IEMS5722
Mobile Network Programming and Distributed Server Architecture

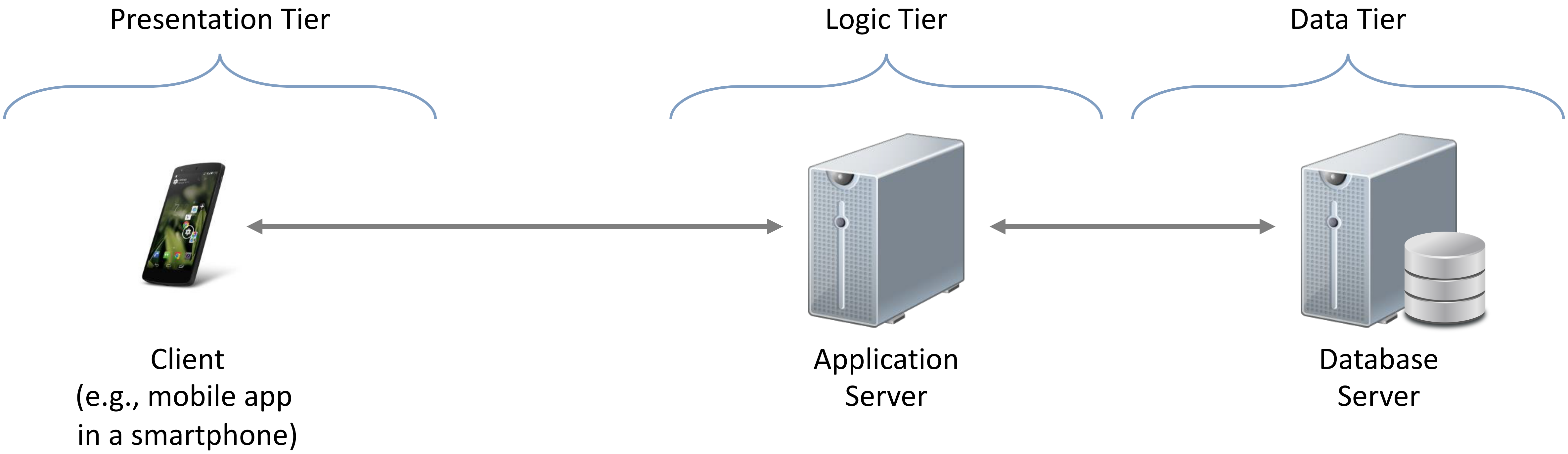
Lecture 6

Databases and Caches

Data and Databases

- Data can be considered as the **most important assets** in many Internet-based services, consider:
  - The social network and users’ interests in Facebook
  - The tweets in Twitter
  - The search index and cache in Google
  - …

Three-Tier Architecture



Presentation Tier

Logic Tier

Data Tier

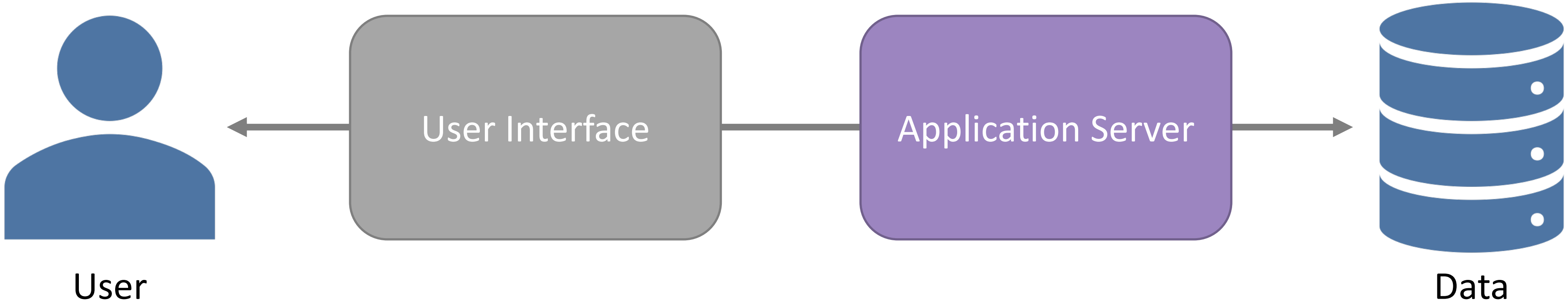
Client
(e.g., mobile app
in a smartphone)

Application
Server

Database
Server

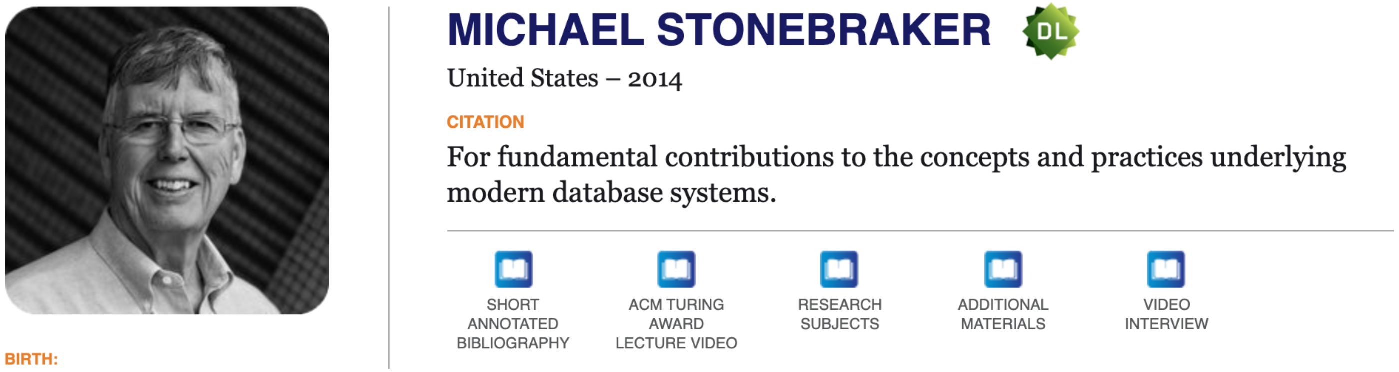
Data and Databases

- Most Internet-based services can be considered as some means for interacting with some data



Turing Award 2014

- Michael Stonebraker
- Involved in the invention and development of many relational database concepts (e.g. the object-relational model, query modification, etc.)


MICHAEL STONEBRAKER

United States – 2014

CITATION

For fundamental contributions to the concepts and practices underlying modern database systems.

BIRTH:

Reference: https://amturing.acm.org/award_winners/stonebraker_1172121.cfm

Relational Databases

Database Management Systems

- **Database Management System (DBMS)**
  - A system that stores and manages a (probably large) collection of data
  - It allows users to perform operations and manage the data collection (e.g. creating a new record, querying existing records)
  - Examples
    - Oracle
    - MS SQL Server
    - MySQL
    - PostgreSQL

Database Management Systems

- **Data Model**
  - A data model describes how data should be organized
  - It describes how data elements relate to one another
  - In most cases, a data model reflects how things are related in the real world

- A widely used data model is the **relational model of data**
  - A table describes a relation between different objects

Relational Databases

- A database is a collection of relations (**tables**)
- Each relation has a list of attributes (**columns**)
- Each attribute has a domain (**data type**)
- Each relation contains a set of tuples (**rows**)
- Each tuple has a value for each attribute of the relation (or **NULL** if no value is given)

Relational Databases

- Simple Example – Student Enrollment in Courses

Students

ID	Name	Year
1	John Chan	3
2	May Lee	4
…	…	…

Courses

ID	Code	Lecturer
1	IEMS 5722	Marco Ho
2	IEMS 5723	Rosanna Chan
…	…	…

Enrollment

ID	Student ID	Course ID
1	1	1
2	2	1
…	…	…

Relational Databases – Schema & Instance

- **Schema** (also known as **metadata**)
  - Specifies how data is to be structured
  - Needs to be defined before the database can be populated

- **Instance**
  - The actual content to be stored in the database
  - The structure of the data must conform to the schema defined beforehand

Relational Databases – Schema & Instance

- Example:
- A table “Student” with the following schema
  - (ID integer, name string, year integer, date_of_birth date)

- Some instances of “Student” in the table:
  - (1, “Peter Chan”, 3, 1999-01-01)
  - (2, “Mike Cheung”, 3, 1999-12-31)
  - …

Relational Databases

- How can we create schema and modify the data in a database management system?

- **SQL – Structured Query Language**
  - A standard language for querying and manipulating data in a relational database
  - It is both a DDL (data definitional language) and a DML (data manipulation language)
  - Defining schemas with “**create**”, “**alter**”, “**delete**”
  - Manipulating tables with “**insert**”, “**update**”, “**delete**”

SQL Introduction

- Let’s assume we have the following three tables

Students

ID	Name	Year
1	John Chan	3
2	May Lee	4
…	…	…

Courses

ID	Code	Lecturer
1	IEMS 5722	Marco Ho
2	IEMS 5723	Rosanna Chan
…	…	…

Enrollment

ID	Student ID	Course ID
1	1	1
2	2	1
…	…	…

SQL Introduction

- How can we create these tables?

```
CREATE TABLE Students (
    id INT NOT NULL AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    year INT NOT NULL,
    PRIMARY KEY (id)
);

CREATE TABLE Courses (
    id INT NOT NULL AUTO_INCREMENT,
    code VARCHAR(10) NOT NULL,
    lecturer VARCHAR(100) NOT NULL,
    PRIMARY KEY (id),
    UNIQUE (code)
);
```

AUTO_INCREMENT:
Wherever a new row is inserted into the table, it is automatically incremented by 1

PRIMARY KEY:
A key of the table, it can be used to uniquely identify a particular record in the table

UNIQUE:
The field must be unique for each row in the table

SQL Introduction

- **SELECT** statement
  - Used to retrieve data from one or more tables given some conditions
  - Example 1: retrieve the year of study of the student ‘John Chan’

```
SELECT year FROM Students WHERE name = 'John Chan';
```

  - Example 2: retrieve the name of the lecturer of course ‘IEMS 5722’

```
SELECT lecturer FROM Courses WHERE code = 'IEMS 5722';
```

SQL Introduction

- Example 3: retrieve a list of students whose name is ‘John’

```
SELECT * FROM Students WHERE name LIKE 'John %';
```

- Example 4: retrieve a list of courses, sort by the course codes in descending order

```
SELECT id, code, lecturer
FROM Courses
ORDER BY code DESC;
```

SQL Introduction

- Example 5: retrieve a list of students who has enrolled in ‘IEMS 5722’

```
SELECT s.id, s.name
FROM Students s, Courses c, Enrollment e
WHERE e.student_id = s.id
AND e.course_id = c.id
AND c.code = 'IEMS 5722';
```

Here, we are (**inner**) **joining** three tables in order to retrieve data based on their relationships

Reference:
https://en.wikipedia.org/wiki/Join_(SQL)
https://www.w3schools.com/sql/sql_join.asp
https://blog.codinghorror.com/a-visual-explanation-of-sql-joins/

SQL Introduction

- **INSERT** statement
  - Used to insert new data into tables
  - Example 1: insert a new student into the Students table

```
INSERT INTO Students (name, year)
VALUES ('Paul Wong', 4);
```

no need to specify id

  - Example 2: insert a new course into the Courses table

```
INSERT INTO Courses (code, lecturer)
VALUES ('IEMS 5678', 'Mike Cheung');
```

SQL Introduction

- **UPDATE** statement
  - Used to modify the data in the tables
  - Example 1: change the study year of the student with ID 12

```
UPDATE Students
SET year = 5
WHERE id = 12;
```

- Example 2: update the lecturer of the course ‘IEMS 5566’

```
UPDATE Courses
SET lecturer = 'David Chan'
WHERE code = 'IEMS 5566';
```

SQL Introduction

- **DELETE** statement
  - Used to remove data from the tables
  - Example 1: remove all year 7 students

```
DELETE FROM Students
WHERE year = 7;
```

  - Example 2: remove all the courses that are taught by ‘Anne Law’

```
DELETE FROM Courses
WHERE lecturer = 'Anne Law';
```

SQL Introduction

- For more complex SQL statements and queries, refer to the tutorials in the following Web sites
- MySQL Reference Manual: https://dev.mysql.com/doc/refman/5.7/en/tutorial.html
- MySQL Tutorial: https://www.mysqltutorial.org/
- W3School SQL Tutorial: https://www.w3schools.com/sql/

ACID Properties of Relational Database

- Relational databases focus on having reliable transactions, and usually have the ACID properties
    - **Atomicity** – Each transaction is either “all done” or “failed”
    - **Consistency** – Data can only be changed according to pre-defined rules
    - **Isolation** – Concurrent queries do not interfere with one another
    - **Durability** – Results are persistent in the databases

MySQL

- An open source relational database management system
- The world’s second most widely used RDBMS
- Most widely used RDBMS in a client-server model
- https://www.mysql.com/
- Community Edition – freely available on Windows, MacOS and Linux
- Enterprise Edition – More advanced functions with technical support
- In Ubuntu, install the MySQL server with

```
$ sudo apt-get install mysql-server
```

MySQL

- Once installed, you can use its command-line client to interact with MySQL. (By default, password for MySQL “root” is empty.)

```
$ sudo mysql -u root -p
...
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
4 rows in set (0.00 sec)
```

- Then, create a new user for the database as root.

```
mysql> CREATE USER 'dbuser'@'localhost' IDENTIFIED BY 'password';
Query OK, 0 rows affected (0.01 sec)

mysql> GRANT ALL PRIVILEGES ON *.* TO 'dbuser'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> exit
Bye
```

MySQL

- Creating a new database using the new “dbuser”

```
$ mysql -u dbuser -p
Enter password: (enter password here)
...

mysql> CREATE DATABASE iems5722;
Query OK, 1 row affected (0.00 sec)

mysql> USE iems5722;
Database changed

mysql> SHOW TABLES;
Empty set (0.00 sec)
```

Interfacing MySQL in Python

MySQL & Python

- In your server application, it is very likely that you will have to access or modify the data stored in the database
- Oracle (which owns MySQL) provides “MySQL Connector” driver for different languages (including Python)
- https://dev.mysql.com/doc/connector-python/en/
- Install the “MySQL Connector” driver using the following command:

```
$ sudo pip3 install mysql-connector-python
```

- Verify it is installed correctly (in Python):

```
$ python3
...
>>> import mysql.connector
>>>
```

No error messages

MySQL & Python

- Connecting to the MySQL database in Python

```
import mysql.connector

conn = mysql.connector.connect(
        host = "localhost",
        port = 3306,              # default, can be omitted
        user = "dbuser",
        password = "password",
        database = "iems5722",
)

# Create a cursor that return rows as dictionaries
cursor = conn.cursor(dictionary = True)

...

cursor.close()
conn.close()
```

MySQL & Python

- Executing a query

Students

ID	Name	Year
1	John Chan	3
2	May Lee	4
3	Paul Wong	4

```
query = "SELECT * FROM Students ORDER BY id ASC"

# Execute the query
cursor.execute(query)

# Retrieve all the results
results = cursor.fetchall()

# "results" is a list of rows, each row is a dictionary
# The following line prints "John Chan"
print(results[0]["name"])
```

MySQL & Python

- You can also fetch records one after another

Students

ID	Name	Year
1	John Chan	3
2	May Lee	4
3	Paul Wong	4

```
query = "SELECT * FROM Students ORDER BY id ASC"

# Execute the query
cursor.execute(query)

# Retrieve one row at a time
row = cursor.fetchone()
while row is not None:
    print(row["name"])
    row = cursor.fetchone()

# Output would be
# "John Chan"
# "May Lee"
# "Paul Wong"
```

MySQL & Python

- Parameter substitution
  - Very often you have values stored in Python variables, and would like to use them in the SQL queries

```
student_id = request.form.get("student_id")
year = request.form.get("year")

query = "UPDATE Students SET year = %s WHERE id = %s"

# Prepare the parameters (must be a tuple!)
params = (year, student_id)

# Execute the query by substituting the parameters
cursor.execute(query, params)
conn.commit() # Remember to commit if you have changed the data!
```

MySQL & Python

- Executing multiple queries
  - Sometimes you may want to execute many queries with a list of values

```
students = [
    ("Peter Lo", "1"),
    ("William Wong", "2"),
]

query = "INSERT INTO Students (name, year) VALUES (%s, %s)"

# Execute multiple queries at at once with a list of parameters
cursor.executemany(query, students)
conn.commit()
```

Useful Resources

- **MySQL**
  - SQL Tutorial
    https://www.w3schools.com/sql/default.asp
  - MySQL Reference Manual
    https://dev.mysql.com/doc/refman/5.7/en/
- **MySQL Connector**
  - MySQL Connector/Python
    https://dev.mysql.com/doc/connector-python/en/
  - W3Schools Python MySQL Guide
    https://www.w3schools.com/python/python_mysql_getstarted.asp

Using MySQL with Flask

Connecting to MySQL in Flask

- Recall that we use Flask to develop our APIs for our mobile apps

```
from flask import Flask
app = Flask(__name__)

@app.route("/get_students")
def get_students():
    ...
    return ...

if __name__ == "__main__":
    app.run()
```

Connecting to MySQL in Flask

- What if we need to develop an API for retrieving the list of students from the database?

```
from flask import Flask
app = Flask(__name__)

@app.route("/get_students")
def get_students():

    # 1. Connect to the database
    # 2. Construct a query
    # 3. Execute the query
    # 4. Retrieve the data
    # 5. Format and return the data

    return ...

if __name__ == "__main__":
    app.run()
```

Connecting to MySQL in Flask

- Create a database class for readability and reusability

```
class MyDatabase:
    conn = None
    cursor = None

    def __init__(self):
        self.connect()
        return

    def connect(self):
        self.conn = mysql.connector.connect(
            host = "localhost",
            port = 3306,                 # default, can be omitted
            user = "dbuser",
            password = "password",
            database = "iems5722",
        )
        self.cursor = self.conn.cursor(dictionary = True)
        return
```

Connecting to MySQL in Flask

- Let’s go ahead and implement the get_students() function

```
@app.route("/get_students")
def get_students():
    # 1. Connect to the database
    mydb = MyDatabase()

    # 2. Construct a query
    query = "SELECT * FROM Students"

    # 3. Execute the query
    mydb.cursor.execute(query)

    # 4. Retrieve the data
    students = mydb.cursor.fetchall()

    # 5. Format and return the data
    return jsonify(data=students)
```

Connecting to MySQL in Flask

- What if we need to implement an API for retrieving the data of a single student?

```
@app.route("/student/<int:student_id>")
def get_single_student(student_id):
    # 1. Connect to the database
    mydb = MyDatabase()
    # 2. Construct a query
    query = "SELECT * FROM Students WHERE id = %s"
    params = (student_id,)       # must be tuple, note the comma!
    # 3. Execute the query
    mydb.cursor.execute(query, params)
    # 4. Retrieve the data
    student = mydb.cursor.fetchone()
    # 5. Format and return the data
    if student is None: # No such student is found!
        return jsonify(status="ERROR", message="Not Found!")
    else:
        return jsonify(status="OK", data=student)
```

To use this API, send a get request to, for example,
/student/2
to retrieve the data of the student with id = 2

Before and After Request

- In Flask, you can specify some codes to be executed before and/or after a request from the client
- This is done by implementing the **before_request** and **teardown_request** functions

```
@app.before_request
def before_request():
    ...
    return

@app.teardown_request
def teardown_request(exception):
    ...
    return
```

Before and After Request

- How would you use these two functions?
    1. Create a database connection before a request, and close the connection after the request
    2. Log the request to the database or to a file before each request
    3. Check user authentication before each request
    4. …

Before and After Request

- Example
  - We create the database connection before the request, store it in the globally available object “g”, and close the connection after the request

```
@app.before_request
def before_request():
    g.mydb = MyDatabase()
    return

@app.teardown_request
def teardown_request(exception):
    mydb = getattr(g, "mydb", None)
    if mydb is not None:
        mydb.conn.close()
    return
```

g is a global object from Flask that is available throughout the whole process.

Remember to import it by:
from flask import g

Before and After Request

- Then, in our API function, we can simply write:

```
@app.route("/student/<int:student_id>")
def get_single_student(student_id):
    # 2. Construct a query
    query = "SELECT * FROM Students WHERE id = %s"
    params = (student_id,)       # must be tuple, note the comma!

    # 3. Execute the query
    g.mydb.cursor.execute(query, params)

    # 4. Retrieve the data
    student = g.mydb.cursor.fetchone()

    # 5. Format and return the data
    if student is None: # No such student is found!
        return jsonify(status="ERROR", message="Not Found!")
    else:
        return jsonify(status="OK", data=student)
```

Reference: https://flask.palletsprojects.com/en/1.1.x/tutorial/database/

NoSQL Databases

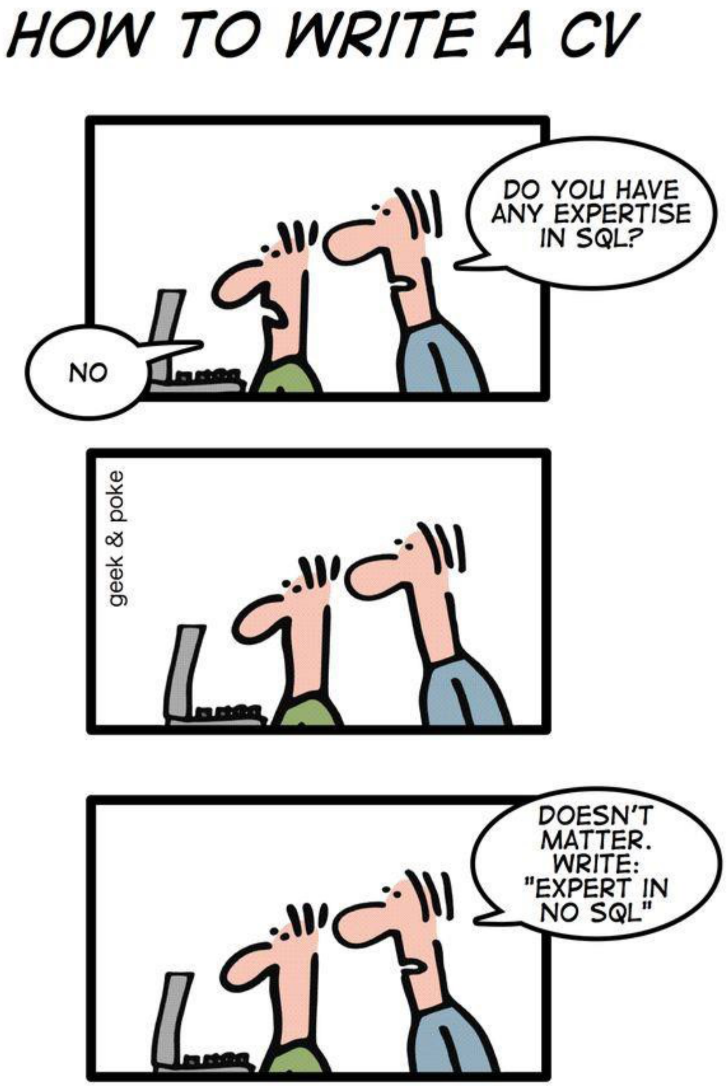
NoSQL

- The relational model of data and relational databases are powerful tools for managing data, but they cannot solve all problems
  - Data Model – data may be better modelled as **objects in a hierarchy** or a **graph**
  - Scheme – in many applications, it can be too restrictive to have **fixed schema**
  - Scalability – it takes a lot of effort to **horizontally scale relational databases**

- Alternative solutions are therefore desirable for solving new problems

NoSQL

- NoSQL (non-SQL, non-relational, not-only-SQL) systems are storage systems that offer users the ability to model data in ways other than relational tables.
  - It is NOT a single technology
  - No single definition of a NoSQL database
  - Many different systems or solutions are available for solving different problems


HOW TO WRITE A CV
DO YOU HAVE ANY EXPERTISE IN SQL?
NO
geek & poke
DOESN'T MATTER. WRITE: "EXPERT IN NO SQL"

Leverage the NoSQL boom

Why do we need NoSQL Databases?

1. **Popularity of Web applications and services**
    - Many reads and writes due to user participation (user-generated content)
    - Complex functions require flexibility in data models (e.g. find friends of friends in a social network, find related items bought by users of the same age group, …)
    - Horizontal scaling is desirable

Why do we need NoSQL Databases?

2. **Flexibility in data schema is required**
    - Relational database requires data schema to be well-defined
    - However, in many applications there can be a lot of attributes and these attributes may change over time

3. **Different solutions are required to handle different types of data**
    - Structured vs. semi/unstructured data
    - Data that needs to be served real-time vs. log data

NoSQL

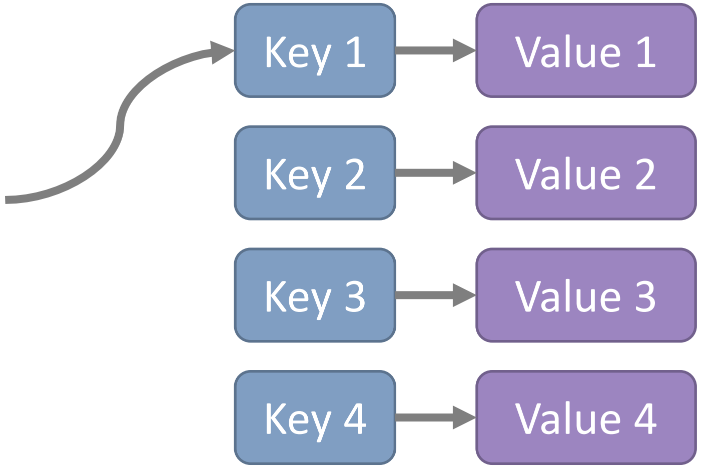
- Some **common features** of NoSQL database systems:
  - Do not require the definition of a fixed schema
  - Scale horizontally (distributed operations, replication and partition) over multiple servers
  - Simple or no query language, offer APIs for data manipulation
  - A weaker concurrency model (not ACID)
  - Distributed storage

NoSQL Database Systems

- The different **types** of NoSQL database systems
  - Key-value stores
  - Document databases
  - Graph databases
  - Column databases
  - Object databases

NoSQL Database Systems

- **Key-value stores**
  - Examples are Redis, Riak, Oracle NoSQL Database
  - Implementing a dictionary or a hash
  - Retrieval of data is very fast
  - For quickly retrieving the value of a known key, but not good for searching



NoSQL Database Systems

- **Document stores**
  - Examples are CouchDB and MongoDB
  - Similar to key-value stores, but value is a document
  - Document is in a semi-structured format (e.g. JSON or XML)
  - Allow retrieval of documents by searching their content

NoSQL Database Systems

- **Graph databases**
  - Examples are Neo4j, DataStax and OrientDB
  - Store data in the form of
    - Nodes (entities)
    - Edges (relations between entities)
    - Properties (attributes of nodes or edges)
  - Perform queries on graphs without the need to carry out expensive JOIN operations

Redis

- Redis (Remote Dictionary Server)
  https://redis.io/topics/introduction
- An open source in-memory data structure store
- Can be used as a key-value database, cache, or message broker
- Install redis in Ubuntu with the following command

```
$ sudo apt-get install redis-server
```

- You can check if the server has been installed successfully by running the redis command line tool:

```
$ redis-cli
127.0.0.1:6379>
```

Redis

- You can easily interface with Redis in Python
- Install the Python redis module with the following command:

```
$ sudo pip3 install redis
```

- Check whether the installation is successful (in Python):

```
>>> import redis
>>>
```

No error messages

Redis

- A simple example

```
# import redis
from redis import StrictRedis

# Establish a connection to redis on localhost
r = StrictRedis('localhost')

# Set the value of a key
r.set('test_key', 'test_value')

# Get the value of a key
# value will be None if no such key is found in redis
value = r.get('test_key')
```

Redis

- You can store strings, lists, sets, or even bit arrays in Redis
- It also supports counters (increment or decrement the value)
- More examples below:

```
# Create a counter, initialize it
r.set('counter', 1)

# Increment the counter
r.incr('counter')
...

# Push a string into a list
r.rpush('user_list', 'John Chan')
...
```

Reference: https://redis.io/topics/data-types-intro

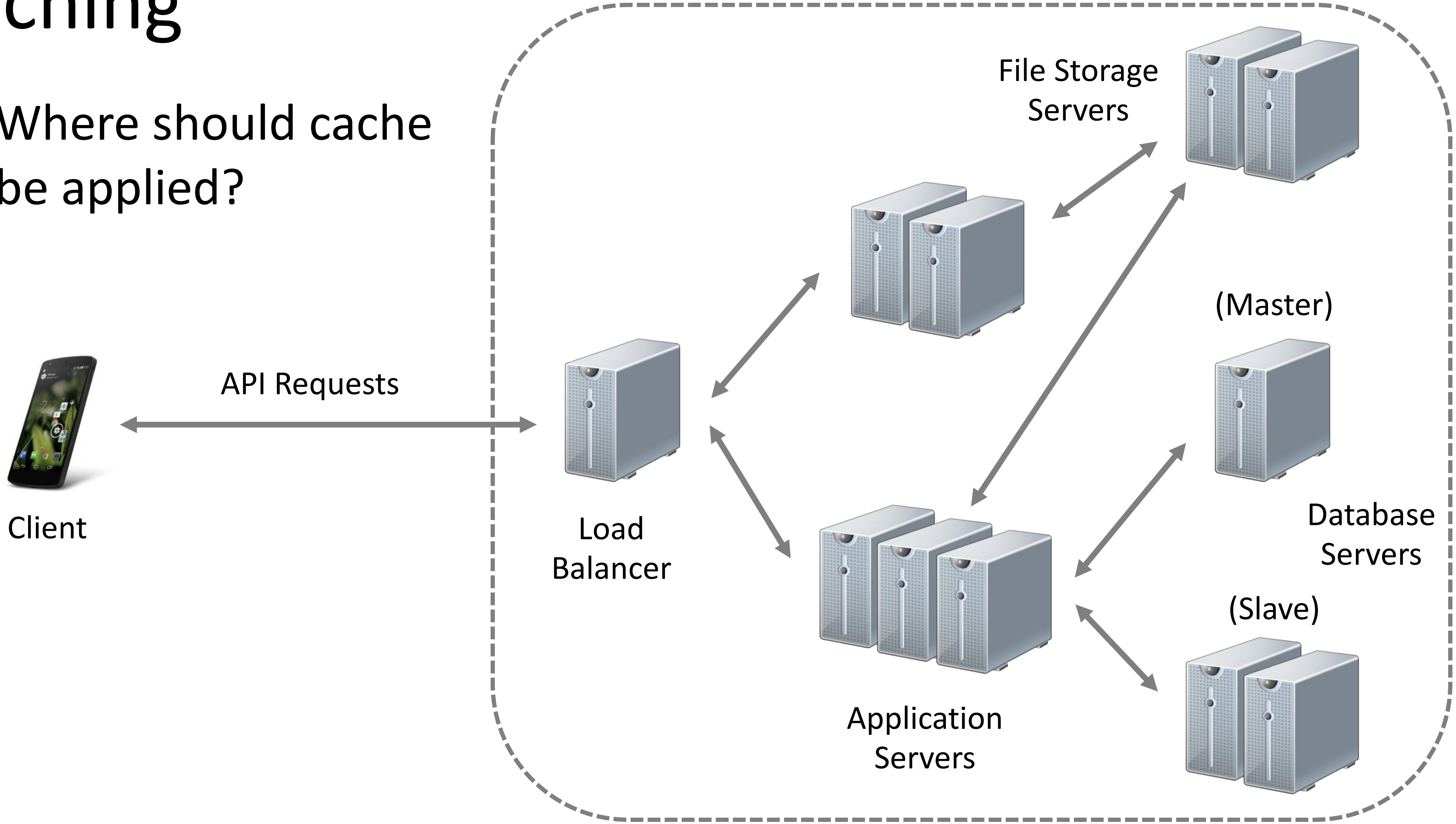
Caching

Caching

- **Cache** is a temporary data storage that stores data for quick retrieval in the future
  - Mostly implemented as a key-value store, where the unique key can be used to retrieve the value at O(1) time
  - Cache is usually small (RAM is expensive!)
  - Hit (found) vs. Miss (not found)
  - Cache can be persistent, if it also stores the current state into some persistent storage (e.g. the hard disk)

Caching

- Where should cache be applied?



Memcached

- A general purpose distributed memory caching system
  - **General purpose** – can be used in front of a Web server, an application server, or a database server
  - **Distributed** – can be operated on multiple servers for scalability
  - **Memory** – stores values in RAM, if not enough RAM, discard old values

Memcached + Nginx

The key-value pair should be inserted into Memcached by the application (external to Nginx)

```
server {
    location / {
        set $memcached_key "$uri?$args";
        memcached_pass host:11211;
        error_page 404 502 504 = @fallback;
    }

    location @fallback {
        proxy_pass http://backend;
    }
}
```

Memcached + MySQL

```
import sys
import mysql.connector
import memcache

mem = memcache.Client(['127.0.0.1:11211'])
conn = mysql.connector.connect(...)
...

user_record = memc.get('user_5')

if not user_record:
    # retrieve user record from MySQL
else:
    # data available, retrieved from Memcached
```

Memcached

- More references can be found at:
  - Official Website
    https://memcached.org/
  - Python Memcached module:
    https://pypi.org/project/python-memcached/
  - Caching in Flask:
    https://flask.palletsprojects.com/en/1.1.x/patterns/caching/
    https://flask-caching.readthedocs.io/en/latest/
  - Using MySQL and Memcached with Python:
    https://dev.mysql.com/doc/mysql-ha-scalability/en/ha-memcached-interfaces-python.html

Using Databases in Android

Using Database in Android

- When developing your Android app, it is not uncommon that you want to store structured data locally on the device
    - In Android, there is a relational database library (SQLite)
    - An abstract layer called Room over SQLite is available
    - Save data to a local database using Room: https://developer.android.com/training/data-storage/room
    - Save data using SQLite API directly https://developer.android.com/training/data-storage/sqlite
    - Another convenient library for interacting with the database, DBFlow: https://github.com/agrosner/DBFlow

Next Lecture:
Instant Messaging and Push Notifications

(Create a Google account first if you don't have one)

End of Lecture 6