Information Processing

Week 4

Cloud Services, Client-Server, NoSQL Database

Today's Topics

- This presentation describes the activities covered in Lab 4
  - **AWS Connectivity, Client-Server, NoSQL Database**
- The presentation has two parts:
  - **Part I**: Describes how to make some basic use of a cloud platform to run your server-side apps
    - How to remotely configure a virtual server on the cloud platform (Amazon's AWS is taken as an example)
    - How to make local apps communicate with remote apps running on the cloud
    - In particular: approaches to multi-client handling
  - **Part II**: Describes how to use a database on a reliable remote server to implement persistence features in your apps
    - How to perform basic CRUD (Create, Read, Update, Delete) and other operations on the database from your server-side code

Intended Leaning Outcomes

After this week's discussion, you will be able to:

- Implement a realistic cloud-based server capable of handling multiple concurrent clients using different concurrency models
- Select appropriate server implementation strategies based on workload and design requirements
- Integrate a NoSQL database with server code, design efficient queries, and evaluate their performance trade-offs

Architecture of a Typical IoT System

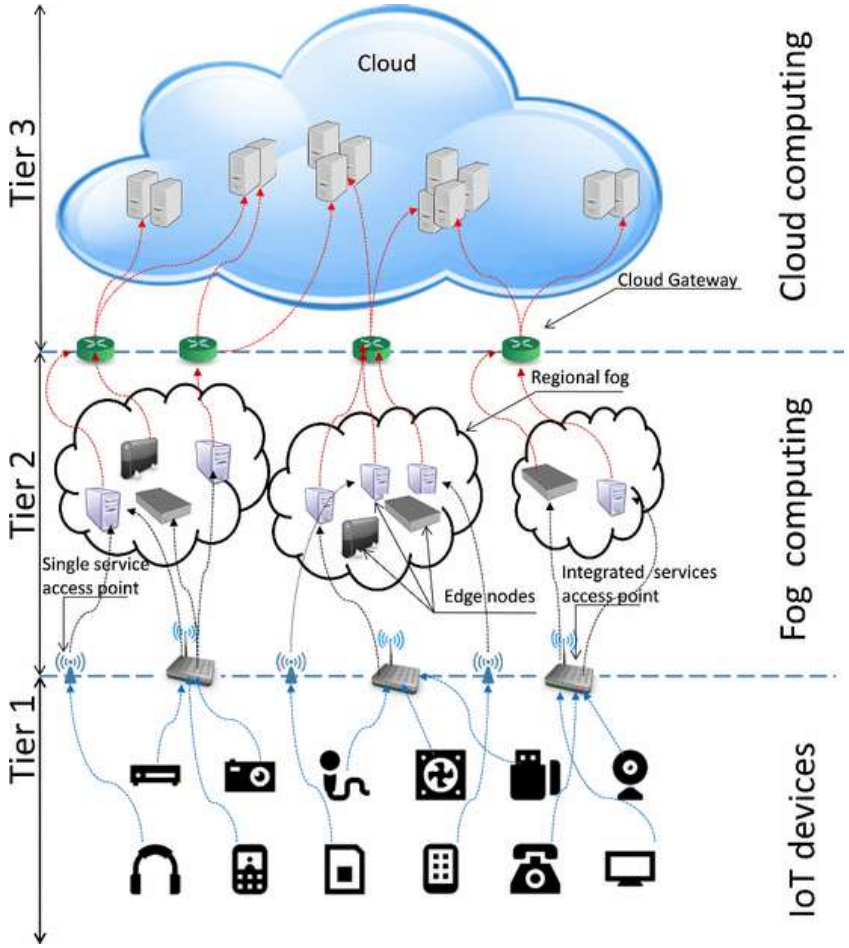
Roughly mapping your current system nodes to the three tiers:

- **Tier 1 — IoT / Edge devices**
  - Closest to the physical system; perform low-latency, hardware-coupled computation and control, e.g., Zynq board (PS + PL with hardware acceleration, etc.)
- **Tier 2 — Fog / Edge Computing**
  - Local intermediary layer for aggregation, preprocessing, control, and cloud communication, e.g., local laptop / local computing component (interface to Zynq PS, AWS client)
- **Tier 3 — Cloud computing**
  - Centralised, elastic compute and storage for coordination, services, and long-term data, e.g., AWS server, AWS database

The centralized Tier-3 cloud layer enables wide and remote access to the application by providing globally reachable services and shared state.

Database belongs in Tier 3 to provide global, persistent, shared state, especially when multiple fog nodes are involved.

Communication between tiers is handled over **networked interfaces**, with Tier-2 acting as the gateway between edge devices and cloud services.



Three-tier architecture of a typical IoT system

Cloud Computing – What is it?

- “Cloud computing is an information technology (IT) paradigm that enables ubiquitous access to shared pools of configurable system resources and higher-level services that can be rapidly provisioned with minimal management effort, often over the Internet. Cloud computing relies on sharing of resources to achieve coherence and economies of scale, similar to a public utility.”
- “Simply put, cloud computing is the delivery of computing services – servers, storage, databases, networking, software, analytics and more – over the Internet (“the cloud”). Companies offering these computing services are called cloud providers and typically charge for cloud computing services based on usage, similar to how you’re billed for gas or electricity at home.”

https://azure.microsoft.com/en-gb/overview/what-is-cloud-computing/

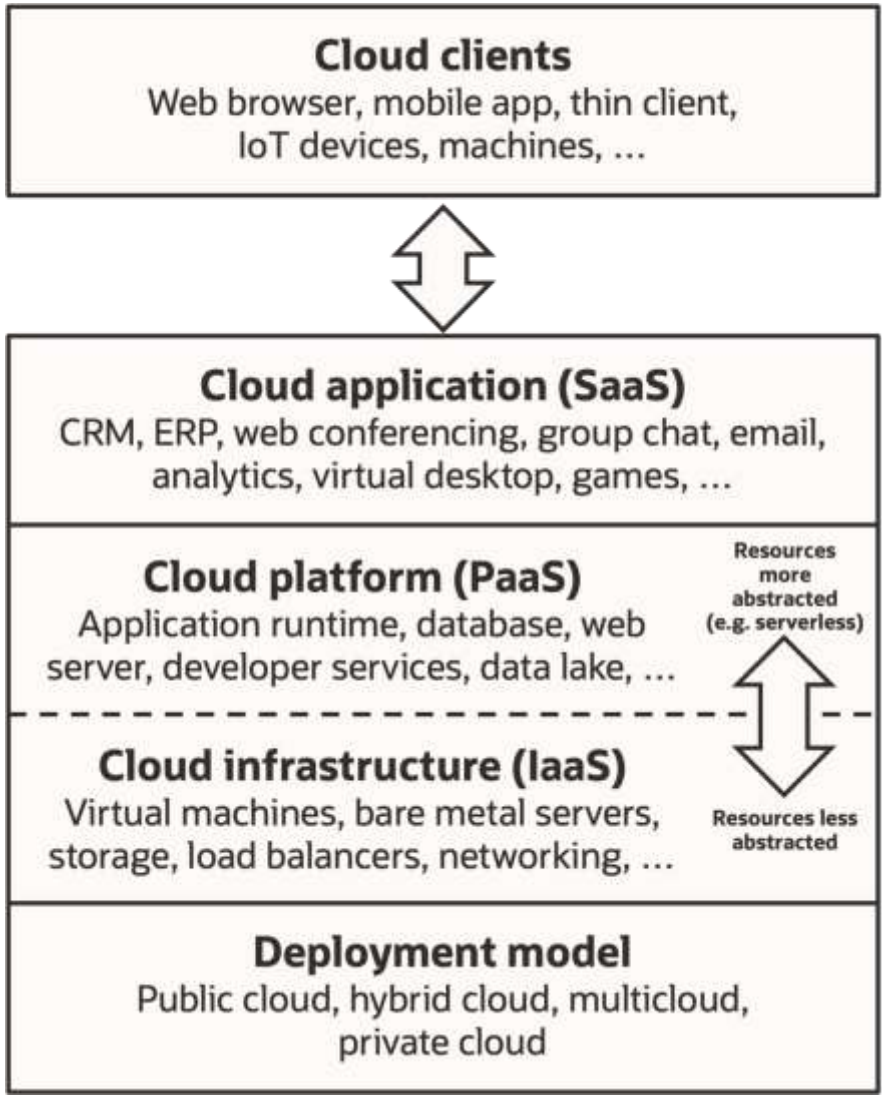
Cloud Delivery Models

Our Use Case

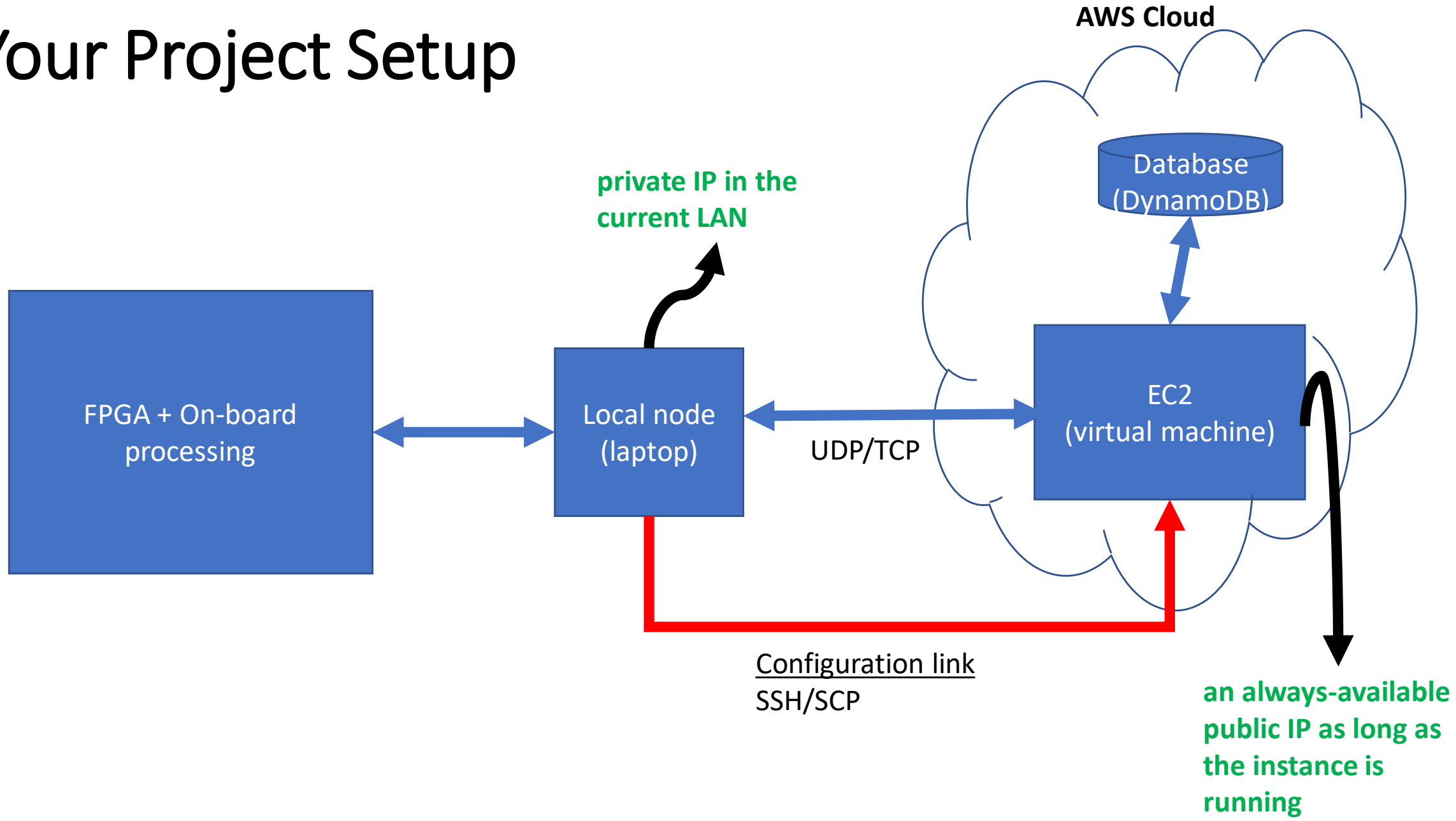
IaaS: Infrastructure as a Service

The consumer is able to deploy and run arbitrary software, which can include operating systems and applications.

The consumer does not manage or control the underlying cloud infrastructure (hardware) but has **control** over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).



Your Project Setup



Amazon’s Elastic Compute Cloud (EC2)

- **Amazon Elastic Compute Cloud (EC2)** is a part of Amazon.com's cloud-computing platform: Amazon Web Services (**AWS**)
  - lends virtual computers on the cloud to configure and run your apps
  - the virtual computer is typically running a Linux distribution (e.g., Ubuntu)

- Using EC2 an example of IaaS
  - We can choose an OS, storage, computing resources, networking configurations, etc.

- Unlike the lower tiers (Edge and Fog), we can rely on EC2 to be **elastic, i.e.** to grow according to our application’s needs

EC2 Configuration Steps (Lab 4)

Using the web-based console on your AWS account

- Choose **a machine image**: processor, storage, operating system, etc.
- Instance type (**computational power**): no of processors, amount of memory, networking capacity, etc. (Default options on the free-tier suffice)
- Configure **security group**
  - What type of connections are permitted? TCP, UDP, SSH only?
  - What port range is permissible?
  - Are only specific IP addresses allowed to connect, or anyone can connect?
- EC2 is one of the AWS services
  - To communicate with other services, it must do so through a predefined role (IAM role)
  - an **IAM role** is an identity you create in your account that provides specific permissions.
    - E.g. the *DynamoDBAccessforEC2* role in your lab gives your EC2 instance to a DynamoDB database.
- The EC2 instance is **always-running, with a public IP**, unless you explicitly shut it down

Basic Software Installations on EC2 (Lab 4)

You can install any software tools on your EC2 instance that are required by your app to run on Linux:

Some basic tools needed for Lab 4:

- Python
- Pip
- Python packages using pip
- **Boto3**: AWS's Python API to interface with services like DynamoDB, S3 etc.
- g++, etc.

Client-Server Communication

Example: A simple TCP Client and a Server on EC2

Server's public IP and port number specifies its connection end-point

```
from socket import *
serverIP = "16.171.58.69"   # Public IP of EC2 instance
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverIP, serverPort))
message = input("Enter a string: ")
clientSocket.send(message.encode())
reply = clientSocket.recv(2048)
print("Server response:", reply.decode())
clientSocket.close()
```

IPv4 TCP socket and 3-way handshake to connect

Blocking calls: **send** and **recv**, to send and receive data

TCP Client on a Local Node

The welcoming socket: to receive incoming clients / the same socket for all clients

```
from socket import *
serverPort = 12000
welcomeSocket = socket(AF_INET, SOCK_STREAM)
welcomeSocket.bind(('0.0.0.0', serverPort))
welcomeSocket.listen(1)
print("TCP server running and listening on
while True:
    connectionSocket, clientAddress = welcomeSocket.accept()
    message = connectionSocket.recv(2048)
    text = message.decode()
    if text.isupper():
        reply = "ALL CAPS"
    else:
        reply = "NOT ALL CAPS"
    connectionSocket.send(reply.encode())
    connectionSocket.close()
```

Listen on all network interfaces on this machine

Per-client connection socket

Blocking calls: **recv** and **send** to/from this particular client

TCP Server on EC2

Some Issues to consider:

- We're assuming short messages of known size range (TCP doesn't guarantee message boundaries)
- The server can only handle one client at a time
- Raw TCP sockets wouldn't work from within browsers, or if you wished to use any web-API; you'd need an application layer protocol, e.g., HTTP

Your server may run as a daemon on your EC2 instance.

Handling Multiple Clients Concurrently

The Multithreading Approach

The modified TCP server with multi-threading

Page 1 of 2

```
from socket import *
import threading

serverPort = 12000
welcomeSocket = socket(AF_INET, SOCK_STREAM)
welcomeSocket.bind(('0.0.0.0', serverPort))
welcomeSocket.listen(5)  # Increased queue size
print("TCP server running and listening on port", serverPort)

def serve_client(connectionSocket, clientAddress):
    """Handle individual client in separate thread"""
    try:
        message = connectionSocket.recv(2048)
        text = message.decode()
        if text.isupper():
            reply = "ALL CAPS"
        else:
            reply = "NOT ALL CAPS"
        connectionSocket.send(reply.encode())
    finally:
        connectionSocket.close()
        print(f"Connection closed for {clientAddress}")
```

Up to 5 clients can wait in the connection queue (could be set to a reasonable upper bound)

Page 2 of 2

```
while True:
    connectionSocket, clientAddress = welcomeSocket.accept()
    print(f"New connection from {cl

    # Create and start new thread f
    client_thread = threading.Threa
        target=serve_client,
        args=(connectionSocket, cli
    )
    client_thread.start()
    # Main thread immediately loops
```

Serve this client in a separate thread

Using **pthread** under-the-hood (UNIX based system)
threading.Thread → pthread_create()
threading.Lock → pthread_mutex_t
threading.Semaphore → POSIX semaphores

The client thread function: would require synchronization i
client threads are accessing/modifying shared data

The usual synchronization tools can be used.
For example:

Mutexes:

```
from threading import Lock
lock = Lock()
with lock:  # acquire and auto-release
   # critical section - only one thread at a time
```

Semaphores:

```
from threading import Semaphore
sem = Semaphore(3)  # allow 3 threads max
with sem:  # one of 3 slots
   # limited concurrent access
```

Similarly, Condition Variables, Barriers, etc.

Concurrency without Multithreading?

Multithreading Pros and Cons

Pros:

- **Intuitive**: Each client gets mapped to one thread, familiar sequential code
- **True parallelism** for I/O operations (threads can wait independently), multiple cores can be exploited
- **Preemptive scheduling**: OS handles switching between threads automatically
- **Works well** for blocking I/O (like our recv() calls)

Cons:

- **Resource overhead**: Each thread uses memory (~8MB stack per thread on Linux)
- **Scalability limits**: Thousands of threads → performance degradation
- **Context switching cost**: OS must save/restore thread state
- **Synchronization complexity**: Mishandling of locks, semaphores → potential deadlocks, race conditions

An alternative approach

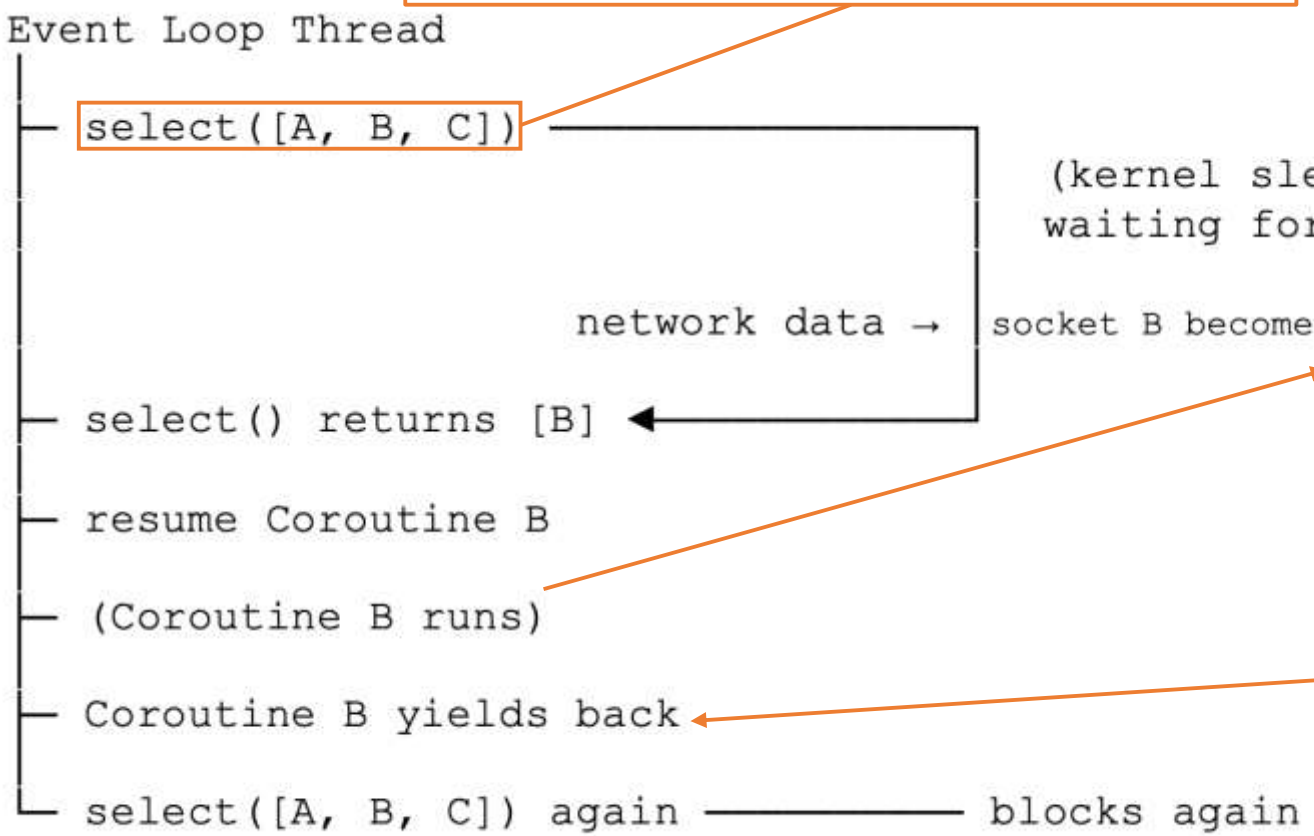
- Implement **user-level cooperative scheduling**
- An **event-loop (a single thread) schedules different coroutines** based on events, e.g., a socket can be read from
- Only the event loop potentially blocks in the kernel waiting for I/O readiness for any coroutine
- Coroutines use **non-blocking sockets** asynchronously when scheduled by the event loop
- Specifically:
  - When a coroutine needs to send/recv on a socket it **yields control to the event loop**
  - The **event loop uses a system call: select/epoll to monitor multiple sockets**, resumes coroutines when their I/O is ready, allowing non-blocking operations

Handling Multiple Clients Concurrently

Using Event-Based Concurrency

Kernel system call via libc (under-the-hood)
works with file descriptors
(**$ man select**)

Timeline




```
# Event loop pseudocode (partial)
ready_sockets = select([socket_A, socket_B, socket_C])   # blocks here

# select returns: [socket_B]  ← socket_B is ready

# Resume coroutine B
data = socket_B.recv(2048)   # safe to read (non-blocking)
process(data)
```

```
Coroutine B
├─ recv(sock_B) # non-blocking; makes progress immediately
├─ process(data)
├─ needs more I/O
└─ yield → event loop
```

(on a non—blocking socket the kernel would let B know if socket not ready)

Event–Based Concurrency with `asyncio`

Page 1 of 2

```
import asyncio

async def handle_client(reader, writer):
    addr = writer.get_extra_info('peername')
    print(f"New connection from {addr}")

    data = await reader.read(2048)
    text = data.decode()

    if text.isupper():
        reply = "ALL CAPS"
    else:
        reply = "NOT ALL CAPS"

    writer.write(reply.encode())
    await writer.drain()

    writer.close()
    await writer.wait_closed()
```

Page 2 of 2

```
async def main():
    server = await asyncio.start_server(
        handle_client, '0.0.0.0', 12000
    )

    print("TCP server listening on port 12000")

    await server.serve_forever()

asyncio.run(main())
```

Creates a listening (welcome) socket and registers it with the event loop

Suspends while waiting for incoming connections (yields to event loop)

The main event loop

Suspends if no data available (yields to event loop)

Suspends until send buffer has space.

Client-Server Communication

Event-Based Concurrency, Multithreading, Multiprocessing

- A real server would use some combination of **event-based concurrency**, **multithreading**, and **multiprocessing**

- **Event-based concurrency (OS mechanisms: epoll, select, framework: asyncio)**
  - handles many sockets efficiently
  - **good for I/O bound applications**, such as a server
  - avoids threading overhead
  - makes it **easier to avoid synchronization overhead**
  - great for I/O-bound work
  - several web frameworks use it

- **Multithreading**
  - overlaps blocking or CPU work
  - good for legacy/blocking libraries
  - **used for background tasks**

- **Multiprocessing**
  - True parallelism, isolation, security
  - Scales well for **CPU-bound work**

Part II

Database

- A database is an organized collection of data
  - Structured so it can be efficiently searched, updated, and managed
- A database is usually accessed through a Database Management System (DBMS)
  - e.g., SQLite, MySQL, PostgreSQL, DynamoDB (a DBMS provides a controlled interface between programs and stored data
- Why not just files?
  - Raw files (e.g. text, CSV, binary) require programs to manually handle reading, writing, parsing, and consistency.
  - Databases store data in structured tables or collections and provide query-based access, handling storage, consistency, and access automatically.

Database Options

1. SQL databases (covered next week)

- Well defined **schema**
  - the database is modeled as a set of joinable tables with pre-defined columns
- Entities are stored in tables, **managed by an RDBMS**
- **Tables can be joined** and using a powerful and flexible querying language: **SQL**
- Data follows are **rigid structure** and is mostly non-redundant
- Changing/updating the structure of data involves change in schema and **costly database alterations**.
- There is a DBA – a database administrator -- between the application developer and the database, responsible for defining and altering the schema if needed.
  - **Using SQL's DDL**

- **Naturally ideal for vertical scaling, i.e.,** increasing the database's capacity by adding more resources (CPU, memory, storage) to a single machine.

Database Options

2. **NoSQL databases**

- Data is not stored in relations (although the tabular form exists)
- There **is no predefined schema** (a flexible schema is maintained)
  - Data is not highly structured: unstructured or semi-structured data may be stored.
- Flexibility of schema speeds up development, as the developer has direct control over the structure of the database: "***data becomes like code"***.
- NoSQL databases could be: document stores, key-value stores, column stores or graph-base.
  - DynamoDB is a key-value store – meaning, they use hashing as their principal storage strategy.

- **Naturally ideal for horizontal scaling**, i.e., increasing the database's capacity by adding more machines and distributing data across them.

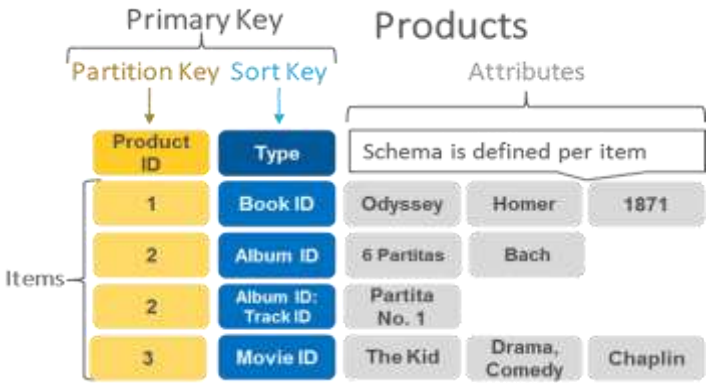
Horizontal Scaling of Databases

- Data needs to be **divided and distributed across multiple partitions**
- A partition is a logical single chunk of storage managed by the cloud service provider
- When queried for data, the system needs to know which partition stores what part of the data.
- The system should be able to:
  - **map each individual record to a partition**
  - **search for records in a data structure** within that partition
- This is the operation of a typical NoSQL database called a Key-Value Store

DynamoDB:

An Example of a NoSQL Key-Value Store

- A simple type of NoSQL database
- Stores data **as a collection of key-value pairs**
  - key serves as a unique identifier for the record.
- Highly partitionable, allows horizontal scaling

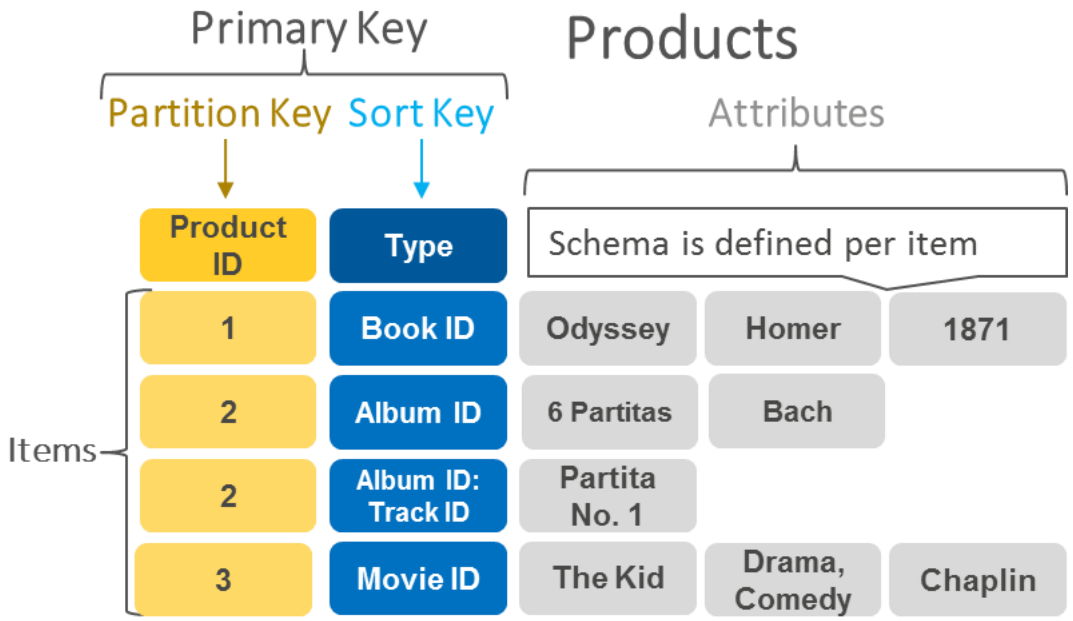


An example table in DynamoDB

Amazon's DynamoDB, a key-value store

- Every 'row' in a 'table' is defined by a composite primary key:
  Primary key = (Partition key, Sort Key)

- A record with key (pk, sk), is assigned the partition *h*(pk), and sorted at *s*(sk), within that partition.
  - *h* is an internally maintained hash function.
  - *s* is an ordering function of an ordered data structure such as a height-balanced tree



A partition is an allocation of storage for a table, backed by solid state drives (SSDs) and automatically replicated across multiple Availability Zones within an AWS Region.

BOOKMARK

Creating a Table in DynamoDB (Boto3 Python)

```
import boto3

def create_movie_table(dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

    table = dynamodb.create_table(
        TableName='Movies',
        KeySchema=[
            {
                'AttributeName': 'year',
                'KeyType': 'HASH'  # Partition key
            },
            {
                'AttributeName': 'title',
                'KeyType': 'RANGE'  # Sort key
            }
        ],
        AttributeDefinitions=[
            {
                'AttributeName': 'year',
                'AttributeType': 'N'
            },
            {
                'AttributeName': 'title',
                'AttributeType': 'S'
            },

        ],
        ProvisionedThroughput={
            'ReadCapacityUnits': 10,
            'WriteCapacityUnits': 10
        }
    )
    return table

if __name__ == '__main__':
    movie_table = create_movie_table()
    print("Table status:", movie_table.table_status)
```

Create DynamoDB service handle (selects service + region)

Use year as the partition key

Use title as the sort key

Two columns in this table:
year: a number
title: a string

Up to 10 reads per second
Up to 10 writes per second

Inserting an Item into a Table

```
from pprint import pprint
import boto3

def put_movie(title, year, plot, rating, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

    table = dynamodb.Table('Movies')
    response = table.put_item(
       Item={
            'year': year,
            'title': title,
            'info': {
                'plot': plot,
                'rating': rating
            }
        }
    )
    return response

if __name__ == '__main__':
    movie_resp = put_movie("The Big New Movie", 2015,
                           "Nothing happens at all.", 0)
    print("Put movie succeeded:")
    pprint(movie_resp, sort_dicts=False)
```

Get a handle/reference to the Movies table

Send a write request to insert an item into the table

Nested attribute for additional movie data.
These attributes are created automatically by this request.
DynamoDB has no fixed columns (except keys).

The database does not have a predefined schema (structure)

The table structure can change,
in a manner similar to objects in code.

Reading an Item From a Table

```
def get_movie(title, year, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

    table = dynamodb.Table('Movies')

    try:
        response = table.get_item(Key={'year': year, 'title': title})
    except ClientError as e:
        print(e.response['Error']['Message'])
    else:
        return response['Item']


if __name__ == '__main__':
    movie = get_movie("The Big New Movie", 2015,)
    if movie:
        print("Get movie succeeded:")
        pprint(movie, sort_dicts=False)
```

Fetch item using its primary key (partition + sort key)

Only primary keys support fast reads/writes
(non-key attributes cannot be queried directly)

Conditionally Deleting Items

```
def delete_underrated_movie(title, year, rating, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

    table = dynamodb.Table('Movies')

    try:
        response = table.delete_item(
            Key={
                'year': year,
                'title': title
            },
            ConditionExpression="info.rating <= :val",
            ExpressionAttributeValues={
                ":val": Decimal(rating)
            }
        )
    except ClientError as e:
        if e.response['Error']['Code'] == "ConditionalCheckFailedException":
            print(e.response['Error']['Message'])
        else:
            raise
    else:
        return response


if __name__ == '__main__':
    print("Attempting a conditional delete...")
    delete_response = delete_underrated_movie("The Big New Movie", 2015, 10)
    if delete_response:
        print("Delete movie succeeded:")
        pprint(delete_response)
```

Identify the item using its primary key

Delete only if the item's rating is less than or equal to the threshold

DynamoDB needs a description of the expression to perform it remotely on its servers

DynamoDB performs these operations **atomically:**

Find item by key
Check condition on server
If true → delete
If false → abort

General Queries

Get all movies released in a given year

```
import boto3
from boto3.dynamodb.conditions import Key

def query_movies(year, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name="us-east-1")

    table = dynamodb.Table('Movies')
    response = table.query(
        KeyConditionExpression=Key('year').eq(year)
    )
    return response['Items']


if __name__ == '__main__':
    query_year = 1985
    print(f"Movies from {query_year}")
    movies = query_movies(query_year)
    for movie in movies:
        print(movie['year'], ":", movie['title'])
```

Import helper to build key-based query expressions

Use the Key convenience class to build a key condition expression (the expression is evaluated on the DynamoDB server)

route to the correct partition and return matching items

General Queries

Get all movies released in a given year, with titles within a range

```
def query_and_project_movies(year, title_range, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name="us-east-1")

    table = dynamodb.Table('Movies')
    print(f"Get year, title, genres, and lead actor")

    # Expression attribute names can only reference items in the projection expression.
    response = table.query(
        ProjectionExpression="#yr, title, info.genres, info.actors[0]",
        ExpressionAttributeNames={"#yr": "year"},
        KeyConditionExpression=
            Key('year').eq(year) & Key('title').between(title_range[0], title_range[1])
    )
    return response['Items']

if __name__ == '__main__':
    query_year = 1992
    query_range = ('A', 'L')
    print(f"Get movies from {query_year} with titles from "
          f"{query_range[0]} to {query_range[1]}")
    movies = query_and_project_movies(query_year, query_range)
    for movie in movies:
        print(f"\n{movie['year']} : {movie['title']}")
        pprint(movie['info'])
```

Return only selected attributes (partial item fetch, reduces data transfer)

The alias (#yr) is not required in this specific case, but shown as an example for alias creation for safe parsing (to avoid potential use of reserved keywords).

Querying Across Multiple Partitions

Print all movies between (and including) the years 1950 and 1959.

In a relational database this would be a straight-forward condition in a WHERE clause, as the entire table is available in one place.

In a key store, each key (i.e., the year) is mapped to a different partition, therefore all partitions in the range need to be **scanned**

Scan Queries

```
def scan_movies(year_range, display_movies, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

    table = dynamodb.Table('Movies')

    #scan and get the first page of results
    response = table.scan(FilterExpression=Key('year').between(year_range[0],
year_range[1]));
    data = response['Items']
    display_movies(data)

    #continue while there are more pages of results
    while 'LastEvaluatedKey' in response:
        response = table.scan(FilterExpression=Key('year').between(year_range[0],
year_range[1]), ExclusiveStartKey=response['LastEvaluatedKey'])
        data.extend(response['Items'])
        display_movies(data)

    return data

if __name__ == '__main__':
    def print_movies(movies):
        for movie in movies:
            #print(f"\n{movie['year']} : {movie['title']}")
            #pprint(movie['info'])
            pprint(movie)

    query_range = (1950, 1959)
    print(f"Scanning for movies released from {query_range[0]} to {query_range[1]}...")
    scan_movies(query_range, print_movies)
```

Perform a Scan operation (checks every item in the table)

Apply server-side filter after scanning items (not key-based lookup)

Full table traversal (slow, O(N), not partition targeted)

Retrieve first page of matching items

Continue scanning additional pages using DynamoDB pagination until all matching items are retrieved.

Stop when the response has no LastEvaluatedKey (pagination cursor)

More Complex Querying and Table Joining

- Possible through scanning and additional code, but slow.
- Instead of joins, design tables around your query patterns.
  - Accept redundancy and controlled data duplication.
- DynamoDB provides features like Global Secondary Indexes (GSIs), which maintain internally duplicated data organized by different partition keys.
- In contrast, relational databases minimize redundancy and rely on joins and a powerful query language to answer complex queries.

Possible Database Use-cases for Your System

- **General application data**
  - Store core application data such as users, content, or domain-specific entities needed for normal operation
- **Data collection and history**
  - Store measurements, summaries, and historical records for monitoring, analysis, display through web interface etc.
- **System recovery and restart state**
  - Store checkpoints, last-known state, and progress markers so the system can resume correctly after crashes, restarts, or network failures
- **Configuration, preferences, and personalization**
  - Maintain system rules, user preferences, and adjustable parameters that shape application behaviour
- **Event tracking and notifications**
  - Record notable events (e.g. anomalies, milestones, interactions) and drive alerts or user-visible actions
- **Individual User activity and Data**
  - Track how the system is used over time to understand behaviour, performance, and trends
- **Shared state and coordination**
  - Maintain common state across components, users, or sessions to enable consistent system behaviour