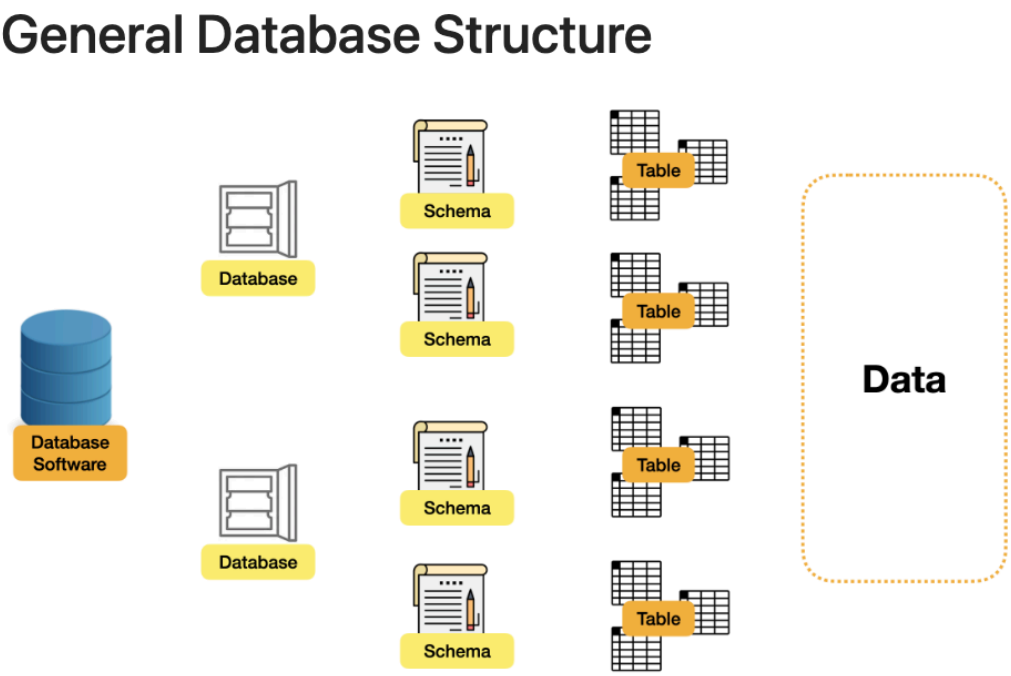
When making SQL queries, divide the work into 2 parts:

1. How to write a query to retrieve data that meets given condition
2. How to display the generated result

High-Level Architecture of Database Software



The following is what is comprised of a database structure:

- **Database Management System (DBMS)**: software used to interact with the actual database
- **Database:** a container within the DBMS that holds related schemas, tables, and data
- **Schemas:** a logical grouping of tables and other database objects (blueprint for how we structure and store data)
    - Some database software combines the schema & database into one entity
- **Tables:** a structured collection of rows and columns (like a spreadsheet), organizing our data
- **Data:** the actual data

Schema Structure & Syntax

When working with a database, we want to create a schema using

The following is an example in SQL:

```
CREATE SCHEMA `new_schema` DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

- CREATE SCHEMA new_schema = creates schema of name `new_schema`
- DEFAULT CHARACTER SET utf8mb4 = uses UTF-8 Unicode Encoding to encode dat
    - *We encode data since we can only store binary, not raw data*
- COLLATE utf8mb4_unicode_ci = uses utf8mb4_unicode_ci collation rules to compare and sort our data

Table Structure & Syntax

Every table lives in a schema

Each table structure includes:

1. Metadata of each column like data type, default value, comments, etc
2. Indices to improve the speed of searching data in the table
3. Relationships between other tables
4. Location of where the data is stored on computer hard drives

id	name	age
1	John	40
2	May	30
3	Tim	25

Types of keys

- **Primary Key**: a column or group of columns that helps uniquely identify each row (one primary key per table)
- **Unique Key:** another column or group of columns that uniquely identifies a row (can have multiple unique keys per table)
    - Used in case we need another way to identify a specific row
- **Foreign Key:** primary keys that belong to another table to create relationships

Columns Structure & Syntax

Each column is responsible for limiting the type & size of values that the data can store

There are 4 major categories of types:

1. Number
    - BIGINT, INT, MEDIUMINT, SMALLINT, TINYINT
      ←Larger Integers, Smaller Integers →
    - DOUBLE, FLOAT, DECIMAL
        - Use DECIMAL for most precision
2. Datatime
    - DATE, MONTH, YEAR
    - DATETIME, TIMESTAMP
        - These types are complete dates with certain formats
            - DATETIME => *8888-01-01 00:00:00*
            - TIMESTAMP => *1970-01-01 00:00:01* or *2038-01-19 03:14:07*
                - Only limited between the 2 values given above
3. Text
    - CHAR, VARCHAR
        - Used to store plain text
        - CHAR => more suitable for data with fixed text length
        - VARCHAR => suitable for data whose length will change
            - VARCHAR(<some_int>) sets the max # of chars that can be stored
    - TEXT, LONGTEXT
        - Used to store text data whose max length is unknown
4. Others
    - BINARY, BLOB
        - Stores files such as images, videos, or large unstructured data
        - Rarely used because it's difficult to manage
    - BOOLEAN
        - Stores 1 or 0 to represent true or false
    - JSON
        - Stores data in JSON format

Column Attribute Functions: settings each column can have to set a specific attribute or constraints to control how our data behaves

- NOT NULL = sets the column so that it must be filled
- DEFAULT = sets the default value if no value is provided on insert
- UNIQUE = sets the column so that allthe values in column are unique
- PRIMARY KEY = sets the column as the unique identifier for each row
- COMMENT = enables the column to have a descriptive note
- COLLATE = controls how text is compared/sorted
- AUTO_INCREMENT = sets the column to generate sequential numbers

Summary of defining a table:

```
CREATE TABLE `<schema>`.`<new_table>` (
`column_name_1` <type1> <column attribute functions separated by space>
<additional parameters for attribute functions>,
`column_name_2` <type2> <column attribute functions separated by space>
<additional parameters for attribute functions>,
more...
PRIMARY KEY (`id`)
);
```

The following is an example of creating a table in SQL:

```
CREATE TABLE 'new_schema`.`new_table` (`id` INT NOT NULL COMMENT
'This is a primary index', PRIMARY KEY (`id`)
);
```

- CREATE TABLE `new_schema`.`new_table`(--other settings); = creates a table called `new_table` in `new_schema`
- `id` INT NOT NULL COMMENT 'This is a primary index' = defines each column of the table
- PRIMARY KEY (`id`) = defines how to set the primary key, which we call `id`

We can destroy a table in SQL as well:

```
DROP TABLE `new_schema`.`.new_table`
```

- This is hardly used since its dangerous to remove everything related to a table

We can clear/delete all data of a table rather than destroy it:

```
TRUNCATE `new_schema`.`.new_table`
```

Tables are NOT static so that we can create & update them:

Creating Column

```
ALTER TABLE `new_schema`.`<table_to_alter>`
ADD COLUMN `<new_column_name>` <column attribute functions
separated by space> AFTER `<column_name>`;
```

Updating Column

```
ALTER TABLE `new_schema`.`<table_to_alter>`
CHANGE COLUMN `<existing_column1>``<new_column1>`
<set_of_new_type&attributes>,
CHANGE COLUMN `<existing_column2>``<new_column2>`
<set_of_new_type&attributes>;
```

NOTE: we creating, adding, updating a column, we always follow a similar format of defining type and attribute functions after defining the column

SQL Syntax

Basic SQL syntax allows us to handle CRUD (Create, Read, Update, Delete) operations

Create data: INSERT

```
INSERT INTO `new_schema`.`<table_name>` (`<column1>`, `<column2>`, `<column3>`) VALUES (<value_for_column1>, value_for_column2, value_for_column3), (<more_records>);
```

- We can choose to add multiple records by adding multiple tuples of values

Read data: SELECT

```
SELECT <column_names_seperated_with_,> FROM `new_schema`.`users`;
```

- This selects the columns of data we want to see from a specified table

```
SELECT * FROM `new_schema`.`<table_name>`;
```

- Selects all rows from the specified table

Update Data: UPDATE

```
UPDATE `new_schema`.`<table_name>` SET `<column1>` = <value>, `<column2>` = <value> WHERE <condition>;
```

- This selects the specific row we want to update and updates it with the supplied values

Remove Data: DELETE

```
DELETE FROM `new_schema`.`users` WHERE `id` = 1;
```

- Removes the specified row from a table

Conditions with the WHERE Keyword

The WHERE keyword is used to query our data

The following are the type of queries we can make:

- **Comparison Queries:**
  - **Equal**
    - `SELECT * FROM `new_schema`.`users` WHERE id = 1;`
  - **Less Than**
    - `SELECT * FROM `new_schema`.`users` WHERE id < 1;`

    - **Greater Than**
        - `SELECT * FROM `new_schema`.`users` WHERE id > 2;`
    - **We can do <=, >=, and != also**
- **Checking For NULL (cannot use =):**
    - `SELECT * FROM `new_schema`.`users` WHERE height IS NULL;`
- **Multiple Conditions:**
    - `SELECT * FROM `new_schema`.`users` WHERE age < 40 AND height > 160;`
        - Uses AND keyword to intersect
    - `SELECT * FROM `new_schema`.`users` WHERE age < 40 OR height > 160;`
        - Uses OR keyword to choose both
    - `SELECT * FROM `new_schema`.`users` WHERE id < 4 AND (age > 30 OR height > 175);`
        - We can use both keywords to make complex queries, use parenthesis to separate logical blocks
        - Operator precedence occurs in SQL
- **Range Conditions:**
    - **Short hand OR and more:**
        - `SELECT * FROM `new_schema`.`users` WHERE `id` IN (1, 3);`
            - The IN keyword can do what OR does but makes queries more readable
            - IN keyword allows use to make more complex queries that OR can't do
            - This example checks for id = 1 OR id = 3
    - **Query based on columns between multiple values:**
        - `SELECT * FROM `new_schema`.`users` WHERE height BETWEEN 160 AND 190;`
            - BETWEEN must be used with AND => we can use BETWEEN to query a range between LEFT AND RIGHT
    - **Query text based columns using reg expressions:**
        - `SELECT * FROM `new_schema`.`users` WHERE name LIKE '%a%';`

Handling JSON in SQL

Reading JSON:

```
SELECT `id`, JSON_EXTRACT(contact, '$.phone') AS phone
```

```
FROM `new_schema`.`users`;
```

- Note: the value will be wrapped in double quotes
  We can also remove the double quotes that come with the JSON values:
  ```
  SELECT `id`, JSON_UNQUOTE(JSON_EXTRACT(contact, '$.phone')) AS phone
  ```

```
FROM `new_schema`.`users`;
```

Querying JSON

```
SELECT `id`, JSON_UNQUOTE(JSON_EXTRACT(contact, '$.phone')) AS phone
FROM `new_schema`.`users`

WHERE JSON_EXTRACT(contact, '$.phone') like '%456%';
```

Adding Data

```
INSERT INTO `new_schema`.`users` (`id`, `name`, `contact`)
VALUES (5, 'Harry', JSON_OBJECT('phone', '1231123', 'address', 'Miami'));
```

Updating Data

```
UPDATE `new_schema`.`users` SET `contact` = JSON_SET(contact, '$.phone', '6666', '$.phone_2', '888') WHERE `id` = 2;
```

Auxiliary SELECT Statements

The following are extra keywords we can add to our queries:

- **Uniqueness: DISTINCT**
  - We can find distinct values of a column
  - `SELECT DISTINCT age FROM `new_schema`.`users`;`
- **Pagination: LIMIT & OFFSET**
  - We can limit how many items we display and also skip the first specified # of items when querying
  - `SELECT * FROM `new_schema`.`users` LIMIT 3 OFFSET 1;`
  - To add pagination: split data into chunks we can use the 2 key words together:
    - `SELECT * FROM `new_schema`.`users` LIMIT 3 OFFSET 0;`
    - `SELECT * FROM `new_schema`.`users` LIMIT 3 OFFSET 3;`

        - `SELECT * FROM `new_schema`.`users` LIMIT 3 OFFSET 6;`
        - Each page holds 3 rows of data
- **Sorting: ORDER**
    - We can choose if we want to order our rows based on a certain column
    - `SELECT * FROM `new_schema`.`users` ORDER BY age;`
        - By default we sort in ascending order
    - `SELECT * FROM `new_schema`.`users` ORDER BY age DESC;`
        - We can sort in descending order
    - We can also sort based on multiple columns
        - `SELECT * FROM `new_schema`.`users` ORDER BY age DESC, height DESC;`
            - NOTE: if the 1st column has equal values, we sort based on 2nd value, and so on
- **Grouping: GROUP BY**
    - We can group based on column
    - `SELECT `age` FROM `new_schema`.`users` GROUP BY age;`
        - This query outputs a similar query to DISTINCT
        - What really happens is that we are looking at grouped up rows and we can apply operations on those grouped rows
    - GROUP BY supports aggregate functions, which is why we use this as opposed to DISTINCT

Aggregate Functions

There are a lot of SQL functions we can apply on our data, pretty much any keyword that requires us to feed arguments is a function

Aggregate Functions: SQL functions which take multiple rows of input & returns a single value as output

- **Counting: COUNT**
    - `SELECT COUNT(*) AS `user_count` FROM `new_schema`.`users` WHERE id > 1;`
- **Total: SUM**
    - `SELECT SUM(`age`) AS `sum_of_user_ages` FROM `new_schema`.`users`;`
- **Average: AVG**
    - `SELECT AVG(`height`) AS `avg_user_height` FROM `new_schema`.`users`;`
- **Minimum & Maximum: MIN & MAX**
    - `SELECT MIN(`height`) AS `user_min` FROM `new_schema`.`users`;`

    - `SELECT MAX(`height`) AS `user_max` FROM `new_schema`.`users`;`
- There are more…

Relationships

Different tables may be related to each other through: 1:1, 1:many, many:many

1:1 Relationship

Each row in Table A is linked to exactly 1 row in Table B

- Object A owns Object B
- Object B is owned by Object A
- Ex: 1 transportation, 1 order

1:Many Relationship

A row in Table A can be linked to many rows in Table B

- Object A owns multiple Object B
- Each Object B is owned by a single Object A
- Ex: 1 user, many orders

Many:Many Relationship

A row in Table A can be related to many rows in Table B, and vice versa
This relationship requires a 3rd Junction table to link:

- Ex: 1 product, many orders | Many products, 1 order

Joins

We can combine tables in our queries

LEFT JOIN

- We a treat a table as the “main” table (left), and the other table as an “attached” table
    - We are adding data from the attached table to our main table
    - `SELECT * FROM `new_schema`.`users` LEFT JOIN `new_schema`.`orders` ON `users`.`id` = `orders`.`user_id`;`

        - NOTE: **LEFT JOIN** <attatched table> **ON** <main table> <condition on joining>
            - This example joins rows where users.id with orders.user_id

RIGHT JOIN

- The join is the exact same, except the right side is the “main” table while the left is the “attactched” table
- The choice of LEFT vs RIGHT is for readability (does it make more sense for the main stuff to be left or right)

NOTE: these joins will still include unmatched rows and replace the columns with NULL

INNER JOIN

- Basically a LEFT or RIGHT JOIN that excludes NULL values

Aliases

Aliases are useful for a variety of things:

- *When joining, we many have tables with the same column labels, so we use aliases so we can represent columns without repeated names*
- ```
  SELECT `orders`.`id` AS order_id , `name` FROM `new_schema`.`users`

  INNER JOIN `new_schema`.`orders` ON `users`.`id` = `orders`.`user_id`;
  ```

- *Useful for shortening names*
- *Useful for self joins*
- *Useful for subqueries*

Subqueries

Subqueries are queries inside another query. We can use 1 query as the input of another

```
SELECT * FROM `new_schema`.`orders`
WHERE user_id = (
  SELECT id FROM `new_schema`.`users`
  WHERE name = 'John'

);
```

- In query 1: We are essentially querying id’s of rows that have John in the name column
- In query 2: We select all orders that match the condition where user_id = query1
```

Common mistakes:

- When you expect a subquery to return multiple values, us **IN** instead of **=**
- Use select properly
    - Use SELECT <specific column> instead of SELECT *, when expecting a single column
    - Just ensure we use right SELECT parameter and don't use * if not needed

Security & Integrity With Foreign Keys

Foreign keys not only connect tables, they enforce data integrity & security

- **You can't insert invalid data where the foreign key references a non-existent row.** Example: You can't add an order with user_id = 999 if no such user exists
- **You can't delete a parent row (e.g. a user) if there are still child rows (e.g. orders) referencing it** — unless you explicitly define what should happen (e.g. ON DELETE CASCADE, SET NULL, or RESTRICT).
- **You can't update the foreign key to a value that doesn't exist in the referenced table.** Example: You can't change orders.user_id to a user_id that doesn't exist in the users table.
- **If you update a referenced primary key, the change won't automatically update foreign keys unless you use ON UPDATE CASCADE.**

The following is some syntax:

- Setting foreign key for a table
- ```
  ALTER TABLE `new_schema`.`orders`
  ADD CONSTRAINT `orders_user_id_key`
  FOREIGN KEY (`user_id`)
  REFERENCES `new_schema`.`users` (`id`);
  ```
    - NOTE:
        - The constraint variable is defined by the **FOREIGN KEY** & **REFERENCES**
        - **ADD CONSTRAINT** is optional when creating a table, but modifying its required

- We can also add modes for certain SQL actions
- ```
  ALTER TABLE `new_schema`.`orders`
  ADD CONSTRAINT `orders_user_id_key`
  FOREIGN KEY (`user_id`)
  REFERENCES `new_schema`.`users` (`id`)
  ON DELETE NO ACTION
  ON UPDATE RESTRICT;
  ```
    - NOTE:

        - We are essentially defining what happens when we use certain CRUD operations on the table

Transactions

This refers to batching multiple SQL operations so they are treated as 1 unit:

- For a transaction to succeed, all operations must succeed
- We only commit a transaction once everything is ok, else we roll back the previous state before the transaction

The following is Syntax to handle transactions:

```
START TRANSACTION;
SELECT `new_schema`.`products` WHERE id = 5;
UPDATE `new_schema`.`products` SET `price` = '500' WHERE id = 5;
IF (@correct) THEN
  COMMIT;
ELSE
  ROLLBACK;
END IF;
```

- Typically, transactions, commits, and rollbacks are handled by other programming languages but we can use pure SQL also

Indices

*This is different from the primary key, when we talk about index we are referring to a specific column that acts as an **INDEX***

Indices is primary used for performance optimization of the data base, they allow us to:

- Directly search for data based on index
- Store data additionally & organize them in some way
- In general, we prevent having to always go through an entire table to find what we want

There are some cons of indices:

- If we have high number of writes, it requires our table to calculate indices which adds to our runtime for writing

How indices work under the hood:

- **In relational databases:** indexes are typically implemented using B-trees (self-balancing tree structures)
    - Then the search process becomes more towards O(log(n))

    - We can technically have O(1) but thats if our table lives in memory (ex: Redis does this)
- **In non-relational databases:** we can also use indices similar to relational databases to accomplish similar runtimes
- **NOTE:** indices help optimize our reads in exchange for longer writes, however, how much it optimizes depends on the data structure used to store our data, type of index, and query pattern

Choosing the right index:

- Technically we can choose any column to be an index, but we want to choose a good index
    - Good indices: columns with types that support =,LIKE, joins, etc
        - Basically columns that are commonly queried and would improve performance

The following is syntax related to it:

- We can add an index column using 2 methods

```
1. ALTER TABLE `new_schema`.`users`
   ADD INDEX `name_index` (`name`);
```

```
2. CREATE INDEX `name_index` ON `new_schema`.`users`
   (`name`)
```

    - **NOTE:** we are converting the `name` column as an index called `name_index`
- We can delete an index column
    - ```
      ALTER TABLE `new_schema`.`users`
      DROP INDEX `name_index`;
      ```

The following are the most common types used for indices:

1. **PRIMARY KEY**
    - Since each row has this, we avoid NULL and duplicates
    - Database automatically builds an index on this column so joining on this is more optimal
2. **UNIQUE INDEX**
    - Good since its unique and improves efficiency of index usage
3. **FULL TEXT**
    - Useful for keyword searches in large blocks of text
    - Instead of indexing on the entire value, we break up the value into tokes and then build an inverted index that maps words to rows

The following are 6 best practices when choosing indices:

1. Choose a UNIQUE INDEX, otherwise, use KEY
2. Add an index for the column that is often referenced by complex operations
3. Index the hotspot column that is frequently queried in WHERE (but make sure difference in column values are large enough)
4. Pay attention to disk space used, indices use disk space
      ```

5. Index the column with a small value (optimize space)
6. Avoid index on columns with NULL

User Privilege

This section teaches good practices to set up users for SQL databases (not users for an application)

We create a SQL account like this:

```
CREATE USER 'john'@'localhost' IDENTIFIED BY 'password';
```

- `localhost` refers to the network context in which the account can be used from (so we would replace it with the corresponding network for the account to be used)
    - If the website server & database server runs on the same machine, localhost works
    - We would replace it with an IP address to be able to use it from a specific network
    - We can use `%` instead to allow a user to connect to the database anywhere
        - Careful with this one since we can have strangers access the database once the password is leaked

We can find who are users are:

```
SELECT * FROM `mysql`.`user`
```

We can grant a user permissions on how & if they can access certain parts of our database:

```
GRANT ALL ON `new_schema`.`orders` TO 'john'@'localhost';
GRANT SELECT ON `new_schema`.* TO 'root'@'150.10.12.1';
```

- GRANT gives permission
- ALL is just a keyword on what permissions we give (e.g: SELECT, INSERT, etc)