be unique and unchanging, but there are a couple of potential problems. One is that not everyone has a SSN. Some foreigners don't. So we could have some null keys. The other issue is about security. People with access to the database would be able to see other people's SSNs.

For some tables, *composite keys* are used. These are keys that are composed of two or more columns. For instance, a database tracking which students are in which classes could use a primary key like (`student_id`, `course_id`) that is composed of foreign keys to a students table and a courses table. As another example, a database that tracks high temperatures in the 50 states by date could use a composite key that is the state's abbreviation and the date.

14 Normalization

An important theoretical topic about database design is *normalization*. It helps give databases a good structure, and it is a way to avoid various problems, called *anomalies*, that can occur from bad database design. Let's look at some of those problems first.

Anomalies

Here are the three types of anomalies:

1. Update anomaly — This happens when the same information is present in multiple parts of a database, and we change the info in one part but not in another. The database then ends up with inconsistent data.

   For instance, suppose we have a table of employee info that has ID, name, etc., and suppose we also have a projects table and one of its fields is the project manager's name. One of the employees gets married and changes her name. If that employee also happens to be a project manager, then we have to remember to change her name in that table as well. If we forget, then there will be different names for the same person in different parts of the database. This problem could be avoided by storing a foreign key to the employee ID instead of the name in the project manager table.

2. Deletion anomaly – This happens when deleting information from a database ends up deleting other related information that we don't want deleted.

   For example, suppose we track student and course info with a single table that has `student_name`, `course_title`, and `prof_title` columns, along with maybe a few others. Everything is in this single table, without separate student and courses tables. Suppose Dr. Smith has only one course that has only one student. That student then drops the course. When we delete that record, Dr. Smith disappears from the database since the only row with him in it was deleted. We could avoid this problem by using separate students, courses, and student-courses tables.

3. Insertion anomaly – This is sort of the inverse of the deletion anomaly. It happens when we want to insert some information in the database, but its structure keeps us from doing so unless we enter in some additional bogus data.

   Sticking with the courses example, suppose Dr. Smith is hired mid-semester. He won't have any courses until the next semester, so he won't show up in the database. We could insert him into the database by creating a bogus course that doesn't actually exist, but that is a bit of a hack. This problem could be avoided, like above, by using separate students, courses, and student-courses tables.

Normal forms

Database normalization is about structuring the database to follow rules for what are called *normal forms*. The most important normal forms are first normal form, second normal form, and third normal form. There are more, like 4th normal form, Boyce-Codd normal form, and others, but they aren't used very often in practice, so we won't cover them. Below are the rules for each normal form.

1. First normal form – A table is in first normal form if every column is single-valued and atomic, and there are no repeated columns.
2. Second normal form — A table is in second normal form if it is in first normal form, and whenever it contains a composite key, every column that is not part of that composite key depends directly on all parts of the key.
3. Third normal form — A table is in third normal form if it is in second normal form and there are no transitive dependencies. A transitive dependency is a situation where A depends on B which in turn depends on C.

First normal form examples The requirement that the columns must be atomic and single-valued means that they can't reasonably be broken up into any smaller units than they are. For instance, suppose for a books table, we have a column called authors that allows multiple authors separated by semicolons for books that have more than one author. That would not be single-valued as we can break that down into separate authors. Another example would be a column for the IDs of courses a person is taking, maybe like `4, 5, 12, 4`. We could break that down into individual courses, so it is not single-valued. Another example would be a column for a person's full name. That would not be atomic as we could break it up into first and last names.[1]

An example of repeated columns would be a table for students that had fields called `major1`, `major2`, and `major3`, in case student are double- or triple-majors. Another example would if that table had `phone1` and `phone2` columns to track a person's different phone numbers. It would be okay, however, if those columns were distinguishable as maybe cell number and work number, or something like that.

The way to fix the various problems keeping something from being in first normal form is usually to create another table or multiple columns. For example, in the problem with books with multiple authors, we could create a new table called authors and another called book-authors. The issue with full names not being atomic could be fixed by creating separate columns for first and last name. In general, if the number of parts of a non-atomic column is fixed and small, new columns are okay. If it's variable, then a new table is best.

Second normal form example Recall that a composite key is one composed of multiple parts. For instance, in an earlier section, we looked at tracking student attendance. The table had a student ID, course ID, and date. Those three values together form a good composite key. For a table to be in second normal form, every field must relate to every part of the key. Suppose this table has fields called `present` (for whether the student was in class) and `class_topic` (for what was covered that day). The `present` field is good since it relates to all three parts of the key – the student, course, and date. The `topic_covered` field, however, is not good, since it only relates to the course and date, but not directly to the student. This keeps it from being in second normal form. To fix the problem, we could create a separate table that tracks the class topics.

Third normal form example Third normal form is about transitive dependence. The idea is that every column of the table should relate directly to the key, not indirectly through some other column. For instance, suppose we have a table called `student_info`. That table has columns called `last_name`, `first_name`, `birthday`, `advisor_name`, and `advisor_office`. Everything is good except the last column. The advisor's office depends on the advisor, not directly on the student. The fix for this would be to create a new table of advisor information, move the advisor name and office info out of the `student_info` table, and replace those with a foreign key `advisor_id` into the new advisor information table.

[1] You could argue that even a first name column is not atomic since you could break it down into individual letters, but that would be taking things to an unreasonable level of atomicity.

Denormalization In general, it's good to try to design databases to satisfy the rules of the first three normal forms. However, doing so often requires creating a bunch of new tables. This then means that queries will involve joining several tables together, which leads to queries that are long and hard to debug as well as slow. So sometimes people *denormalize*, which is to break the rules of the normal forms in some cases in order to get simpler and faster queries. The drawback of this is that you lose some of the benefits of normalization on data consistency.

15 Database types

Other SQL databases

In these notes, we have focused on MySQL. There are several other SQL vendors that are good to know about. Here are the most important vendors as of 2026:

- MySQL — Reasonably fast and powerful free product. It is widely used in the internet. It is probably the easiest database for beginners to use.
- PostgreSQL — Like MySQL, it is free and widely used. It's a little less beginner-friendly than MySQL, but also has a few more advanced features that MySQL doesn't have.
- Oracle — Oracle is the industry standard. It is the fastest and most powerful SQL database. It is also the most expensive.
- Microsoft SQL Sever — This is Microsoft's SQL offering. It is fast and powerful, though not quite as good as Oracle. It is not quite as expensive. It is a good choice if you want easy integration of a database with other Microsoft products.
- SQLite — Simple, free database that is built into Python, Android, and other systems. It won't work well if you have a lot of people trying to write to the database at the same time, but is a good choice if you want a simple database to use on a single-user application. For instance, web browsers use it to store history and bookmarks.

  Here is a quick demo of some SQLite code you can run in Python right now without having to install anything.

```
import sqlite3

conn = sqlite3.connect("example.db")
cursor = conn.cursor()
cursor.execute('''
    CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        age INTEGER
    )''')

cursor.execute('INSERT INTO users (name, age) VALUES (?, ?)', ('Alice', 30))
conn.commit()

cursor.execute("SELECT id, name, age FROM users")
rows = cursor.fetchall()
for row in rows:
    print(f'ID: {row[0]}, Name: {row[1]}, Age: {row[2]}')

conn.close()
```

NoSQL databases

There are many categories of databases. SQL is the main language used for the category of *relational databases*. For several decades, relational databases with SQL made up the vast majority of databases in use. But by the 2000s, people were finding SQL and the relational model hard to work with for certain types of websites and

certain types of datasets. People started referring to other databases that worked better for these use cases as NoSQL.

Here are places where a NoSQL database might work better than a traditional relational SQL database.

1. The schema in SQL is fairly rigid. When you create a database, you define what the columns and their datatypes in each table are. This is the schema. This is great for data consistency, and you can put in all sorts of constraints to minimize the chance of errors. However, the rigid schema means you have to put a lot of work in up front, and it can make it painful if you are continually making changes to the schema. This often requires taking the database down and migrating over to the new one.

   For some projects, requirements change frequently. Also, some people work better with a more agile approach where they develop the structure of the database as the system develops. Some NoSQL databases have this flexibility in their schema.

2. One issue with SQL relates to what are called horizontal and vertical scaling. These are about what you do if you need more resources for your database. Vertical scaling is where you add more resources to your database server, like more storage, more RAM, etc. Horizontal scaling is where you increase your resources by adding more servers. SQL databases are easy to scale vertically, but often not so easy to scale horizontally. They can scale horizontally via a process called sharding, but it can be tricky. There are various NoSQL databases that are designed to scale horizontally more easily. Also, some NoSQL databases work better in cloud architectures than SQL.

3. Some NoSQL databases are optimized for tasks that are common on the internet, like real-time analytics and faster for systems that need to do a ton of reads and writes. They sacrifice some of the data consistency features of SQL for faster read/write time.

4. Some NoSQL databases are better than SQL at handling certain kinds of really huge data sets.

The major categories of NoSQL databases are below.

- Key-value stores — These are a simple idea. Data is stored in a key-value format similar to dictionaries/maps in programming languages. They are useful when a full SQL database would be overkill. They are good for simple and fast data storage, like for session IDs, user preferences, and shopping carts. The most popular one is Redis.
- Document-based databases — These are like key-value stores except that instead of storing a simple value, you store an entire “document”, which allows for more complex storage and queries. The documents can have schemas and are often stored in a JSON format. These have similar use cases to SQL, but with a more flexible schema and are faster at some use cases. The most popular one is MongoDB.
- Columnar databases — Instead of storing data in rows, like relational databases, these store data by columns. These are good for time-series data and logging.
- Graph databases — These store data in a graph data structure. They are appropriate for studying connections and relationships between different entities, such as in a social network of people. The most popular is Neo4j.

OLTP vs OLAP

Two common types of databases are *online transaction processing* (OLTP) and *online analytical processing* (OLAP). OLTP is the more common one. The key word in its name is *transaction*, which here refers to common tasks like adding or removing things from the database or updating things. OLTP databases are good for things like managing inventory systems, managing customer orders, basic accounting tacks, etc.

In OLAP, the key word is *analytical*. These are databases where we are interested in learning about or analyzing a data set to look for patterns. An OLAP database is something a data scientist would use, while OLTP is more

for day-to-day tasks of managing a business. OLAP databases tend to be large and not normalized, but instead are designed to give fast results to complex queries. NoSQL databases are often used for these.

As an example, Amazon.com probably uses a OLTP system to handle day-to-day customer orders. These involve simple operations like adding items, changing addresses, etc. They also probably use an OLAP for their vast repository of past orders to learn about customer habits.

16 Database concepts

In this section, we will a few slightly theoretical topics about databases that are good to know.

ACID

ACID stands for *atomicity*, *consistency*, *isolation*, and *durability*. These are four properties that are good for a database to have. Let's look at each one.

Atomicity To say something is *atomic* is to say that it can't be broken up. For databases, atomicity is about a database being able to group several statements into a single unit so that they either all execute or none of them do. For example, suppose we have a database for a bank. When A writes a check to B, we have to remove the money from A's account and add the money to B's account. Suppose we do this like below:

```
UPDATE accounts SET balance = balance - 100 WHERE account_id = 'A';
UPDATE accounts SET balance = balance + 100 WHERE account_id = 'B';
```

What would happen if the database crashed in between the two statements? Then A's account would be down by $100, but B would never get the money. SQL provides a way to group these into a single object, using transactions, like below:

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 'A';
UPDATE accounts SET balance = balance + 100 WHERE account_id = 'B';
COMMIT;
```

The changes don't take effect until we reach the COMMIT instruction, so if a crash happens in between the UPDATE statements, we won't have a problem with missing money. It will be as if neither UPDATE was run.

Consistency When creating a database, we learned that we can apply all sorts of restrictions to our data, such as making sure certain columns are unique, never null, always greater than 0, etc. The consistency part of the ACID acronym refers to how the database will make sure that all the constraints and rules are met after every update, insertion, or deletion.[1]

Isolation Many real-world databases are used by multiple people at the same time. If two users are both working on the same part of a table at the same time, especially if at least one of them is writing to it, we have to be careful that they don't interfere with each other. For instance, if A is writing a bunch of rows of the table while B is reading from it, it's possible that B could read a combination of old and new rows. Or worse yet, if B was also writing, we could get some mixed up combination of both A's and B's writes.

Isolation is about keeping concurrent users isolated from each other. The idea is that if two people are making changes to the database at the same time, the final result should be the same as if one person's updates were followed by another. MySQL uses locks to do this. Locks are a concept from operating systems that provide *mutual exclusion*. That is, once someone gets a lock on a part of the database, they are the only one allowed on that part of the database until they release the lock.

[1] Versions of MySQL before 8.0 would allow you to define CHECK constraints, but it wouldn't actually enforce them. Newer versions are better about this.

In the typical implementation of this, any number of readers are allowed on a system at once. Once someone wants to write to the database, they wait until they are allowed to have a lock. That lock keeps everyone else off the database until the lock is released. Usually MySQL allows you to lock parts of the database, like just specific rows, tables, or memory pages. Having a lock over the entire database would be inefficient.

With locks comes the possibility of *deadlocks*. One deadlock scenario is where one person has lock A and wants lock B, while another has lock B and wants lock A. Each has the lock the other one wants and won't give up the lock they have. In this case, the system gets stuck (or deadlocked). The fix is that either A or B has to be stopped, which is unfortunate. This is not a very common occurrence, but it can happen.

Durability This is where once we write something, we can be sure that it's in the database permanently, even if the system crashes. MySQL implements this via a logging system. Before it does anything, it writes what it is about to do in a log file. If the system crashes midway through the operations, when it comes back up again, it reads the log file and can replay the operations.

CAP Theorem

The C, A, and P in the CAP theorem are *consistency*, *availability*, and *partitition tolerance*. The theorem applies to databases spread across several servers with multiple concurrent users. Before stating it, let's look at what the three terms mean.

Consistency is about how the database should return correct and up-to-date information. For instance, if B makes a change to the database on one server and A then reads those same entries on another server, A should see the updated data.

Availability is about how the database should be responsive to queries. People should get responses to their queries, and they shouldn't have to wait a long time to get them.

A *partition* is a communications break in the network, where two servers in the network are unable to communicate. Partition tolerance means that the system should still work even if there is a partition.

The CAP theorem says that you can't have all three of these desirable properties. The best you can have is two of them. The three possible groups of two are CP, AP, and CA.

In a CP system, we have partition tolerance and consistency, but we lose availability. That is, if there is a communications breakdown, we can handle it. But since we are prioritizing consistency over availability, we are saying if we can't give an up-to-date answer, then we won't give any answer at all. In a CP system, correctness is more important than responsiveness. People using the system will get an error message saying the system isn't available. An example of a CP system would be something to do with a bank. We don't want to give people incorrect information about how much money they have, so it's better in this case not to give an answer at all.

In an AP system, we have partition tolerance and availability, but we lose consistency. As above, we can handle communications breakdowns. But now, we are prioritizing availability over consistency. That is, we are saying it is important that people get an answer, even if it isn't the most up-to-date answer. An example of an AP system would be a site with product reviews. Even if we can't give the last few hours of reviews, it still would be better to give the older reviews than nothing at all.

The last possibility, CA, doesn't really exist in the real world. The problem is that a real network will almost certainly have partitions, and if we don't do anything to deal with them, then we won't have a working system at all. Basically, the CAP theorem says that when a partition happens, you have to choose between consistency and availability, and you can't have both.

17 Data formats

This section is about some common data storage types that are good to know about.

CSV

CSV stands for comma-separated value. It is a plain-text format that stores data in lines of a text file, with each column separated by a comma. For instance, the first few lines of a CSV storing course info might look like below.[1]

```
CMSCI-358-A,Database, MWF,1-1:50
CMSCI-447-A,Programming Languages,TR,2-2:50
```

CSVs are nice because you can open them with a text editor and read them right away. Microsoft Excel and other spreadsheets can also load them, and you can save spreadsheets as CSVs. Because they have a simple structure, it is easy for programs to read them and parse out the data. For instance, here is some Python code to read a file structured like above and print out just the course title.

```
import csv
for line in csv.reader(open('courses.csv', encoding='utf-8')):
    print(line[1])
```

Each line is treated like a tuple, and we can use `line[0]`, `line[1]`, etc. to get access to the columns.

JSON

JavaScript Object Notation (JSON) is a very common way of transferring data over the internet. It's especially useful for data in which there are several fields. Storing the data in a standard format like JSON allows us to easily parse the data, especially since there are libraries built into most programming languages for that purpose. Here is a sample JSON document:

```
{"talks":[
        {"date":"September 18, 1992",
         "name":"Carrie Oakey ",
         "title":"How to sing badly"},

        {"date":"October 26, 1983",
         "name":"Cory Ander",
         "title":"Cooking with spices"}
]}
```

Below is some Python code that will parse through this for us. Assume that the data is located in a file called `talk_data.json`.

```
import json
parsed = json.loads(open('talk_data.json').read())

for x in parsed['talks']:
    print(x['name'])
```

The first line after the import reads the file into a string, which then is passed into the `json.loads` function. If we already have the JSON in a string, then we wouldn't need the `open` line. The loop reads items in the list `parsed['talks']`. The `talks` part of it comes from the fact that the JSON data we are looking at contains a list called `talks`. We can then access individual fields in each item using things like `x['name']`, `x['date']`, etc.

XML

Extensible Markup Language (XML) is another way of structuring data. It plays a similar role to JSON. Some internet services use XML, while others use JSON. Both are pretty common. XML's syntax was inspired by HTML. Various fields are started with a tag, like `<date>` in the example below and the end of the field is marked by a corresponding closing tag, like `</date>` in the example below.

[1] If a column contains a comma, then it is put inside double quotes, and if it contains quotes, then they are escaped by doubling them.

```
<?xml version="1.0" encoding="UTF-8"?>
<talk_information>
    <talk>
        <date>September 18, 1992</date>
        <name>Carrie Oakey</name>
        <title>How to sing badly</title>
    </talk>

    <talk>
        <date>October 26, 1983</date>
        <name>Cory Ander</name>
        <title>Cooking with spices</title>
    </talk>
</talk_information>
```

Here is some Python code to parse this XML.

```
import xml.etree.ElementTree as ET
tree = ET.parse('smalltalk.xml')

for x in tree.getroot():
    print(x.find('name').text)
```

Notice that there are some differences to how the XML parser works versus how the JSON parser works. But the overall approach is similar.

18 Redis

Redis (pronounced with a short i) is a key-value store. The name is short for remote dictionary server. It is designed for speed.

To test it out, you can sign up for a free account that has a limited amount of storage. We will be using the `redis` package for Python. To install it, use `pip install redis` at the command line. Below is code that sets things up. We just have placeholders for the host, port, and password (the actual info comes from your Redis account).

```
import redis
r = redis.Redis(
  host = 'www.example.com',
  port = 12345,
  password = 'password')
```

Storing keys Here is a command that stores a key:

```
r.set('first_message', 'hello')
```

If you're running Redis locally, this will store it on your local machine. If you're using a cloud service, then this is stored on a server somewhere on the internet. To read the key, use this:

```
print(r.get('first_message'))
```

Note that `r.get` returns a Python bytes object. If you want an actual string, decode it by doing `r.get('first_message').decode()`.

Expiring keys You can set a key to expire after a certain time. Below are some examples

```
r.set('some_key', 'hello!')    # create the key
r.expire('some_key', 15)       # set key to expire in 15 seconds
r.expireat('some_key', datetime(2024, 4, 3, 18, 1))
r.ttl('some_key')              # tells how long until the key expires
```

The third line sets it to expire at 6:01 pm on April 4, 2024. You would need to do `from datetime import datetime` to do it.

Key management Here are a couple of useful commands:

```
r.delete('keyname')  # deletes a key
r.exists('keyname')  # check if a key exists
r.keys('*')          # show all the keys.
r.keys('ab*')        # show all the keys that start with ab.
```

Counting It's common to use Redis to count things. Here are a few basics to working with them.

```
r.set('counter', 1)
r.incr('counter')
r.incrby('counter', 10)
```

Maps Use a map/dictionary to store multiple fields. Here is an example to show how to set and get a map.

```
r.hset('mapname', mapping={'name':'smith', 'age':107})
r.hgetall('mapname')
```

There are a variety of other methods for working with maps. See an online reference for details.

Lists Redis has a list type. Here are a few operations

```
r.rpush('listname', 'hello')    # adds 'hello' to end of the list
r.rpop('listname')              # pulls off the last item and returns it
r.lindex('listname', 0)         # gets the item at index 0
r.lrange('listname', 0, -1)     # shows everything in the list
r.llen('listname')              # shows the length of the list
```

Note that the first time you call `push`, it will automatically create the list. All of these methods can start with either `r` or `l` for "right" or "left", depending on which side of the list you want to work with. See a good online reference for more list methods.

Sets Redis also can also store sets, which are like lists, but where we don't care about order. They are much faster than lists if you just care about keeping track of whether things are or are not in a collection. Here are some things we can do with sets. See an online reference for more.

```
r.sadd('setname', 'hello')        # adds 'hello' to the set
r.smembers('setname')             # lists all the items in the set
r.sismember('setname', 'hello')   # tells if 'hello' is in the set
r.scard('setname')                # number of items in the set
```

Sorted sets Redis has a sorted set type. When storing items in the set, we also store a score with them. Here are some examples:

```
r.zadd('zsetname', {'hey': 2, 'hello': 5})    # store 2 items
r.zrange('zsetname', 0, -1, withscores=True)  # show whole set, sorted by score
r.zrank('zsetname', 'hey')                    # show where 'hey' is in the ranking
```

Publish/Subscribe Redis provides a publish/subscribe feature where you can subscribe to a channel and be notified whenever there are updates. To publish an update, a line like below is used.

```
r.publish('pubsub_name', 'hello')
```

To listen for updates, first subscribe, and then use the `listen` method, like below.

```
pubsub = r.pubsub()
pubsub.subscribe('pubsub_name')

for message in pubsub.listen():
    if message['type'] == 'message':
        print(message['data'])
```

The `listen` method blocks, sort of like Python's `input` function, until it gets an input, so your application will essentially sleep until it receives a message. You can put this code into a thread if you want your application to be doing other things while it's waiting for updates.

19 MongoDB Introduction

MongoDB is a popular document-oriented database. People often use it instead of SQL-based (relational) databases because of its flexible schema, because of its very fast read/write times, and because it is easier to scale horizontally than SQL. That is, it's easier to improve performance with MongoDB by adding servers than it is for SQL-based databases. The flexible schema is useful for situations where a rigid schema, like is required in SQL, would be too difficult. One downside of Mongo compared to SQL is that its query language is nowhere near as flexible or easy to work with as SQL. SQL is also better for data consistency.

Installation

The free MongoDB community server (free) is at `https://www.mongodb.com/try/download/community`. The default installation options are fine. After it is installed, the MongoDB server should be up and running at port 27017.

A simple way to work with MongoDB is through the shell, which is at `https://www.mongodb.com/docs/mongodb-shell/`. The shell comes in a zip file. Unzip it in a convenient directory. To run it, navigate at the command line to the directory it's installed in. Then navigate to the `/bin` subdirectory, and run it via `mongosh` or `./mongosh`.

The notes here are based on `https://www.mongodbtutorial.org`, which is a nice tutorial if you want something a little more in depth than these notes.

A little about MongoDB

MongoDB works with JSON data. It stores data on disk in BSON (binary JSON) format. Data is stored in *documents*, which are sort of like rows in SQL. Documents are grouped into *collections*, which are like tables in SQL. Tables are grouped into *databases*, which are like databases in SQL.

A few useful commands at the shell

Here are a few useful commands.

- `show dbs` — lists the databases
- `show collections` — lists the collections
- `use someDatabase` — switches to a database called `someDatabase`. If the database doesn't exist, this command will create it. After calling this, the `db` variable will contain the name of the current database.
- `console.clear()` – clears the screen

The shell is actually a full-featured JavaScript shell, so you can run any JavaScript code in it.

To try things out, trying running the following sequence of commands (the `>` character is not meant to be typed. It just indicates the shell prompt).

```
> use test
> db.createCollection("books")
> db.books.insertOne({title:"Database", author:"Smith"})
> db.books.find()
```

This switches to (and creates) a database called `test`. It then creates a collection (table) called `books`. We then insert info for one book into the database. Notice that the data is in a JSON format. The last command shows the contents of the collection.

Data types

Here is a brief introduction to the most common data types.

- Numbers — Whole numbers will default to a 32-bit integer. Decimal numbers default to doubles. For instance, `numPages: 140` will store 140 as a 32-bit integer.
- 64-bit integers — If you need to store values larger than around 2 billion, use a long, which can handle up to around 10^{18}. The syntax looks like this `numPages: NumberLong("533409210221")` to store a ridiculous number of pages.
- Strings — Just use quotes (either single or double, but double are preferred). Strings can hold any UTF-8 characters.
- Boolean — These hold true/false values
- Dates — To create a field with the current date, use `new Date()`. Make sure to use `new` since without it, Mongo will just store a string representation of the date instead of a date object. Dates are stored in Unix time, as the number of milliseconds since January 1, 1970. To create a specific date, use something like `ISODate("2024-03-25")`. This is in UTC time. There are things you can do if you want to convert it to a specific timezone.
- Arrays — These are indicated with square brackets. You can mix data types inside Mongo's arrays.
- Embedded documents — You can embed or nest documents inside another. for instance, we could have something like this: `{title: "Databases", author: {first: "John", last:"Smith"}}`.
- ObjectId — Every document has an ID in a field called `_id`. Mongo will automatically create one, but you can specify it yourself if needed.

Adding things to collections

There are two main ways to add things: `insertOne` and `insertMany`. Here is one example, demonstrating the data types given above.

```
db.books.insertOne({
    title: "Databases",
    author: {first: "John", last:"Smith"},
    numPages: 400,
    price: 29.99,
    ebook: false,
    pubDate: ISODate("2024-03-25"),
    recordCreation: new Date(),
    subjectAreas: ["Computers", "Databases"]
})
```

Note that the command is `db.books.insertOne`. Here `db` is a variable holding the currently selected database (selected by the `use` command) and `books` is the name of the collection we are inserting into. Note also that field names don't need quotes unless they have spaces or certain special characters.

If you want to insert more than one thing at a time, use `insertMany`. Here is a short example:

```
db.books.insertMany([
{title: "Databases", author:"Smith"},
{title: "SQL", author: "Thomas", numPages: 300}
])
```

The documents to be inserted are put into an array. Note that in MongoDB, since schemas are flexible, not all documents need to have the same fields.

Queries

The two commands or queries are `findOne`, which returns just one result, and `find`, which returns all the results. Both work similarly. For the examples in this sections, suppose we have a collection keeping track of information about students in a college. The `find` method takes two arguments. The first parameter is the query that tells what to search for, and the second parameter specifies which fields to return. Here are two examples of the query parameter:

1. `db.students.find({hometown: "Boston"})` — Finds all the students whose hometown is Boston.
2. `db.students.find({hometown: "Boston", credits:100})` — Finds all the students whose hometown is Boston who have taken exactly 100 credits.
3. `db.students.find({"name.first": "alice"})` — When searching an embedded field, use dot notation like this. Quotes are needed around the field name when searching an embedded field.

Here are examples of the second parameter. All of them find the students whose hometown is Boston.

1. `db.students.find({hometown: "Boston"}, {name: 1})` — Just display the names (and IDs)
2. `db.students.find({hometown: "Boston"}, {name: 1, credits: 1})` — Just display names, credits (and IDs)
3. `db.students.find({hometown: "Boston"}, {name: 1, _id: 0})` — The ID is always displayed unless we tell it not to, like here.
4. `db.students.find({hometown: "Boston"}, {name: 0})` — Displays all the fields except the name.

There are various operators for doing comparisons. They are `$eq`, `$ne`, `$gt`, `$gte`, `$lt`, and `$lte`, for equals, not equals, greater than, greater than or equal to, less than, and less than or equal to, respectively. The `$eq` operator is the default, so it isn't usually necessary to specify it. Here are examples of some of the others.

1. `db.students.find({credits: {$lt:100}})` — Finds all students that have taken less than 100 credits
2. `db.students.find({credits: {$gte:100}, birthday: {$lte:ISODate("1999-12-31")}})` — Finds all students taking at least 100 credits who were born before the year 2000.

Note that the syntax takes a little getting used to. There are several other useful operators.

- `$in` and `$nin` — These are used to check if something is or isn't in a list. For instance, `db.students.find({credits:{$in: [95,100]}})` finds all students who have taken either 95 or 100 credits.
- `$and, $or, $not` — These are logical operators. If we just have multiple fields separated by commas, then an AND operation is assumed, so we don't need the operator. The operations being OR-ed are grouped in an array. For an OR operation, we can do something like the following, which finds people who took 20 credits or who were born in 2001 or later.

  ```
  db.students.find({$or: [{credits:20}, {birthday: {$gt:ISODate("2024-01-01")}}]})
  ```

- `exists` — This tells if a document contains a field. This is useful since the schema in Mongo can be variable, so not all documents may contain a certain field. A simple example is `db.students.find({birthday: {$exists: true}})` that returns all the students that have a birthday field.
- Regular expressions — We can use a JavaScript regular expression to do more sophisticated searches. For instance, `db.students.find({name:/^A[a-z]*/})` finds all students whose name starts with a capital A.

- Sorting — If you want to sort the results, append the `sort` method. For example, each of the queries below returns students whose hometown is Boston, sorted in different ways.

  1. `db.students.find({hometown: "Boston"}).sort({credits: 1})` — Sort by credits
  2. `db.students.find({hometown: "Boston"}).sort({credits: -1})` — Sort by credits in reverse order
  3. `db.students.find({hometown: "Boston"}).sort({credits: 1, birthday: 1})` — Sort by credits first, then by birthday.

- Limits and skips — We can also append methods to limit the number of results or to start at a certain point in the results. Here are a few examples.

  1. `db.students.find({hometown: "Boston"}).limit(1)` — Just return 1 result
  2. `db.students.find({hometown: "Boston"}).skip(2)` — Return all the results except the first 2
  3. `db.students.find({hometown: "Boston"}).sort({credits: 1}).limit(10).skip(3)` — Return all the results sorted by credit, limited to 10, and starting with the 4th record.

- Arrays — There are various operators for working with arrays. Two useful ones are `$size` and `$elemMatch`. Here are some examples:

  1. `db.students.find({classList: {$size: 5}})` — Assuming `classList` is a list of classes a student is taking, this query finds all students taking exactly 5 classes.
  2. `db.students.find({scores: {$elemMatch: {$eq: 100}}})` — Assuming `scores` is a list of test scores for the student, this determines if the list contains the values 100.
  3. `db.students.find({scores: {$elemMatch: {$in: [95, 100]}}})` — Determines if the list contains 95 or 100.
  4. `db.students.find({scores: {$elemMatch: {$gt:90, $lt:100}}})` — Determines if the list has a value between 90 and 100.

Updating and deleting

The `updateOne` and `updateMany` methods change fields in a document. Here are two examples:

```
db.books.updateOne({_id: ObjectId("670dc8324131892a60a44a5d")}, {$set:{z:9}})
db.books.updateMany({credits: {$gt:50}}, {$set:{grad: true}})
```

The first sets a field called `z` to 9 in an object with a particular ID. If the `z` field doesn't exist, it will be created. The second updates all the documents where the credits field is at least 50. To remove a field from a document, use `$unset`, like below, which removes the `z` field from the document with ID 2.

```
db.books.updateOne({_id: 2}, {$unset:{z: ""}})
```

The `$inc` and `$mul` operators can be used to update a numerical value. The first line below adds 3 to `x`, the second decreases it by 1, and the third multiplies it by 2.

```
db.books.updateOne({_id: 2}, {$inc:{x: 3}})
db.books.updateOne({_id: 2}, {$inc:{x: -1}})
db.books.updateOne({_id: 2}, {$mul:{x: 2}})
```

The `$push` operator is useful for updating an array field. The example below adds 99 to the end of an array.

```
db.books.updateOne({_id: 2}, {$push: {array: 99}})
```

To delete a document, use `deleteOne` or `deleteMany`. Here are some examples:

```
db.students.deleteOne({_id: 2})
db.students.deleteMany({credits: {$gt:50}})
db.students.deleteMany({})
```

The first deletes a single record, matching a specific ID. The second deletes all records of students with more than 50 credits. The third deletes everything in the table.

Aggregating

Mongo provides a useful function called `aggregate`. It allows us to chain a bunch of operations together in a pipeline. The result of the first operation is passed to the second. The result of that is passed to the third, etc. The key operations we can do are listed below.

- `$match` — Used to filter down the results according to various conditions
- `$group` — Used to group results into categories, similar to SQL's `GROUP BY`
- `$project` — Used to determine which fields to include in the results and which ones not to
- `$limit` — Limit the number of results
- `$skip` — Skip the first few results
- `$sort` — Put things in order
- `$unwind` — Breaks down an array field into individual documents
- `$lookup` — Used to access other collections, Like an SQL join

For the examples here, assume we have data on students with their birthday, hometown, year (freshman, sophomore, etc.), and list of classes.

1. The example below first restricts the query to all students who are born in 1999 or later and then it lists all the hometowns and the total number of credits taken by students from that town. Finally, it sorts by that total.

   ```
   db.students.aggregate([
    {$match: {birthday: {$gte: ISODate("1999-01-01")}}},
    {$group: {_id: "$hometown", total:{$sum: "$credits"}}},
    {$sort: {total: 1}}])
   ```

   This is like the SQL query below.

   ```
   SELECT hometown AS `_id`, SUM(credits) AS total
   FROM students
   WHERE YEAR(birthday) >= 1999
   GROUP BY hometown
   ORDER BY SUM(credits)
   ```

   Note that in the Mongo query, `_id` and `total` are names we give to the fields. They act similarly to aliases in SQL.

2. Here is another example, where we add a second match to restrict it to just show hometowns whose total credits are over 100. This is like using `HAVING` in an SQL `GROUP BY` query.

   ```
   db.students.aggregate([
    {$match: {birthday:{$lt:ISODate("1999-12-31")}}},
    {$group: {_id: "$hometown", total:{$sum: "$credits"}}},
    {$match: {total: {$gt:100}}},
    {$sort: {total:1}}])
   ```

3. Here is a shorter example that lists the hometowns and how many students are from each, in decreasing order.

```
db.students.aggregate([
 {$group: {_id: "$hometown", total: {$count: {}}}},
 {$sort: {total: -1}}])
```

4. This one finds the student that takes the most credits.

```
db.students.aggregate([
 {$group: {_id: null, total: {$max: "$credits"}}}])
```

5. This also finds the student that takes the most credits, but it uses `$project` to make sure the `_id` field is not displayed in the final result.

```
db.students.aggregate([
 {$group: {_id: null, total: {$max: "$credits"}}},
 {$project: {_id: 0}}])
```

6. In our example collection, we are assuming each student has a list of courses they are taking. The following returns a new collection where the courses are now separated so that if a student named Smith were taking classes 1, 2, and 3, we would get three separate documents.

```
db.students.aggregate([{$unwind: "$classes"}])
```

7. The example below applies `$project` first, so in the three separate documents, we just include the name field and none of the other fields.

```
db.students.aggregate([
 {$project: {name: 1, classes: 1}},
{$unwind: "$classes"}])
```

8. This query builds on the previous to tell how many students are taking each course.

```
db.students.aggregate([
{$project: {name: 1, courses: 1}},
{$unwind: "$classes"},
{$group:{_id:"$classes", total: {$count:{}}}}])
```

9. Suppose that the `classes` field is a list of IDs of classes, and there is a separate `course_info` collection that has information about each course. To join these two tables together, we use `$lookup`. The query below displays each student's info, and in place of the `classes` array, it gets info about each class from the `course_info` collection.

```
db.students.aggregate([
  {$lookup: {from: "course_info",
             localField: "classes",
             foreignField:"_id",
             as: "ClassDetails"}},
  {$project: {classes:0, "ClassDetails._id":0}}])
```

   In the `$project` statement, `"Classes._id"` tells it not to display the ID gotten from the `course_info` table. The quotes are needed there.

Using MongoDB in Python

To use MongoDB in Python, install the `pymongo` package. You can do this via `pip install pymongo` from the command line. Here is a quick demo of how to use PyMongo:

```
from pymongo import MongoClient

client = MongoClient('mongodb://localhost:27017/')
db = client['practice']
collection = db['students']

results = collection.find({'credits':{'$gt':100}})

for doc in results:
    for x in doc:
        print(x, ':', doc[x])
    print()
```

The code connects to the database, assuming it's running on your computer. If it's running remotely, you would want a password as well. The `db = client['practice']` is like `use practice` at the shell, switching to that database. Then we choose which collection we want to work with. After that, `collection.find` can be used to run queries. We can loop over the results and print things out, like shown above. The `collection` object has plenty of other methods, like `insertMany`, `updateOne`, etc., that correspond to the shell commands we saw earlier.