Department of Information Engineering, CUHK
MScIE – 2nd Semester, 2015/16

IEMS 5722
Mobile Network Programming and Distributed Server Architecture

Lecture 6
Databases and Caches

Lecturer: Albert C. M. Au Yeung

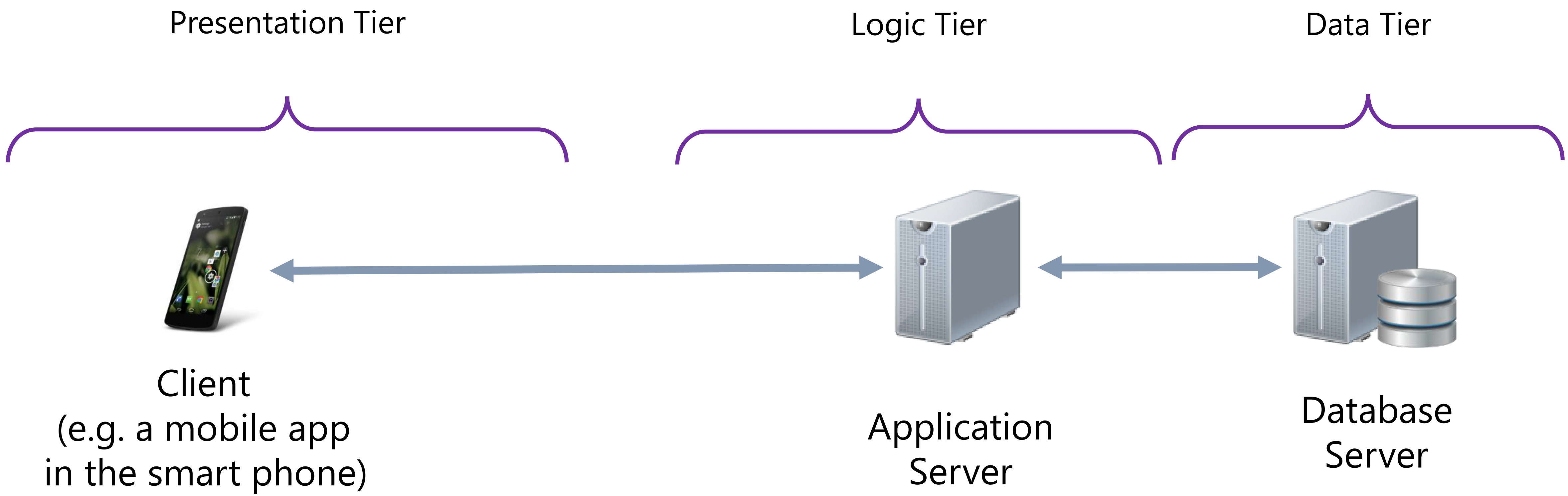
18th February, 2016

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



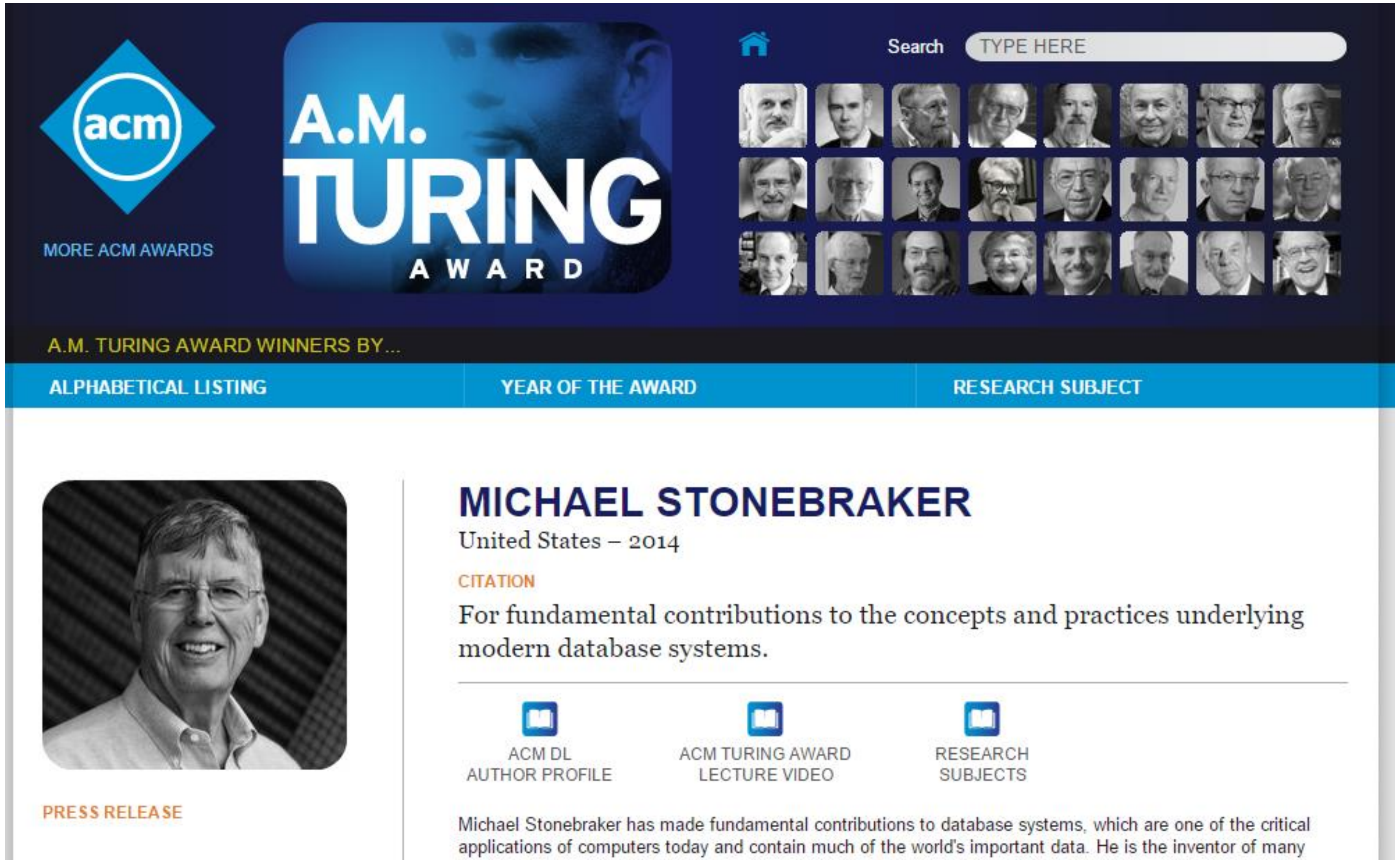
Data and Databases

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.)



http://amturing.acm.org/award_winners/stonebraker_1172121.cfm

Relational Databases

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
  - Postgre SQL

Data Model

- A data model describes how data should be organised
- 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)

Simple Example – Student Enrollment in Courses

Students

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

Courses

ID	Code	Lecturer
1	IEMS 5723	...
2	IEMS 5722	...
...	...	...

Students 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, 1996-03-17)
  - (2, ‘Mike Cheung’, 3, 1996-05-19)
  - …

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 two tables

Students

id	name	year
1	John Chan	3
2	May Lee	4
…	…	…

Courses

id	code	lecturer
1	IEMS 5723	…
2	IEMS 5722	…
…	…	…

Enrollment

id	student_id	course_id
1	1	1
2	2	1
…	…	…

SQL Introduction

How can we create these tables?

```
CREATE TABLE Student (
        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 you insert a new row into the table, it automatically increments 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

SELECT statement

- Used to retrieve data from one or more tables given some conditions
- Example 1: retrieve the E-mail address of the student 'John Chan'

```
SELECT email 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 their course code in descending order

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

- Example 5: Retrieve a list of students who have 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 **joining (inner join)** the three tables in order to retrieve data based on their relationships

References:

- https://en.wikipedia.org/wiki/Join_(SQL)
- http://blog.codinghorror.com/a-visual-explanation-of-sql-joins/

INSERT statement

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

```
INSERT INTO Students (name, email)
VALUES (‘Paul Wong’, ‘pw@gmail.com’)
```

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

```
INSERT INTO Courses (code, lecturer)
VALUES (‘IEMS 5678’, ‘Prof. Cheung’)
```

UPDATE statement

- Used to modify the data in the tables
- Example 1: Change the email address of the student with id = 12

```
UPDATE Students
SET email = ‘abc123@gmail.com’
WHERE id = 12
```

- Example 2: Update the lecturer of the course with course code ‘IEMS 3456’

```
UPDATE Courses
SET lecturer = ‘Prof. Chan’
WHERE code = ‘IEMS 3456’
```

DELETE statement

- Used to modify the data in the tables
- Example 1: Change the email address of the student with id = 12

```
UPDATE Students
SET email = ‘abc123@gmail.com’
WHERE id = 12
```

- Example 2: Update the lecturer of the course with course code ‘IEMS 3456’

```
UPDATE Courses
SET lecturer = ‘Prof. Chan’
WHERE code = ‘IEMS 3456’
```

SQL Introduction

For more complex SQL statements and queries, refer to the tutorials in the following Web sites

- MySQL Reference Manual: http://dev.mysql.com/doc/refman/5.7/en/tutorial.html
- MySQL Tutorial: http://www.mysqltutorial.org/
- W3School SQL Tutorial: http://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
- http://www.mysql.com/
- Community Edition – freely available on Windows, Mac OS 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 MySQL

```
$ mysql -uroot -p
...
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| phpmyadmin         |
+--------------------+
5 rows in set (0.01 sec)
```

```
mysql> create database iems5722;
Query OK, 1 row affected (0.03 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
- In Python, you can make use of the MySQLdb module to help you execute SQL statements (http://mysql-python.sourceforge.net/MySQLdb.html)
- Install the MySQLdb module using the following command:

```
$ sudo apt-get install python-mysqldb
```

- Check if it has been installed successfully by importing MySQLdb

```
>>> import MySQLdb
>>>
```

Connecting to the MySQL database

```
import MySQLdb

db = MySQLdb.connect(
      host = "localhost",
      port = 3306,
      user = "dbuser",
      passwd = "password",
      db = "mydb",
      use_unicode = True,
      charset = "utf8",
      cursorclass = MySQLdb.cursors.DictCursor
)
```

Executing an SQL query

```
query = “SELECT * FROM Students ORDER BY id ASC”

# Execute the query
db.cursor.execute(query)

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

# results is a list of rows, each is a dictionary
# The following line prints ‘John Chan’
print results[0][‘name’]
```

MySQL & Python

- You can also fetch records one after another

```
query = “SELECT * FROM Students ORDER BY id ASC”

# Execute the query
db.cursor.execute(query)

# Retrieve rows one by one
row = db.cursor.fetchone()
while row is not None:
      print row[‘name’]

...
```

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”)
email = request.form.get(“email”)

query = “UPDATE Students SET email = %s WHERE id = %s”

# prepare the parameters (must be a tuple!)
params = (email, student_id)

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

Executing multiple queries

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

```
students = [
       ('May Chan', 'mc@gmail.com'),
       ('Peter Lo', 'pl@gmail.com'),
       ('William Wong', 'ww@gmail.com')
]

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

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

Using MySQL with Your Flask App

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():
    ...

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?

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

@app.route('/get_students')
def get_students():
    # 1. Connect to database
    # 2. Construct a query
    # 3. Execute the query
    # 4. Retrieve data
    # 5. Format and return the data

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

Connecting to MySQL in Flask

- For readability and reusability, let's create a class that will help us connect to the database

```
class MyDatabase:
    db = None

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

    def connect(self):
        self.db = MySQLdb.connect(
                          host = "localhost",
                          port = 3306,
                          user = "...",
                        passwd = "...",
                            db = "..."
                   use_unicode = True,
                       charset = "utf8",
                   cursorclass = MySQLdb.cursors.DictCursor
             )
        return
```

Connecting to MySQL in Flask

- Let's implement the get_students() function

```
@app.route('/get_students')
def get_students():

    # Create the database object
    mydb = MyDatabase()

    # Prepare and execute the query
    query = "SELECT * FROM Students"
    mydb.db.cursor.execute(query)

    # Retrieve the data and send response
    students = mydb.db.cursor.fetchall()
    return jsonify(data=students)
```

Connecting to MySQL in Flask

- Let's see another example, 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():

    mydb = MyDatabase()
    query = "SELECT * FROM Students WHERE id = %s"
    params = (student_id,) # Note the comma here!
    mydb.db.cursor.execute(query, params)

    student = mydb.db.cursor.fetchone()
    if student is None: # No such student is found!
        return jsonify(status="ERROR", message="Not Found!")
    else:
        return jsonify(status="OK", data=students)
```

To use this API, send a GET request to, for example, **/student/2**

(Retrieve 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():
    # Your code here...
    return

@app.teardown_request
def teardown_request(exception):
    # Your code here...
    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.db.close()
    return
```

g is an object that is available throughout the whole request, thus it will be available to you in the API functions you implement

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():

    query = “SELECT * FROM Students WHERE id = %s”
    params = (student_id,) # Note the comma here!
    g.mydb.db.cursor.execute(query, params)

    student = g.mydb.db.cursor.fetchone()
    if student is None: # No such student is found!
        return jsonify(status=“ERROR”, message=“Not Found!”)
    else:
        return jsonify(status=“OK”, data=students)
```

Reference: http://flask.pocoo.org/docs/0.10/tutorial/dbcon/

NoSQL Databases

NoSQL

The relational model of data and relational databases are powerful tools for managing data, but they cannot solve all problems

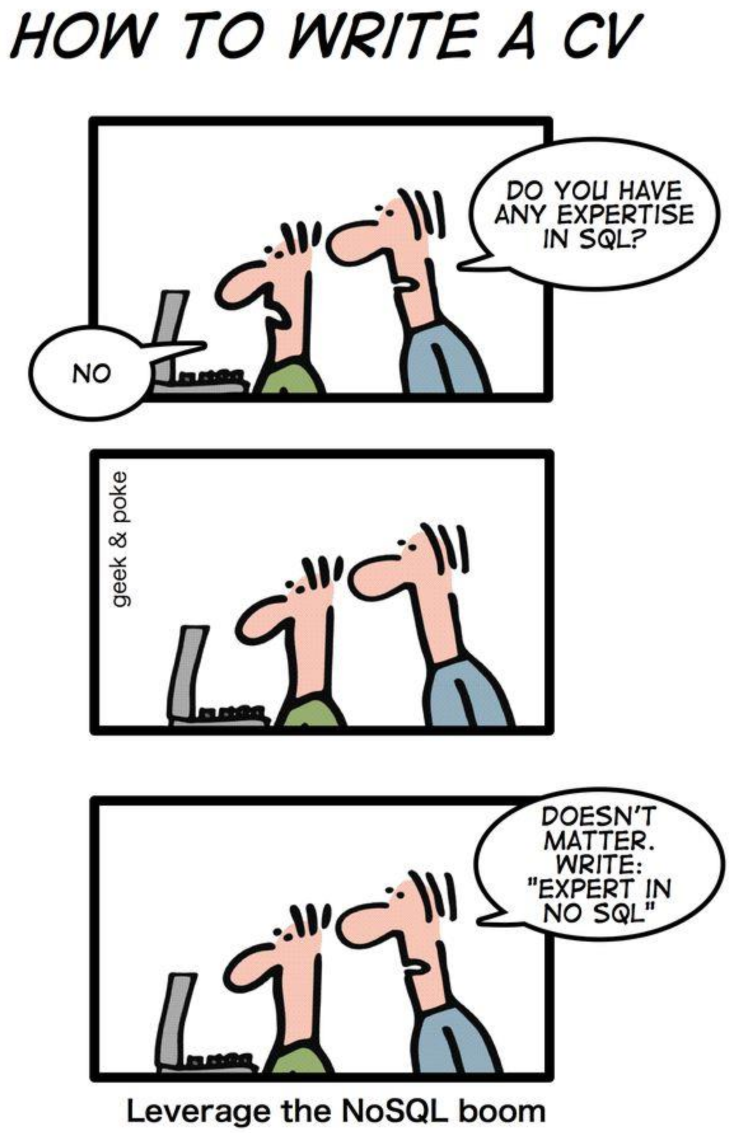
- Data Model - data may be better modelled as objects in a hierarchy or a graph
- Scheme - in many applications, it can be too restrictive to have fixed schema
- Scalability - it takes a lot of effort to horizontally scale relational databases

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



Why do we need NoSQL Databases?

1. Popularity of Web applications and services

- Many writes and reads because of 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 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 manipulating the data
- A weaker concurrency model (not ACID)
- Distributed storage

NoSQL Database Systems

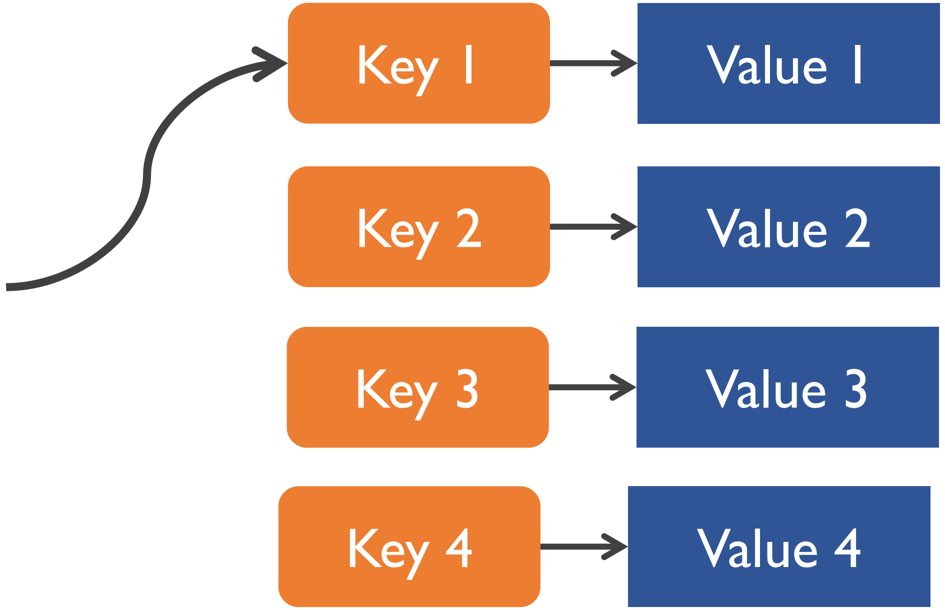
The different types of NoSQL database systems

- Key-value stores
- Document databases
- Graph databases
- Column databases
- Object databases

NoSQL Database Systems

Key-value stores

- Examples are Redis, Riak, Orcale 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, Titan 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

NoSQL Data Models

Data models in NoSQL databases are very different from that in relational databases

Major principles in NoSQL data modelling are:

- Denormalisation
- Aggregation
- Application-level Join

NoSQL Data Models

Normalisation

- Database normalisation is the process of organising tables in a relational database to minimize data redundancy

Image ID	Image Name	User	Tag
1	…	A	Cat
2	…	A	Dog
3	…	B	Cat
4	…	B	Fish

Image ID	User Id
1	1
2	1
3	2
4	2

Image ID	Tag ID
1	1
2	2
3	1
4	3

Image ID	Image Name
1	…
2	…
3	…
4	…

NoSQL Data Models

Denormalisation

- Normalisation ensures minimal redundancy, but then you will need to perform (a lot of) join operations to get what you want
- Denormalisation is the opposite, to improve *performance* and *scalability*, we add redundant data such that we can avoid joins
- Very fast read, but may have more complex and slower write/update logic
  → *not a problem because writes can wait*

NoSQL Data Models

Aggregation

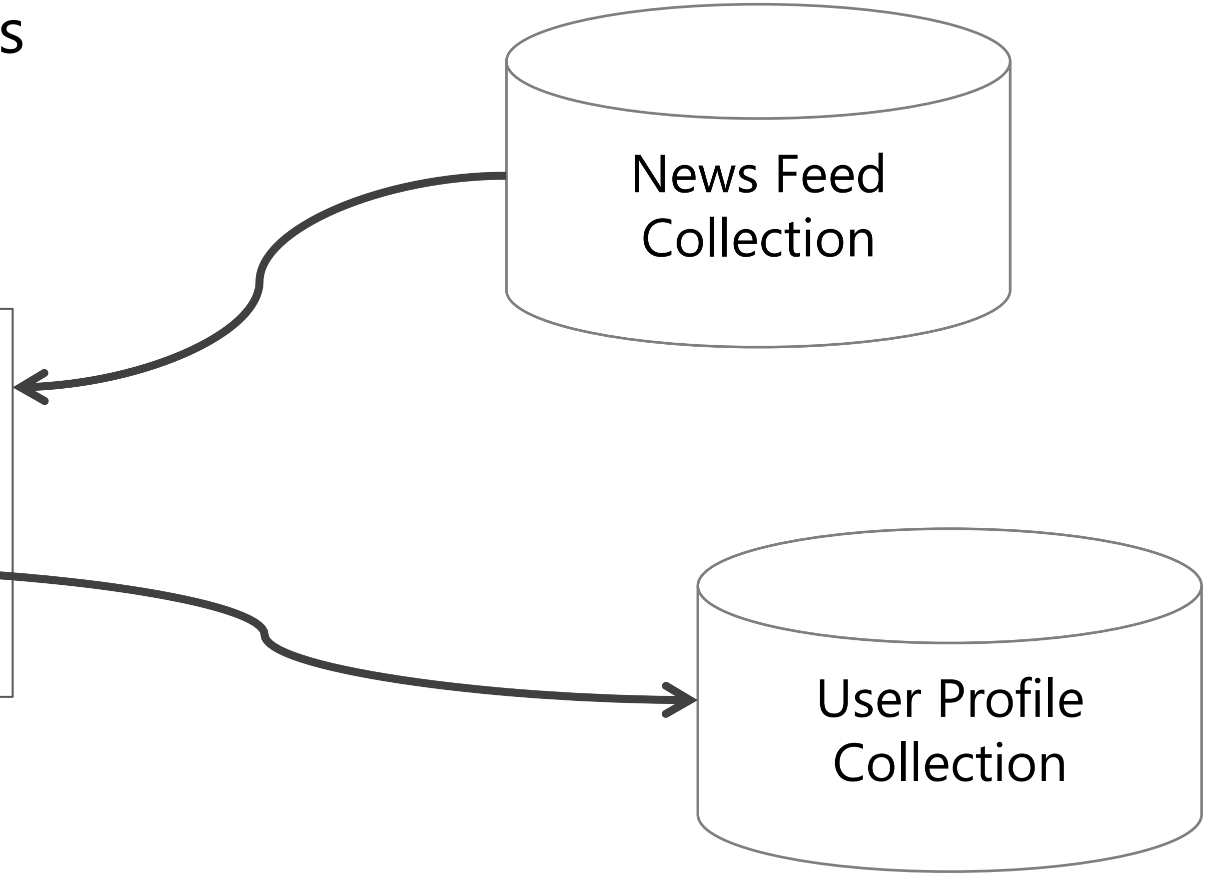
- Because NoSQL databases usually do not place constraints on values and schemas, we can aggregate different objects and attributes under the same record for the sake of performance
- Especially when dealing with many-to-many relations
  (e.g. hashtags and images, tracks and albums, users' comments on one another, etc.)

NoSQL Data Models

Application-Level Join

- Combines data in the application instead of relying on complex join queries in the database

```
{
    “news”: [
        “id”: 10,
        “title”: “Great Restaurant!”,
        “likes”: [1,9,12,133,234]
    ]
}
```



NoSQL Data Models

Application-Level Join

Pros:

- No complex join operations on the database side, retrieval of data can be very fast
- Frequently used data can be cached for even faster retrieval

Cons:

- Most likely you will have to issue multiple queries to the database to retrieve data
- Application may need to be optimised to ensure performance

Redis

- http://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 pip install redis
```

- Check whether the installation is successful:

```
>>> import 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, initialise it
r.set('counter', 1)

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

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

References: http://redis.io/topics/data-types-intro

Caching

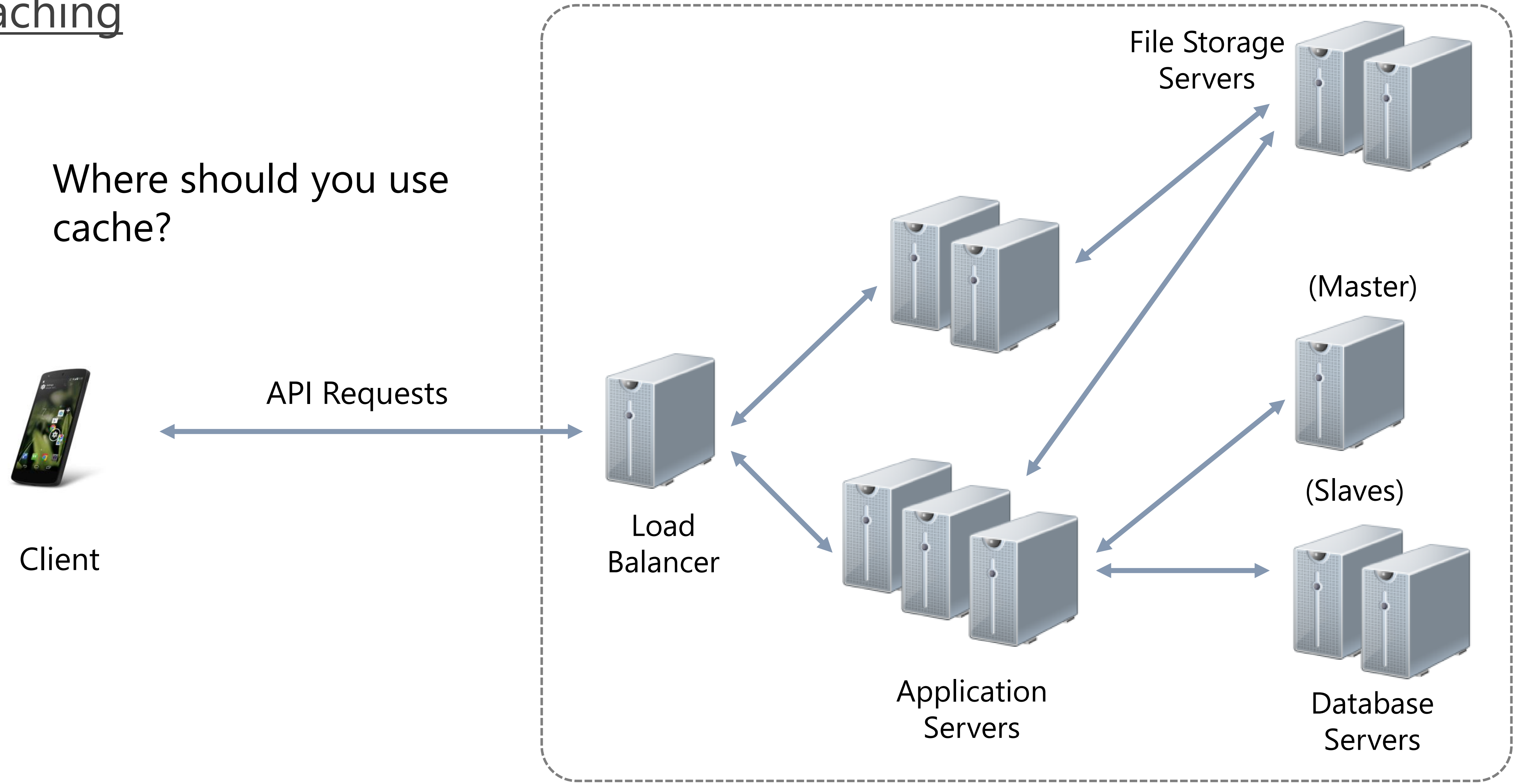
Caching

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 you use cache?



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

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

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

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

Memcached + MySQL

```
import sys
import MySQL
import memcache

mem = memcache.Client([‘127.0.0.1:11211’])
conn = MySQLdb.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:

- http://memcached.org/
- Python Memcached module: https://pypi.python.org/pypi/python-memcached
- Caching in Flask: http://flask.pocoo.org/docs/0.10/patterns/caching/
- Using MySQL and Memcached with Python: https://dev.mysql.com/doc/mysql-ha-scalability/en/ha-memcached-interfaces-python.html

Next Lecture:
Instant Messaging and Push Notifications

(Create a Google account if you have not)

End of Lecture 6