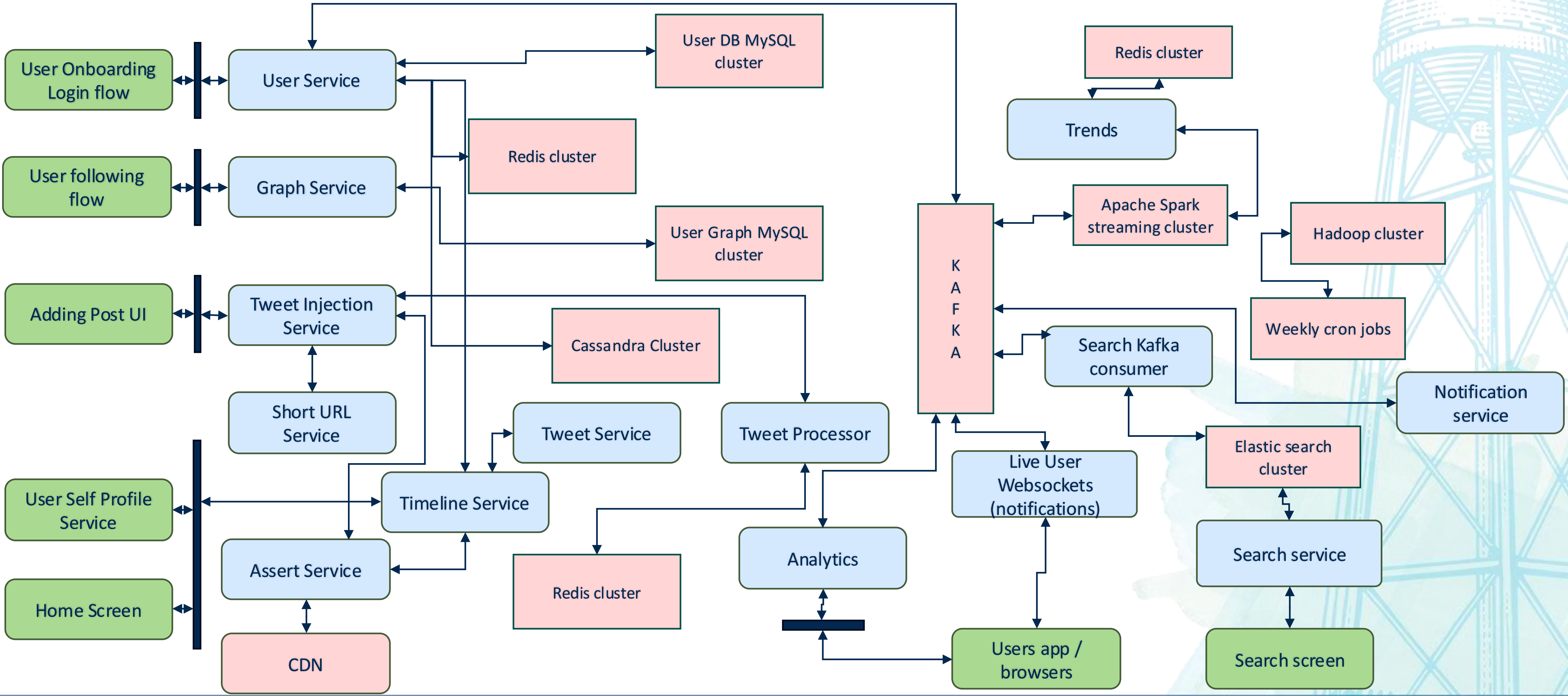
# Modern software architectures

Tapti Palit

# Modern software architecture




UCDAVIS

# Outline

- Microservice architecture and communication
- Event-driven architecture with Kafka
- Kubernetes

# Microservices outline

- Monolithic applications
- Microservices and decentralized data
- Data model and storage engine
  - Relational databases, log-structured merge trees (LSM), event logs (more in Kafka section), in-memory cache
- Communication styles
  - Text-based vs binary data exchange formats
  - Synchronous (RPC), asynchronous (MQs), publish-subscribe models

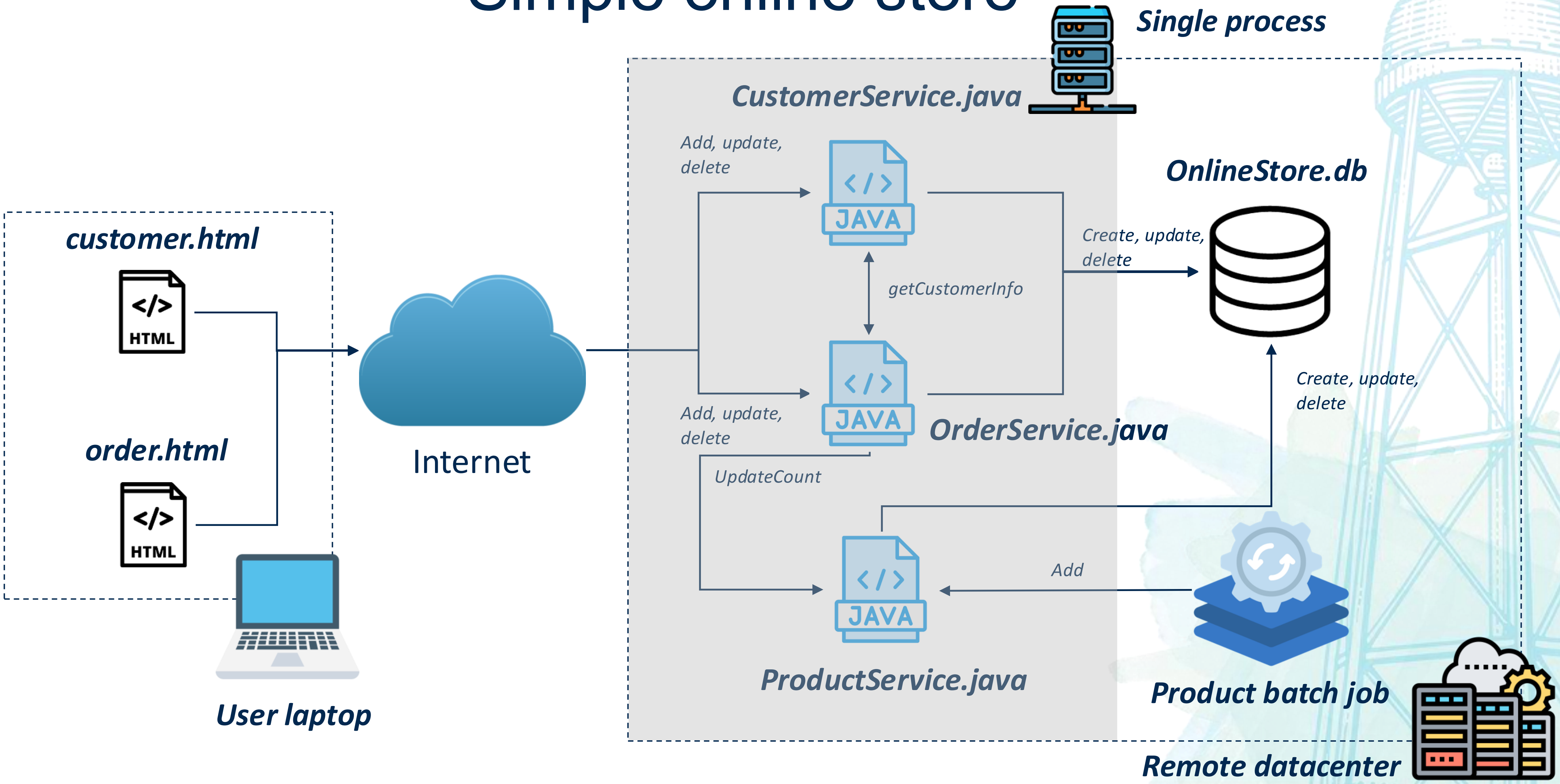
# All about the data!

- Modern software architecture is data-intensive
- Primary concerns
  - Who owns data?
  - How to optimally store data?
  - How is data shared?
  - How to limit overhead due to data-sharing?

# Monolithic software architecture

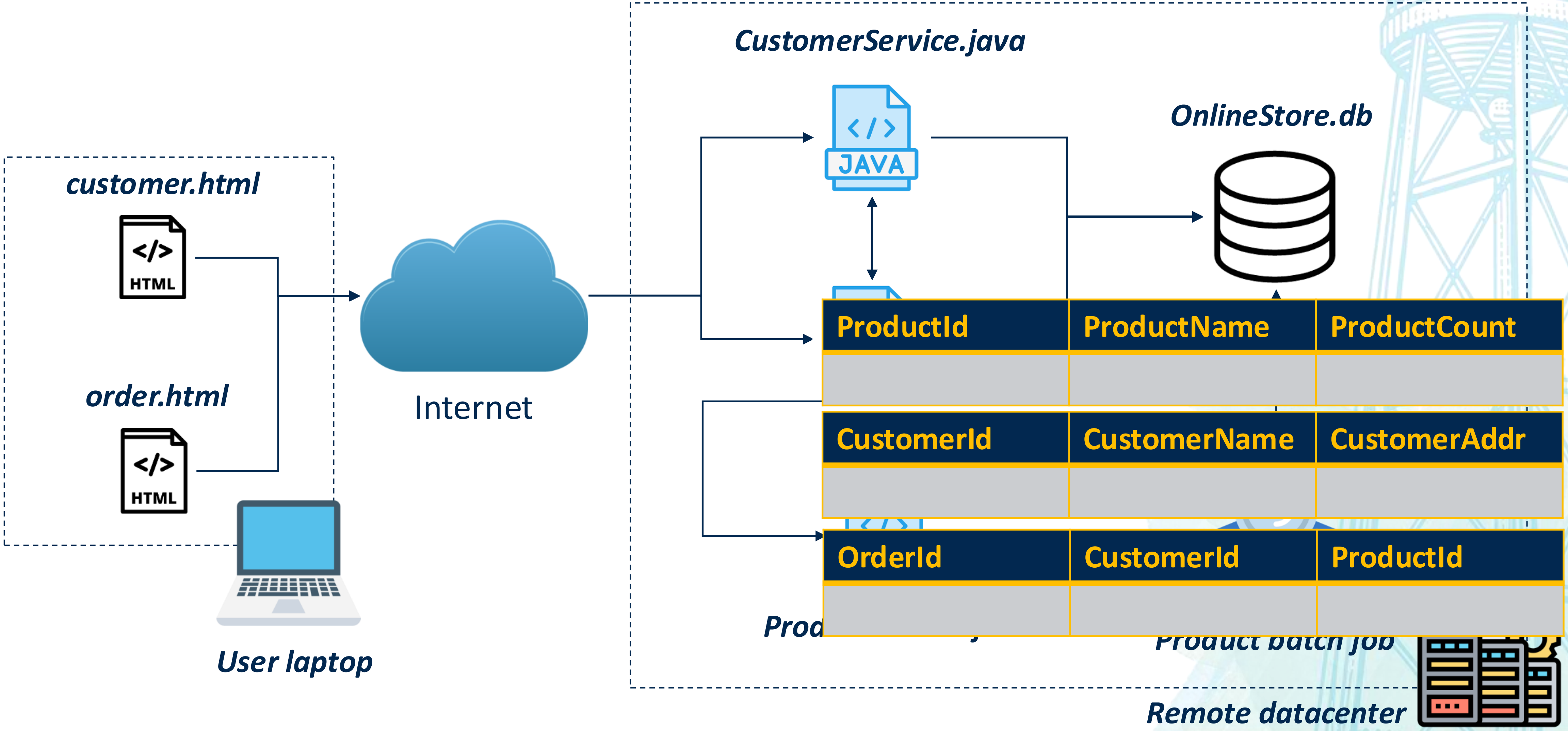
- Widely used in early, mid 2000s
- All software components are part of the same application
- All software components run on the same machine, in same process
- Typically developed in the same language stack

# Simple online store




UCDAVIS

# Simple online store




| ProductId | ProductName | ProductCount |
|---|---|---|
| | | |

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| | | |

| OrderId | CustomerId | ProductId |
|---|---|---|
| | | |

# Relational databases

- Consists of tables
- Each table contains a primary key
  - Database will not allow insertion of two records with same primary key

*Products tbl*

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

*Customer tbl*

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |

*Order tbl*

| OrderId | CustomerId | ProductId |
|---|---|---|
| 110001 | 9001 | 1001 |

# Foreign keys

- Relational databases maintain relations through foreign keys
- Foreign keys ***must refer*** to primary keys of other tables
  - Enforce referential integrity
- A table can contain one or more foreign keys

***Products tbl***

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

***Customer tbl***

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | ABCD |

***Order tbl***

| OrderId | CustomerId | ProductId |
|---|---|---|
| 110001 | 9001 | 1001 |

# Structured query language (SQL)

- SQL used to interface between application and database

- ```
  CREATE TABLE Products (
      ProductId INT PRIMARY KEY,
      ProductName VARCHAR(100) NOT NULL,
      ProductCount INT NOT NULL);
  ```

- ```
  INSERT INTO Products (ProductId, ProductName,
  ProductCount) VALUES (1001, 'ABC', 10);
  ```

# SQL joins

- Allows table joins
- For e.g. find order details for shipping, including product name, customer name, and address

*Products tbl*

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

*Customer tbl*

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | ABCD |

*Order tbl*

| OrderId | CustomerId | ProductId |
|---|---|---|
| 110001 | 9001 | 1001 |

# SQL joins

```sql
SELECT
    o.OrderId,
    p.ProductName,
    c.CustomerName,
    c.CustomerAddr
FROM Orders o
JOIN Products p
    ON o.ProductId = p.ProductId
JOIN Customers c
    ON o.CustomerId =
          c.CustomerId;
```

*Products tbl*

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

*Customer tbl*

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | ABCD |

*Order tbl*

| OrderId | CustomerId | ProductId |
|---|---|---|
| 10001 | 9001 | 1001 |

| o.OrderId | p.ProductName | c.CustomerName | c.CustomerAddr |
|---|---|---|---|
| 110001 | 9001 | XYZ | CDEF |

# Need for joins

- Imagine no support for joins
- Order information must contain product name, customer name, and customer address to ship the order

*Products tbl*

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

*Customer tbl*

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | ABCD |

*Order tbl*

| OrderId | ProductName | CustomerName | CustomerAddr |
|---|---|---|---|
| 110001 | ABC | XYZ | CDEF |

# Need for joins

- Lack of join support increases data duplication
- Data **denormalization**

***Products tbl***

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

***Customer tbl***

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | ABCD |

| OrderId | ProductName | CustomerName | CustomerAddr |
|---|---|---|---|
| 110001 | ABC | XYZ | CDEF |
| 110002 | ABC | XYZ | CDEF |
| 110003 | ABC | UVW | ABCD |

***Order tbl***

# Normalization

- Joins reduce data duplication and allow data normalization
- Database can build indexes to speed up joins

*Products tbl*

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

*Customer tbl*

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | ABCD |

*Order tbl*

| OrderId | ProductId | CustomerId |
|---|---|---|
| 110001 | 1001 | 9001 |
| 110002 | 1001 | 9001 |
| 110003 | 1001 | 9002 |

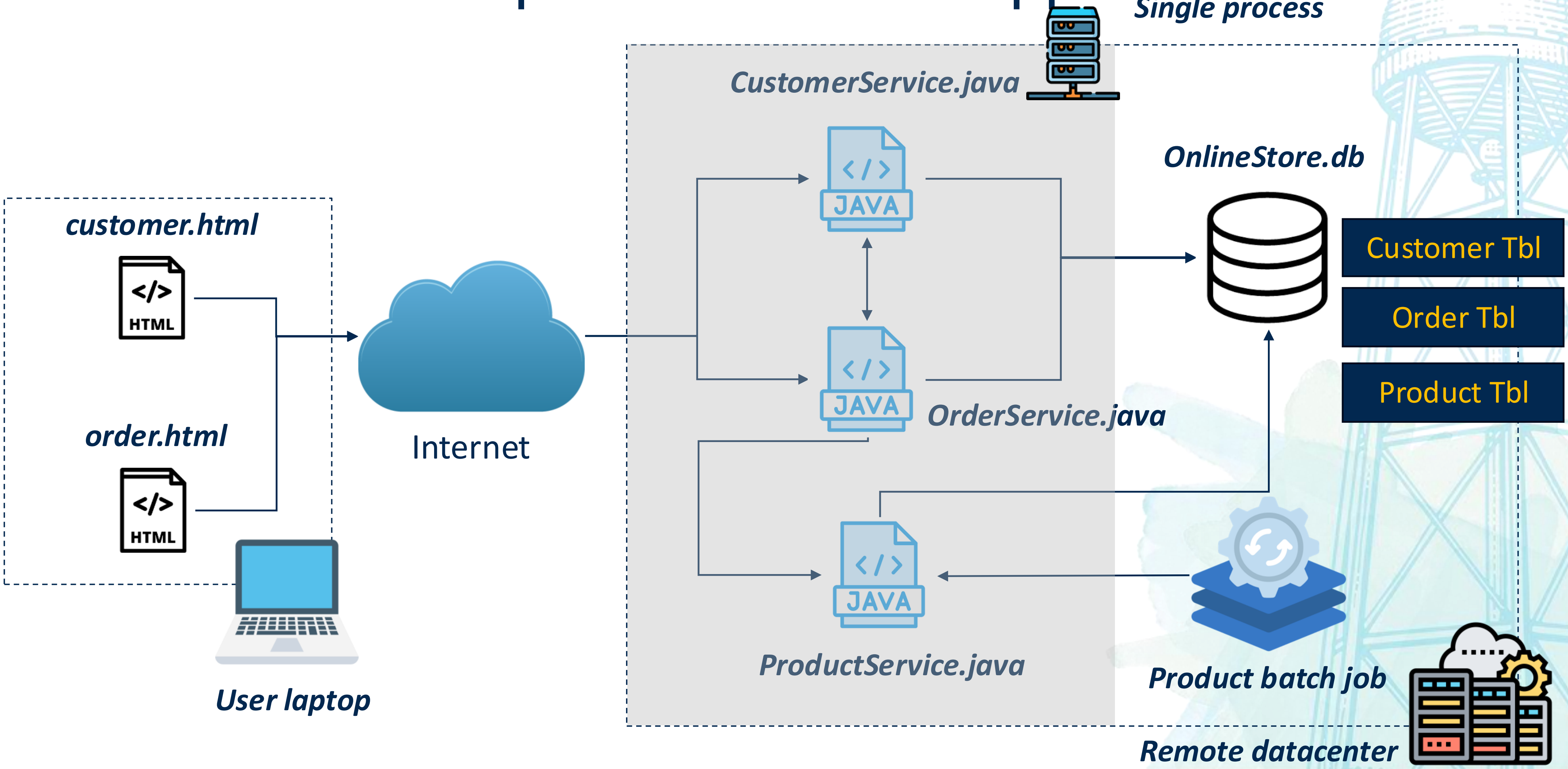
# Relational database implications

- Tighter coupling
  - E.g., Customer service cannot alter the Customers table without impacting the Orders service
- Fixed schema
  - E.g., Orders service must know about the schema Customers table

# SQL != relational databases

- Note: not quite specific to relational databases
- Apache Cassandra, Apache HBase, Apache Kafka + ksqlDB are not relational databases, but support SQL-like query language

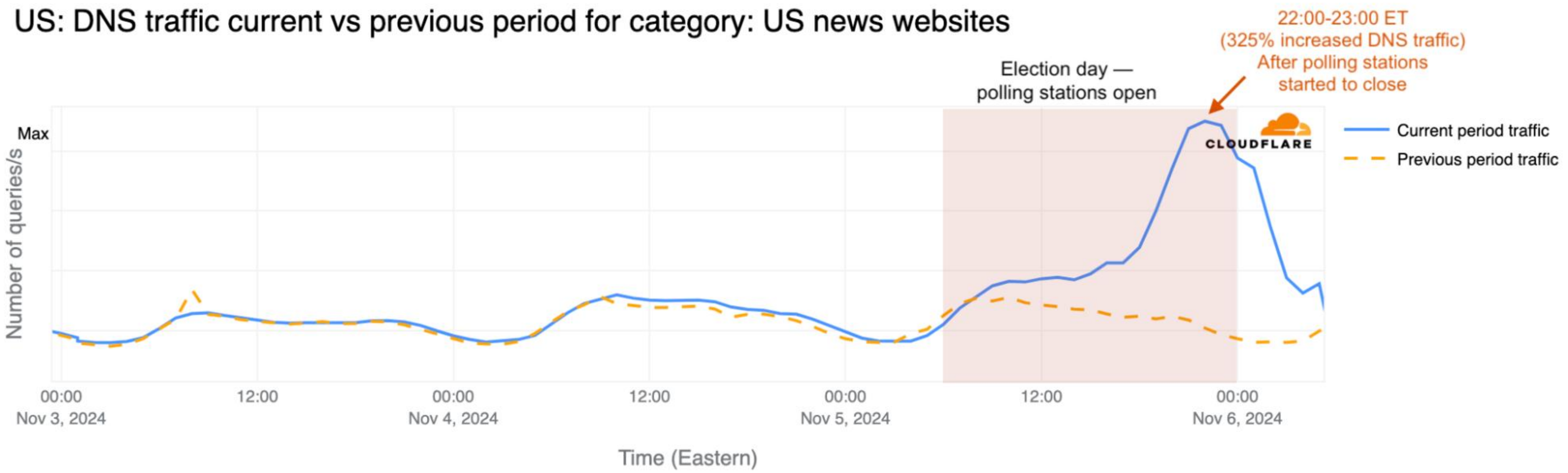
# Simple online store app



# Scalability

- Online traffic is variable
- Can spike due to planned events
  - Product launches
  - Political events

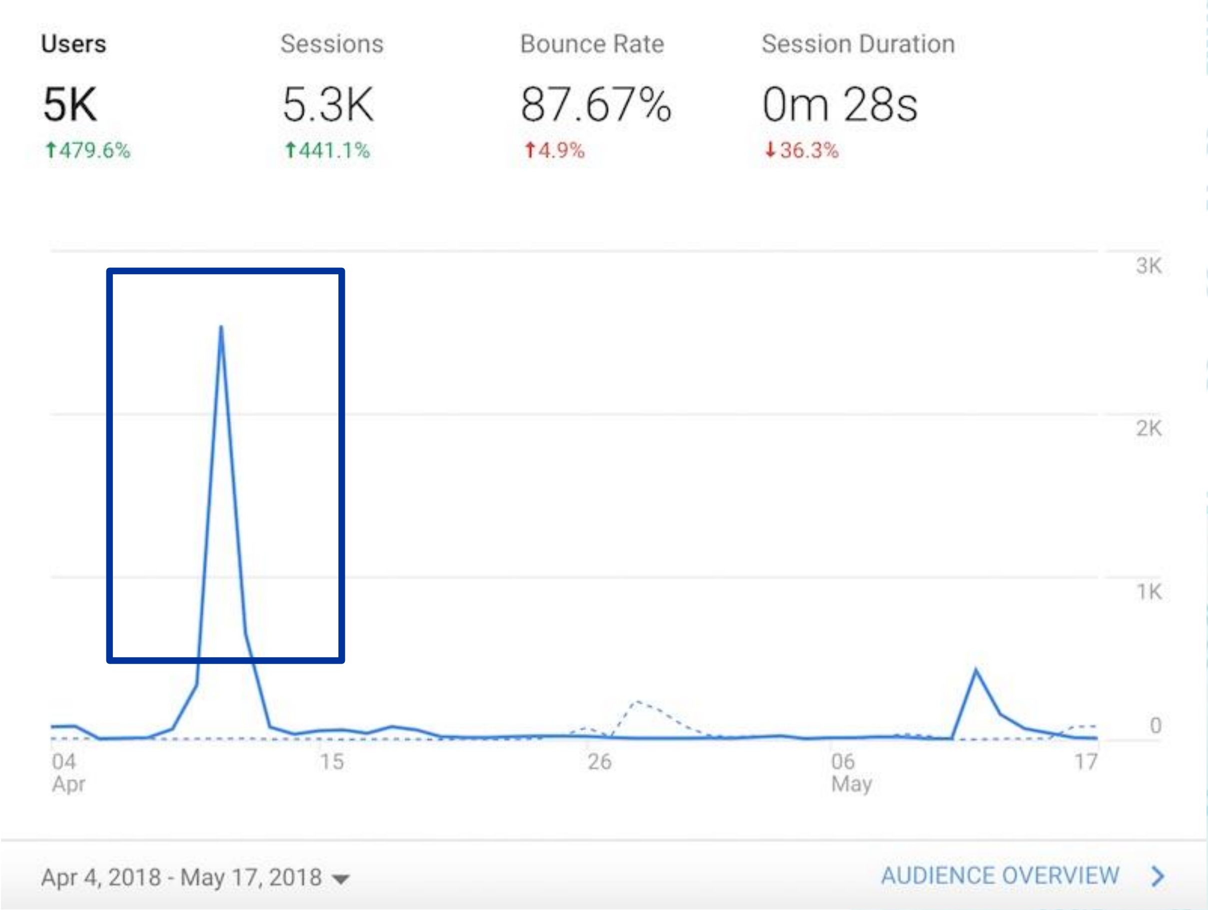
https://blog.cloudflare.com/exploring-internet-traffic-shifts-and-cyber-attacks-during-the-2024-us-election/



# Scalability

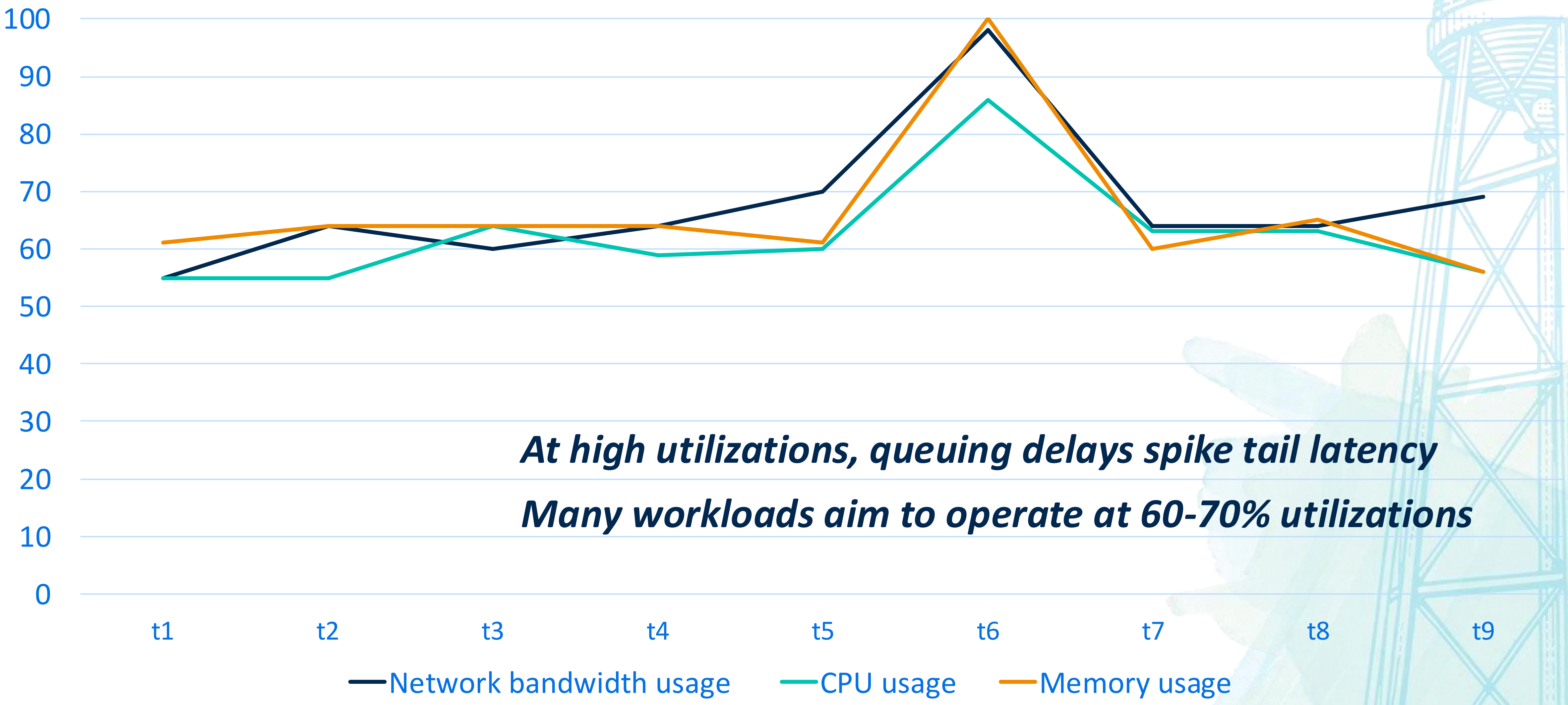
- Unplanned events
  - Blog post goes viral
- System design should *adapt* to handle such events

https://www.residualthoughts.com/2018/05/20/traffic-data-from-a-viral-post/




UCDAVIS

# Traffic spike to online store



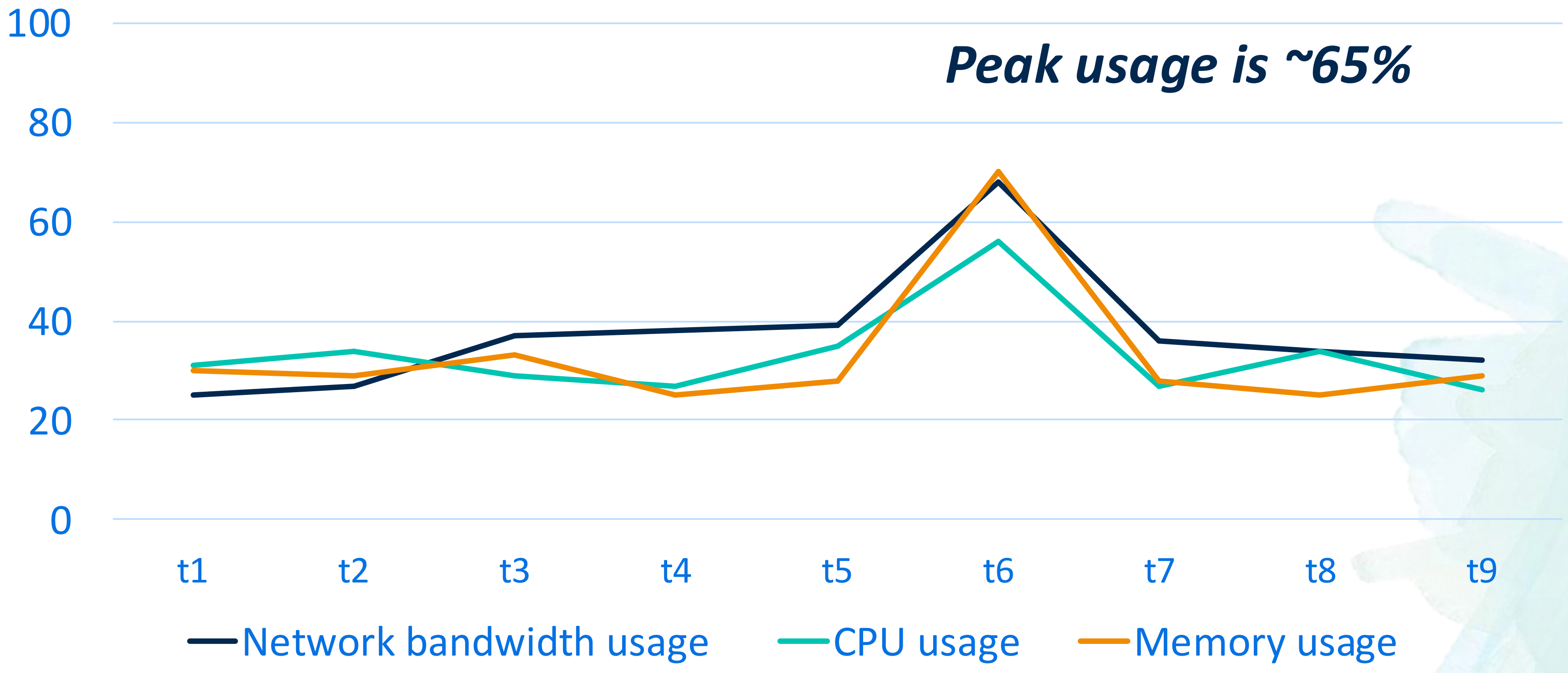

***At high utilizations, queuing delays spike tail latency***

***Many workloads aim to operate at 60-70% utilizations***

UCDAVIS

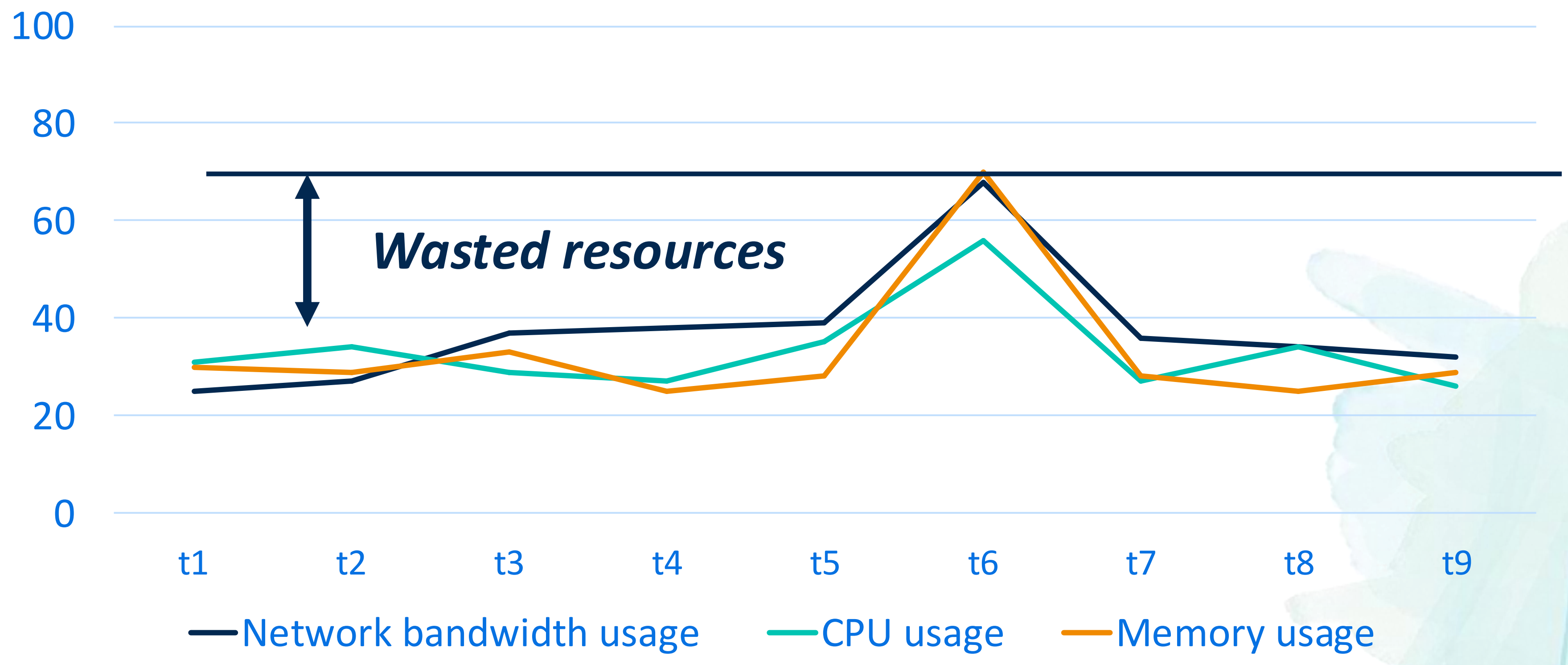
# Vertical scaling

- Use more powerful machines
- Add more CPUs, DRAM, network bandwidth



# Vertical scaling limitations

## Resource wastage when requests are not bursty

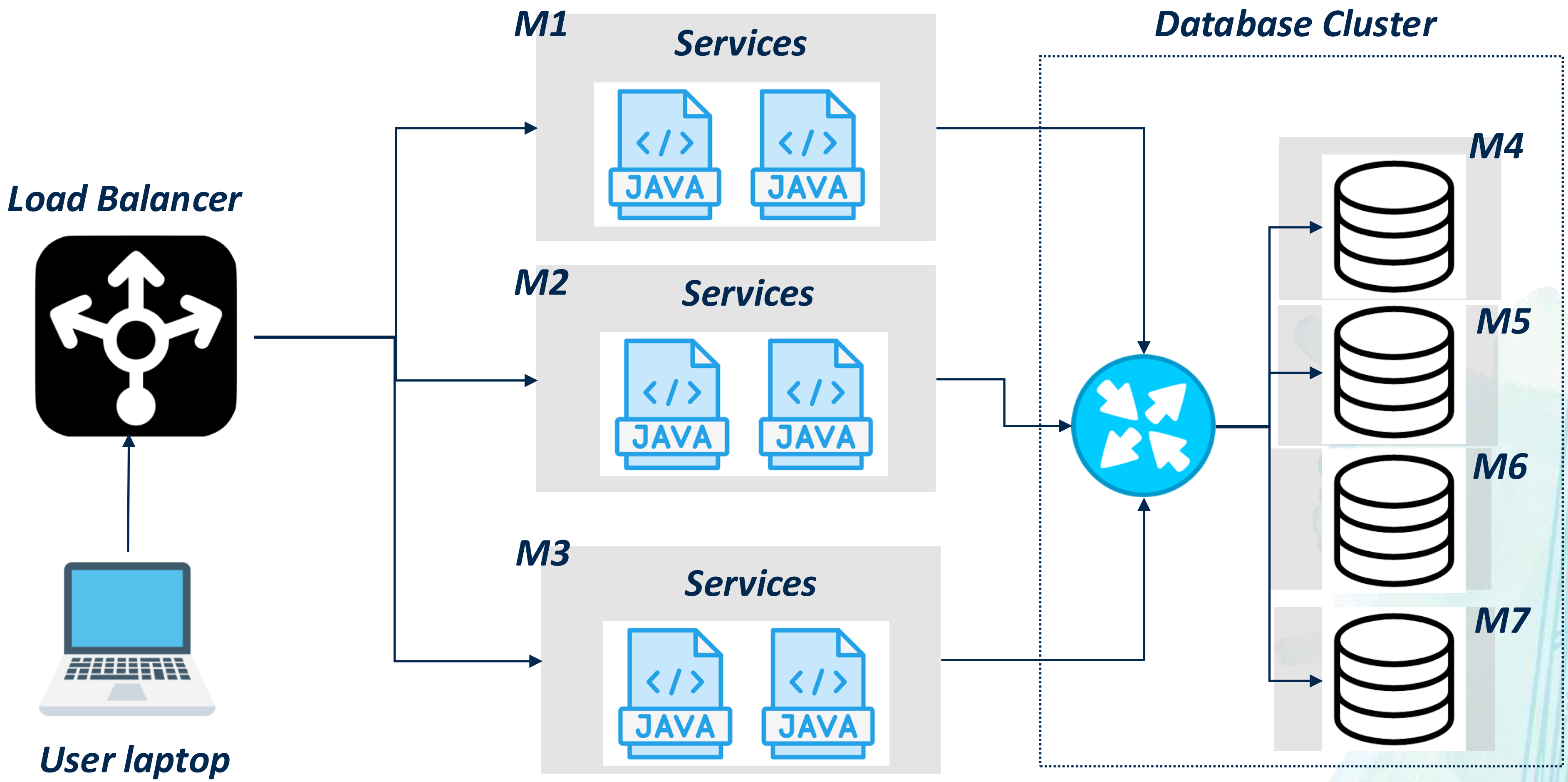


# Vertical scaling limitations

- Twitter has 200M-300M active users per day
- No single machine, no matter how powerful, can support such high loads
- Goal: autoscaling
  - **Dynamically** spawn new machines during **high loads**
  - Not possible using vertical scaling alone
  - More in Kubernetes module

# Horizontal scaling for monolithic apps

Add more machines and replicate application on each machine



# Load balancer

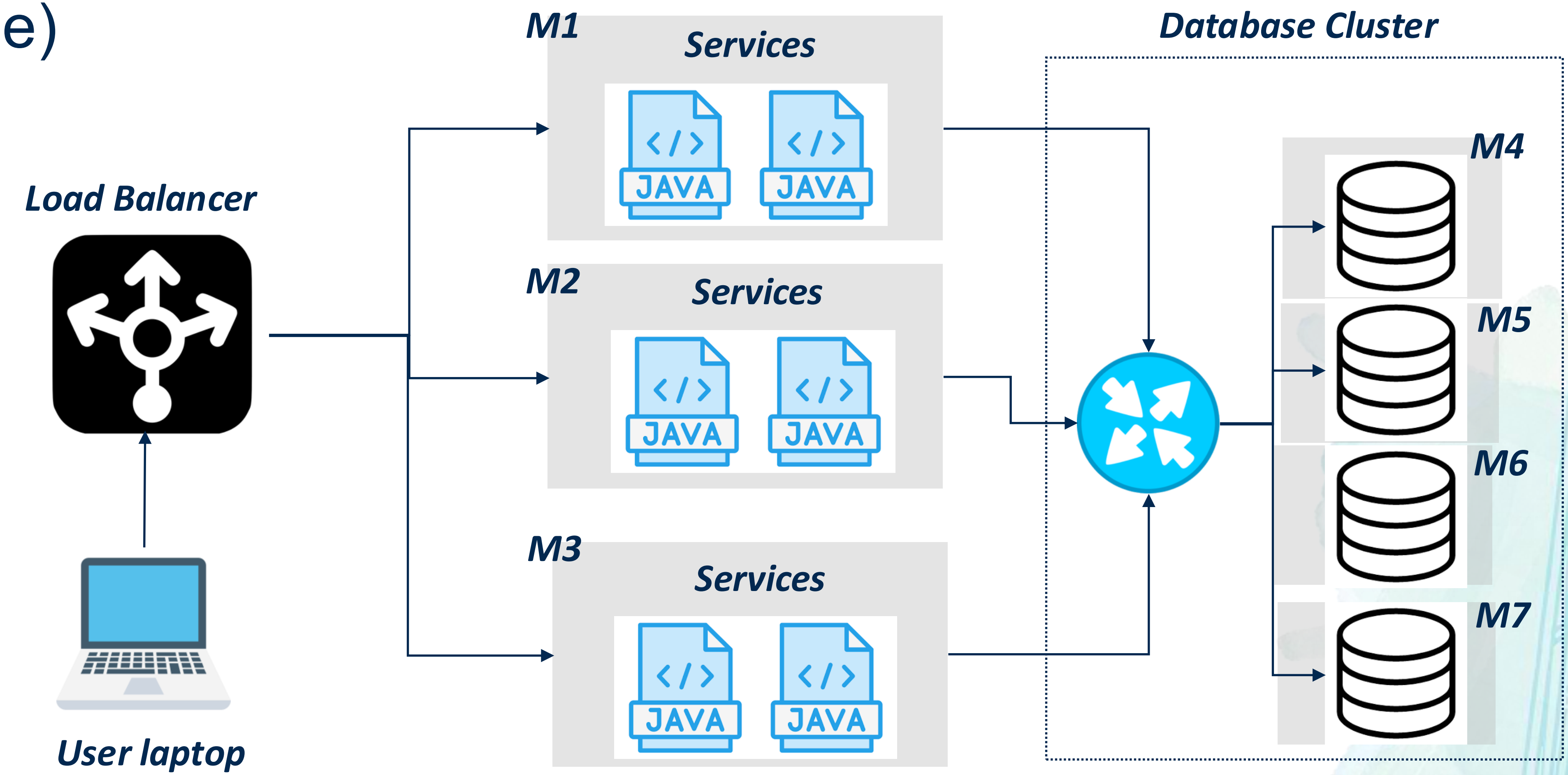
- Distributes incoming requests across multiple servers to improve scalability and availability
- Strategies
  - Round Robin – Sequentially routes requests across servers; simple but doesn’t account for server load
  - Least Connections – Directs traffic to the server with the fewest active connections; adapts well to uneven load
  - Least Response Time – Chooses the server with the fastest response time and fewest connections; performance-oriented
  - Random Policy – Selects servers randomly; useful in stateless, uniform environments
  - Weighted Distribution – Allocates requests based on server capacity (e.g., CPU power, memory)
- Each strategy has tradeoffs
- More details in Kubernetes module

# Horizonal scaling is unbounded

- Necessary for applications with high load (like Twitter/X)
  - Can add more machines to support more requests (theoretically unbounded performance)
- Better resource utilization

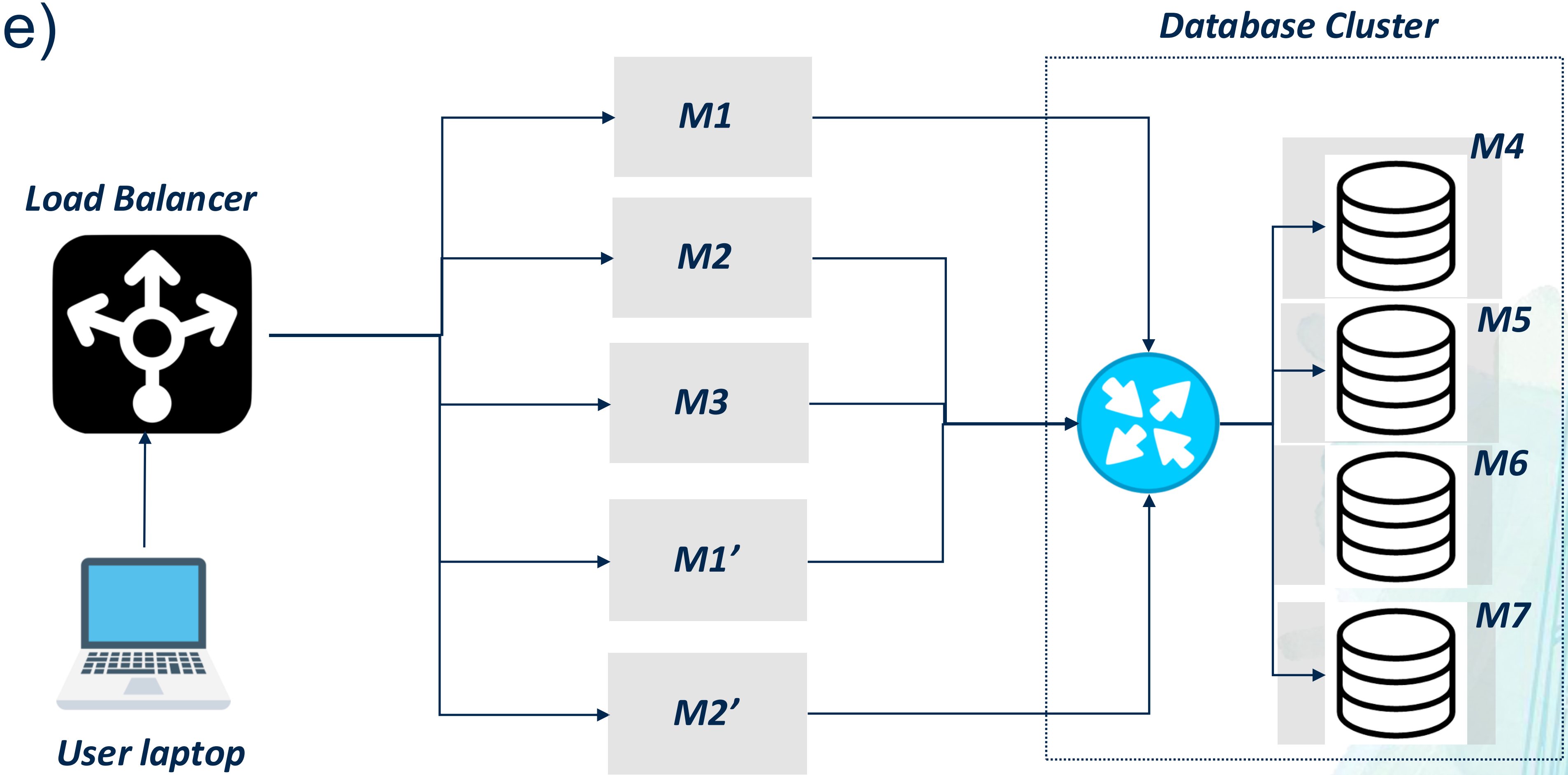
# Horizontal scaling is necessary for autoscaling

Auto-scale to more machines during traffic spike (more in Kubernetes module)



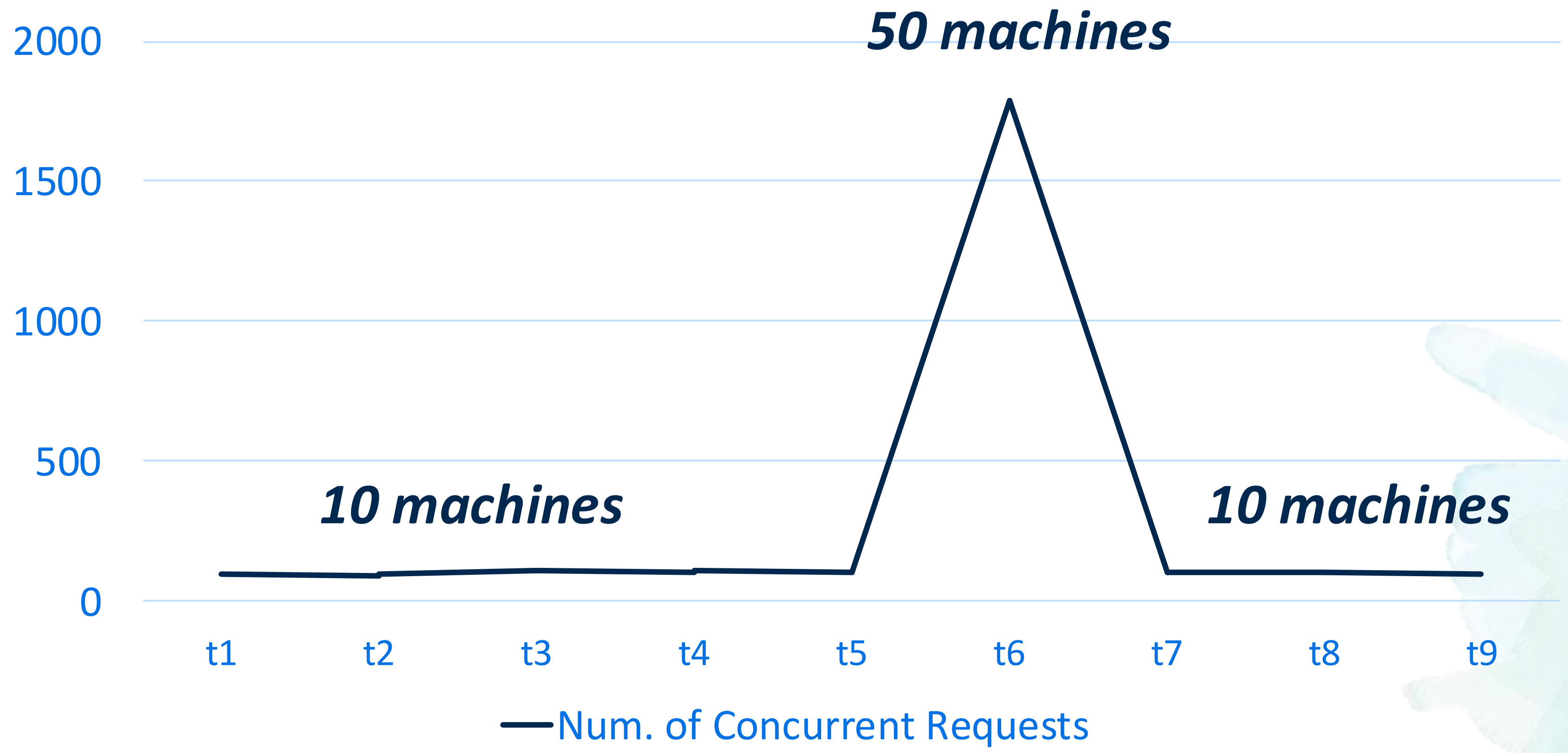
# Horizontal scaling is necessary for autoscaling

Auto-scale to more machines during traffic spike (more in Kubernetes module)



# Autoscaling

## Minimum resource wastage



***Horizontal scaling allows autoscaling, but did we solve all problems?***

# Heterogenous resource requirements across services

- Provisioning is driven by the most resource-hungry service
- Example RAM requirements
  - CustomerService: 32 GB
  - OrderService: 18 GB
  - ProductService: 16 GB
  - Minimum machine RAM? 32 GB

# Deployability concerns

- Updating one component requires redeploying the entire application
- Reverting a change requires redeploying the entire application
- Slow, error-prone process

# Need for low interdependence

- Software often consists of thousands of components
  - Each component has a dedicated team working on it
- Teams need to work independently
  - E.g., ProductService team should be able to update the Products Tbl schema without consulting other service teams
- Need low coupling between services
- Solution: microservices

# Microservices

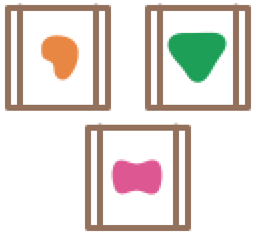
- An approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API
- Independently deployable by automated processes
- Bare minimum centralized management
- Smart endpoints connected by “dumb” pipes

https://martinfowler.com/articles/microservices.html

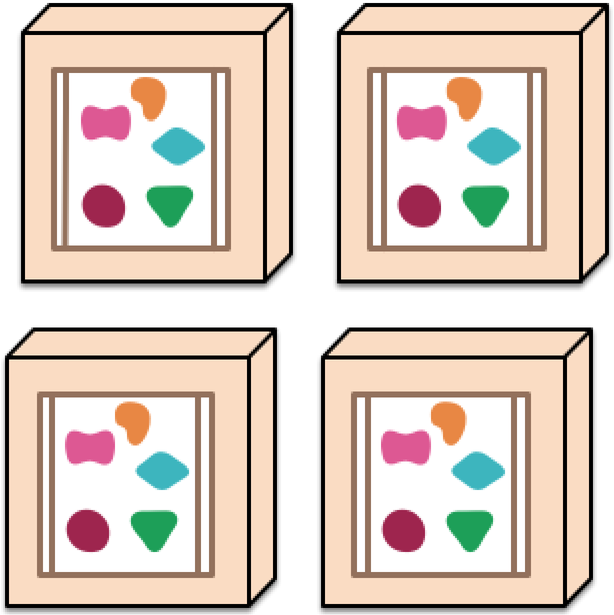
# Microservices overview

*A monolithic application puts all its functionality into a single process...*

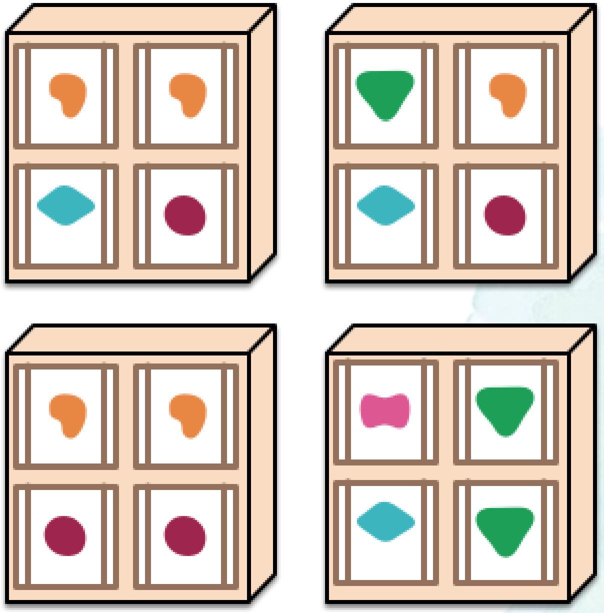
*A microservices architecture puts each element of functionality into a separate service...*



*... and scales by replicating the monolith on multiple servers*

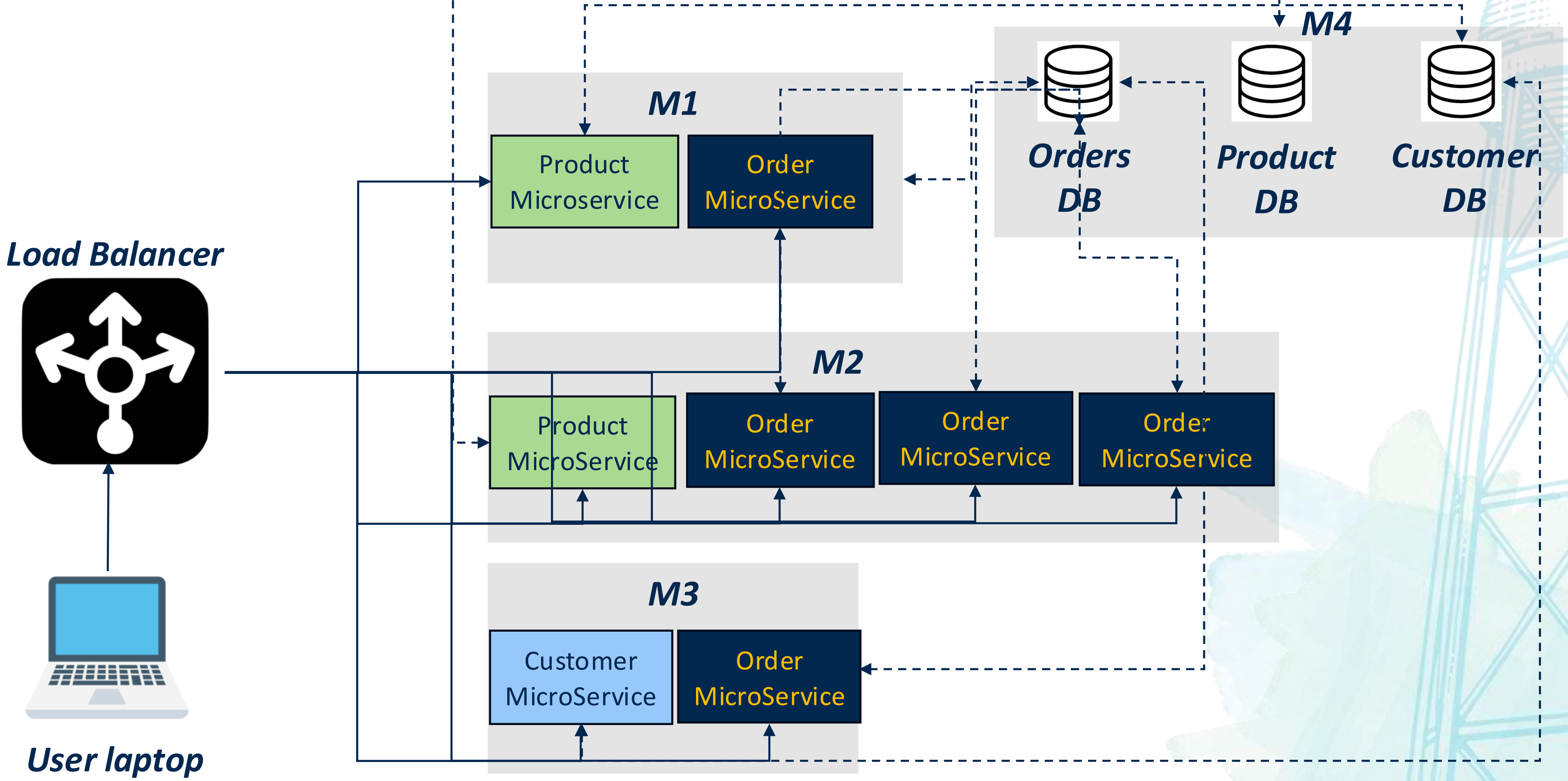


*... and scales by distributing these services across servers, replicating as needed.*

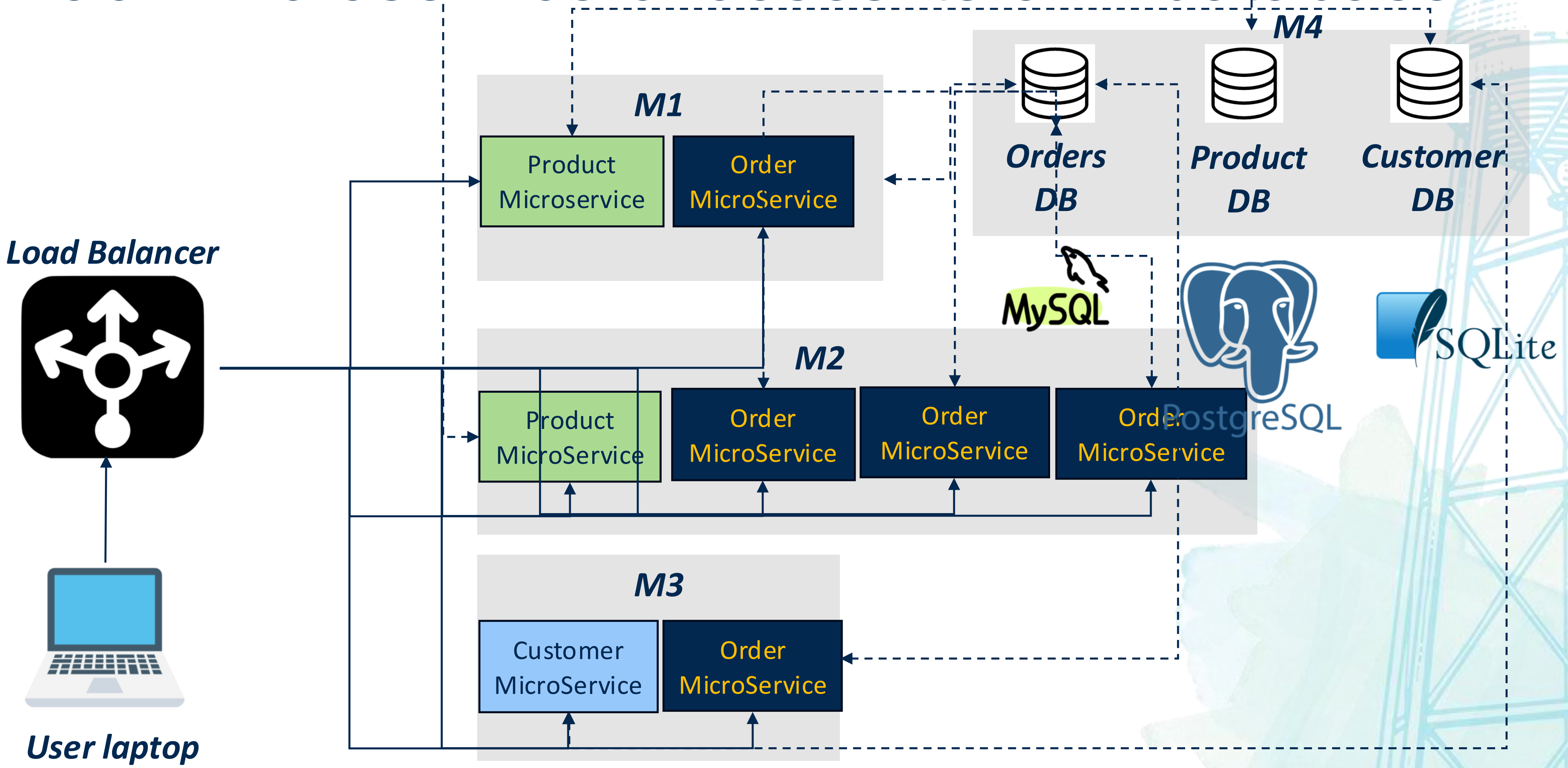


https://martinfowler.com/articles/microservices.html

# Microservice architecture for online store



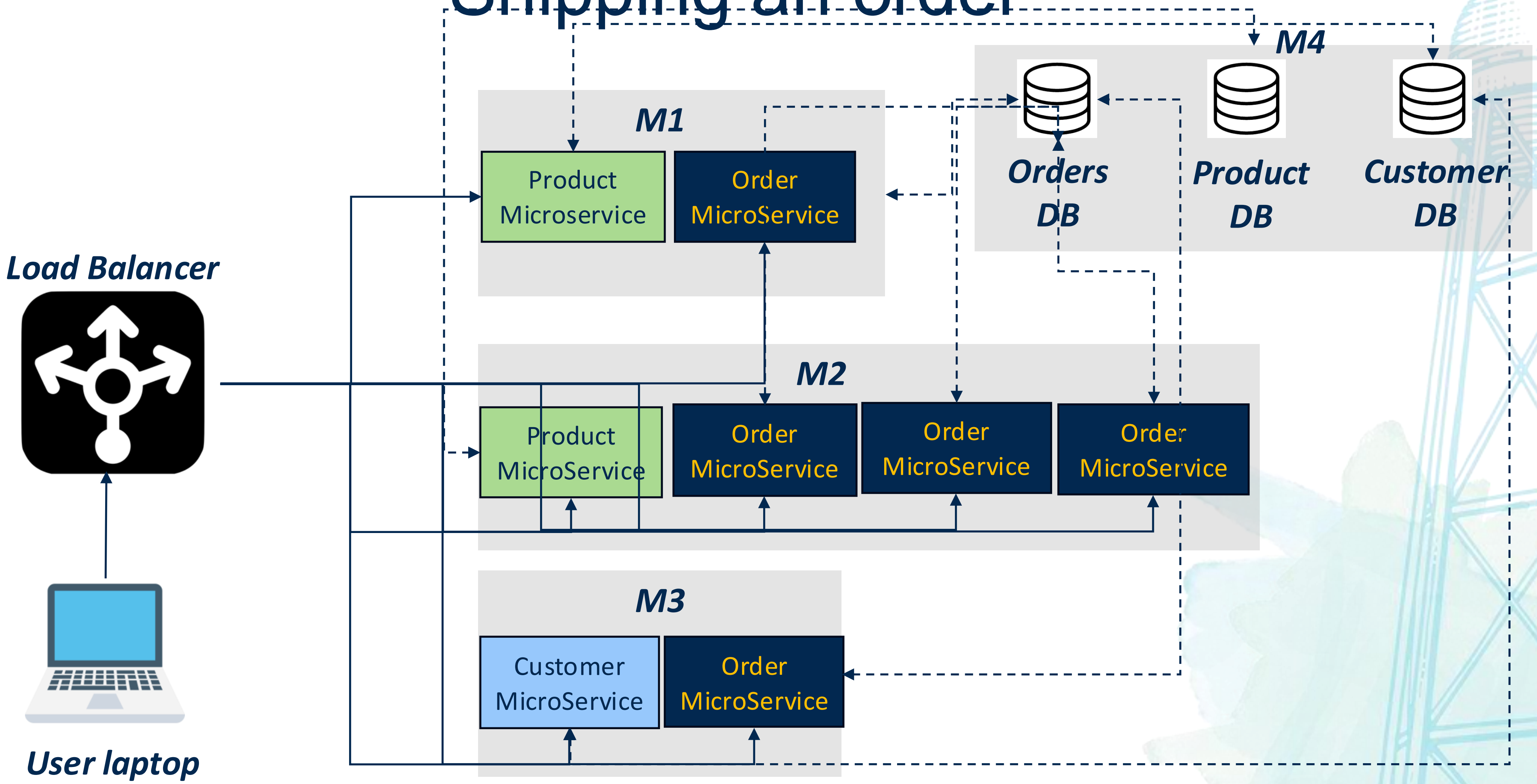
# Each microservice chooses its own database



# What did we achieve?

- Decompose monolithic app into microservices
- Improve decoupling
  - Each microservice can be scaled independently
  - Each microservice can be deployed independently
  - Each microservice can evolve independently – DB schema, choice of programming languages
- ***Did we solve all problems?***

# Shipping an order



# Reusing monolithic order schema

Product MicroService

Order MicroService

Customer MicroService

*Products tbl*

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

MySQL

*Customer tbl*

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | ABCD |

PostgreSQL

*Order tbl*

| OrderId | ProductId | CustomerId |
|---|---|---|
| 110001 | 1001 | 9001 |

SQLite

***Cannot perform joins!***

# Non-solution

- DO NOT want to query the Customer and Product microservices during shipping
- Will introduce tight coupling and interdependence
  - For example, what if the Customer microservice is down during shipment?

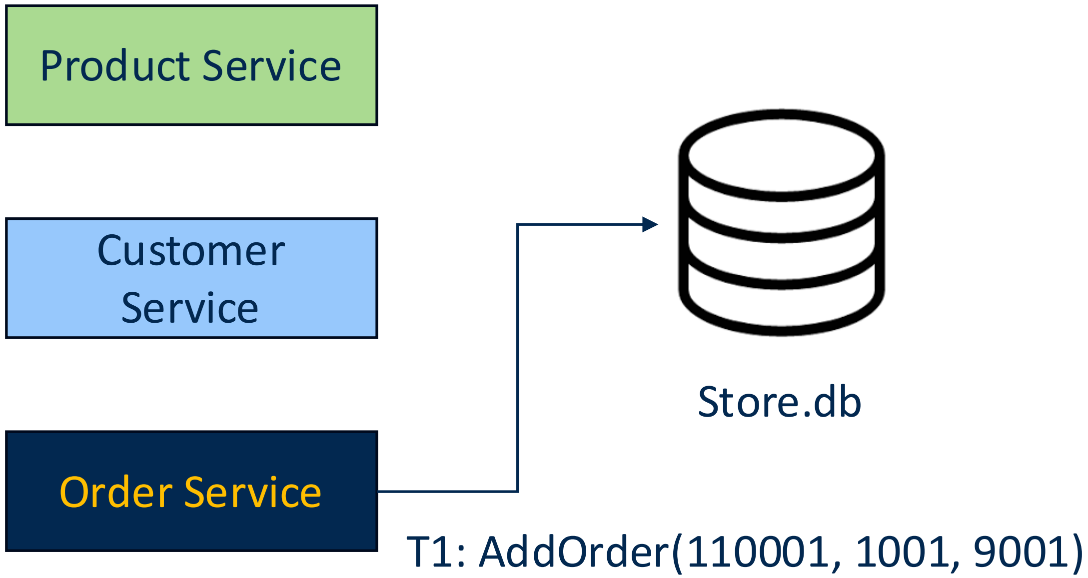
# Denormalization

- Must denormalize the data
- Order microservice duplicates and store the customer details in the Order db
- How? (after a few slides)

| OrderId | ProductName | CustomerName | CustomerAddr |
|---|---|---|---|
| 110001 | ABC | XYZ | CDEF |
| 110002 | ABC | XYZ | CDEF |
| 110003 | ABC | UVW | ABCD |

# Consistency guarantees

- Consistency property of a system governs how and when updates to shared data become visible to different components of the system
- Monolithic apps typically have strong consistency – updates reflect immediately to subsequent reads



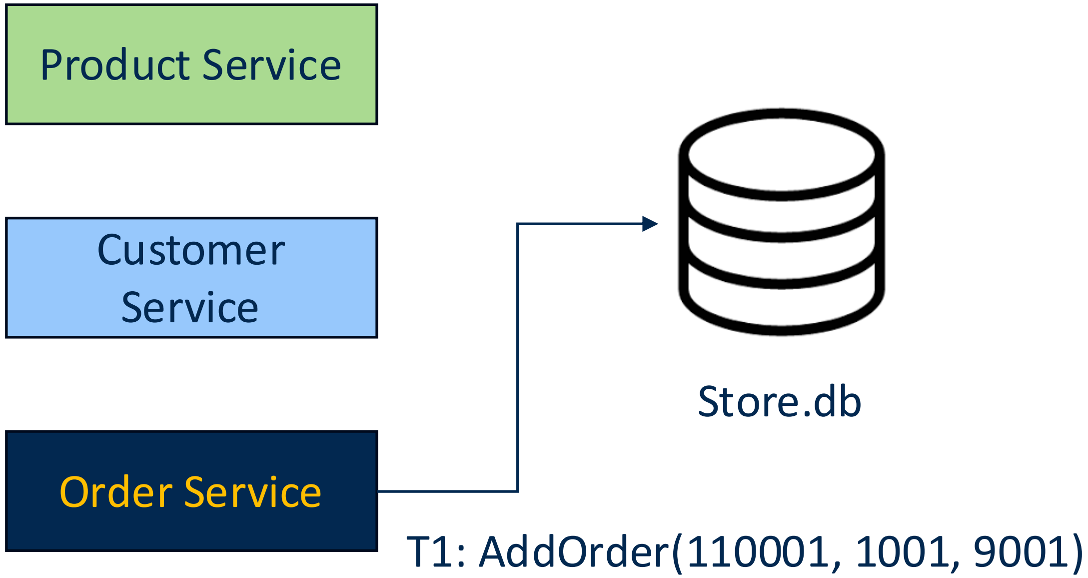

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | ABCD |

| OrderId | ProductId | CustomerId |
|---|---|---|
| | | |

# Consistency guarantees

- Consistency property of a system governs how and when updates to shared data become visible to different components of the system
- Monolithic apps typically have strong consistency – updates reflect immediately to subsequent reads



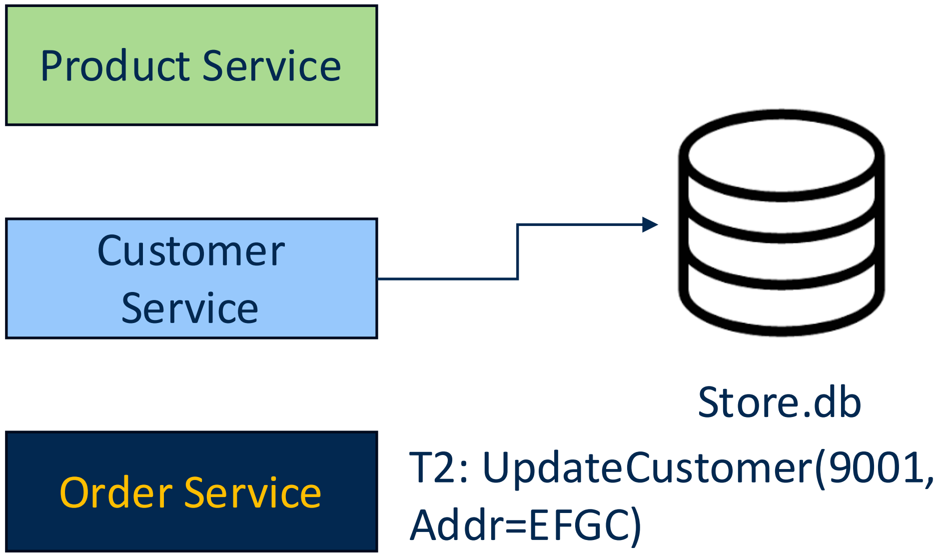

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | ABCD |

| OrderId | ProductId | CustomerId |
|---|---|---|
| 110001 | 1001 | 9001 |

# Consistency guarantees

- Consistency property of a system governs how and when updates to shared data become visible to different components of the system
- Monolithic apps typically have strong consistency – updates reflect immediately to subsequent reads



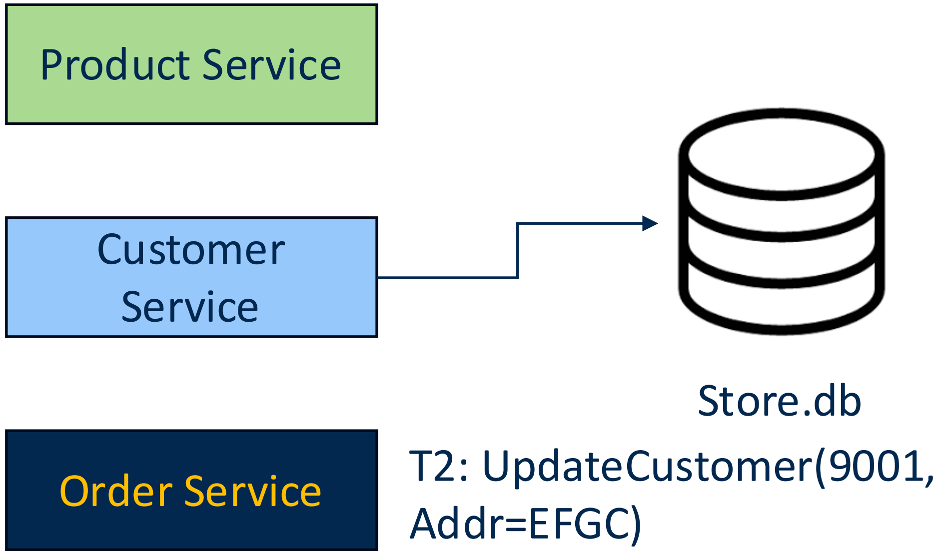

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | ABCD |

| OrderId | ProductId | CustomerId |
|---|---|---|
| 110001 | 1001 | 9001 |

# Consistency guarantees

- Consistency property of a system governs how and when updates to shared data become visible to different components of the system
- Monolithic apps typically have strong consistency – updates reflect immediately to subsequent reads



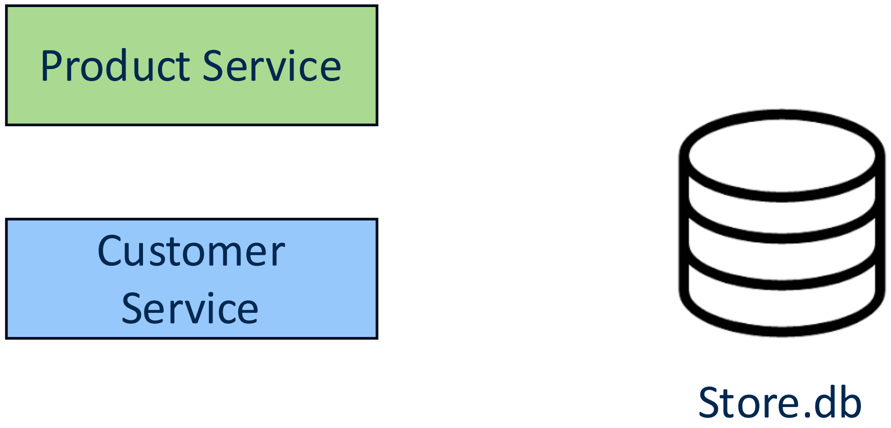

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | EFGC |

| OrderId | ProductId | CustomerId |
|---|---|---|
| 110001 | 1001 | 9001 |

# Consistency guarantees

- Consistency property of a system governs how and when updates to shared data become visible to different components of the system
- Monolithic apps typically have strong consistency – updates reflect immediately to subsequent reads




T3: ShipOrder -> JOIN(1001, 9001) -> (ABC, UVW, EFGC)

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | EFGC |

| OrderId | ProductId | CustomerId |
|---|---|---|
| 110001 | 1001 | 9001 |

# Microservice consistency guarantees

- Core idea: a microservice ***owns*** some data
  - ***Notifies*** dependent microservices of change
  - ***Soft guarantees*** on when the updates synchronize

# Microservice consistency guarantees

- Microservice-oriented apps are distributed
- Enforcing strong consistency guarantees in distributed systems is very hard
  - Network overhead, network partition, node failures
- Typically have **eventual consistency** guarantees – updates are **eventually** visible to all components
- System must work around these limitations

# Working with eventual consistency

Product MicroService ↔ Products.db

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

*Products tbl*

Customer MicroService ↔ Customer.db

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| | | |

*Customer tbl*

***CustomerMicroservice owns the customer data***

Order MicroService ↔ Orders.db

| OrderId | ProductName | CustomerAddr |
|---|---|---|
| | | |

*Order tbl*

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| | | |
| | | |

*Customer Cache tbl*

***OrderMicroservice maintains a cache of the customer information***

# Working with eventual consistency

Product MicroService

T1: AddCustomer(9002, UVW, ABCD)

Customer MicroService

Order MicroService

Products.db

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

***Products tbl***

Customer.db

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | ABCD |

***Customer tbl***

Orders.db

| OrderId | ProductName | CustomerAddr |
|---|---|---|
| | | |

***Order tbl***

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| | | |
| | | |

***Customer Cache tbl***

# Working with eventual consistency

Product MicroService

Products.db

| ProductId | ProductName | ProductCount |
| --- | --- | --- |
| 1001 | ABC | 10 |

*Products tbl*

Customer MicroService

Customer.db

| CustomerId | CustomerName | CustomerAddr |
| --- | --- | --- |
| 9001 | XYZ | CDEF |
| 9002 | UVW | ABCD |

*Customer tbl*

T2: UpdateCust (9002, UVW, ABCD)

Order MicroService

Orders.db

| OrderId | ProductName | CustomerAddr |
| --- | --- | --- |
| | | |

*Order tbl*

| CustomerId | CustomerName | CustomerAddr |
| --- | --- | --- |
| 9002 | UVW | ABCD |
| | | |

*Customer Cache tbl*

# Working with eventual consistency

Product MicroService

Products.db

| ProductId | ProductName | ProductCount |
| --- | --- | --- |
| 1001 | ABC | 10 |

***Products tbl***

Customer MicroService

Customer.db

| CustomerId | CustomerName | CustomerAddr |
| --- | --- | --- |
| 9001 | XYZ | CDEF |
| 9002 | UVW | ABCD |

***Customer tbl***

Order MicroService

Orders.db

| OrderId | ProductName | CustomerAddr |
| --- | --- | --- |
| | | |

***Order tbl***

| CustomerId | CustomerName | CustomerAddr |
| --- | --- | --- |
| 9002 | UVW | ABCD |
| | | |

***Customer Cache tbl***

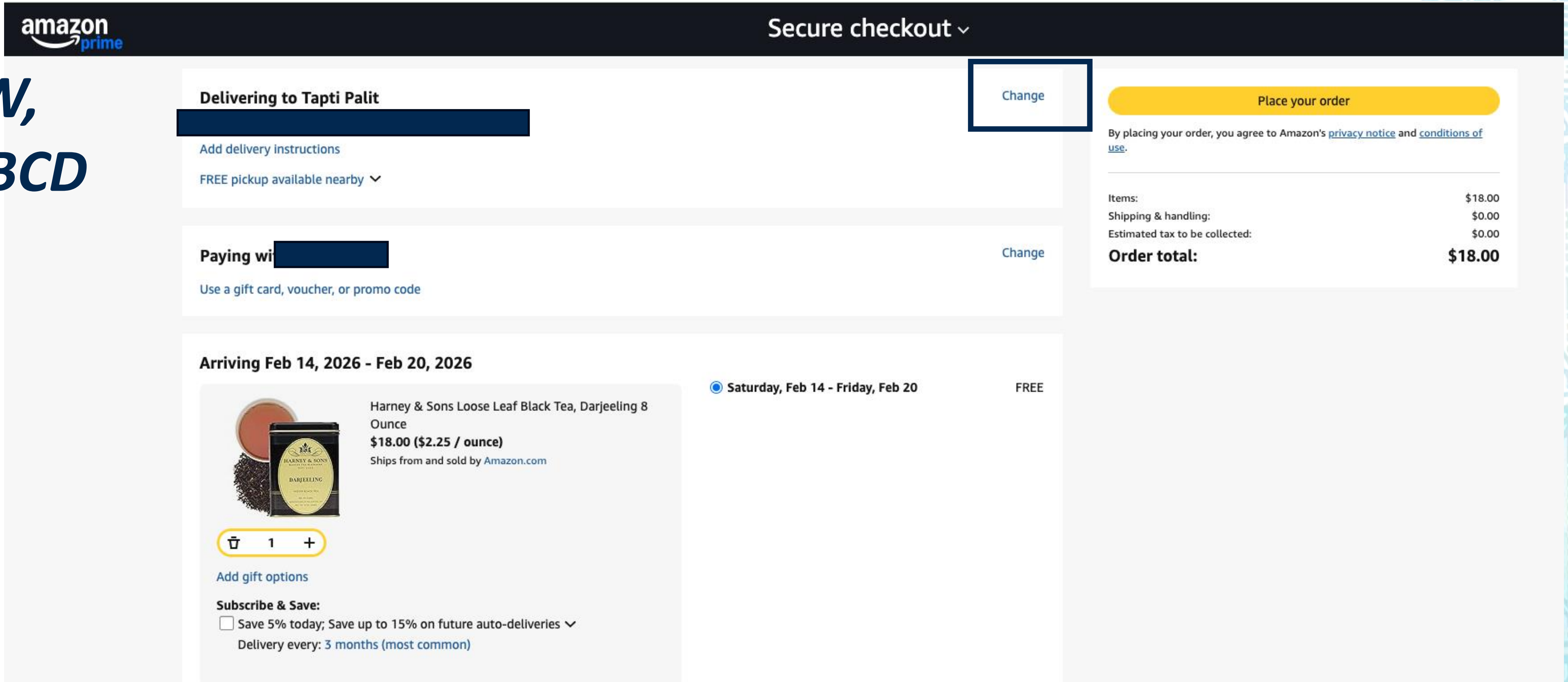
User 9002 tries placing an order

T3: FetchOrderAddrForConfirm() -> (9002, UWV, ABCD)

# Working with eventual consistency

***Name: UVW,***

***Address: ABCD***



# Working with eventual consistency

Product MicroService

Products.db

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

***Products tbl***

Customer MicroService

Customer.db

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | ABCD |

***Customer tbl***

Order MicroService

T4: AddOrder(110001, ABC, ABCD)

Orders.db

| OrderId | ProductName | CustomerAddr |
|---|---|---|
| 110001 | ABC | ABCD |

***Order tbl***

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9002 | UVW | ABCD |
| | | |

***Customer Cache tbl***

# Working with eventual consistency

Product MicroService

T5: UpdateCust(9002, UVW, EFGC)

Customer MicroService

Order MicroService

Products.db

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

*Products tbl*

Customer.db

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | EFGC |

*Customer tbl*

Orders.db

| OrderId | ProductName | CustomerAddr |
|---|---|---|
| 110001 | ABC | ABCD |

*Order tbl*

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9002 | UVW | ABCD |
| | | |

*Customer Cache tbl*

# Working with eventual consistency

Product MicroService

Products.db

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

***Products tbl***

Customer MicroService

Customer.db

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | EFGC |

***Customer tbl***

Order MicroService

Orders.db

| OrderId | ProductName | CustomerAddr |
|---|---|---|
| 110001 | ABC | ABCD |

***Order tbl***

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9002 | UVW | ABCD |
| | | |

***Customer Cache tbl***

T6: ShipOrder(11001, ABC, ABCD)

# Working with eventual consistency

Product MicroService

Products.db

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

***Products tbl***

Customer MicroService

T7: UpdateCustCache(9002, UVW, EFGC)

Customer.db

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | EFGC |

***Customer tbl***

Order MicroService

Orders.db

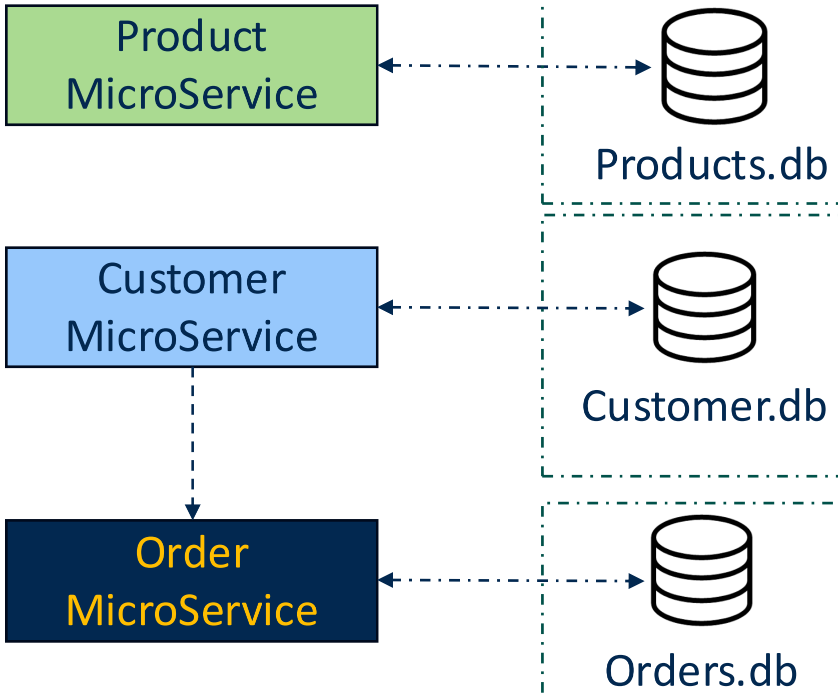
| OrderId | ProductName | CustomerAddr |
|---|---|---|
| 110001 | ABC | ABCD |

***Order tbl***

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9002 | UVW | EFGC |
| | | |

***Customer Cache tbl***

# Working with eventual consistency




| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

*Products tbl*

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | EFGC |

*Customer tbl*

| OrderId | ProductName | CustomerAddr |
|---|---|---|
| 110001 | ABC | ABCD |

*Order tbl*

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9002 | UVW | EFGC |
| | | |

*Customer Cache tbl*

***Subsequent orders load updated customer address***

UCDAVIS

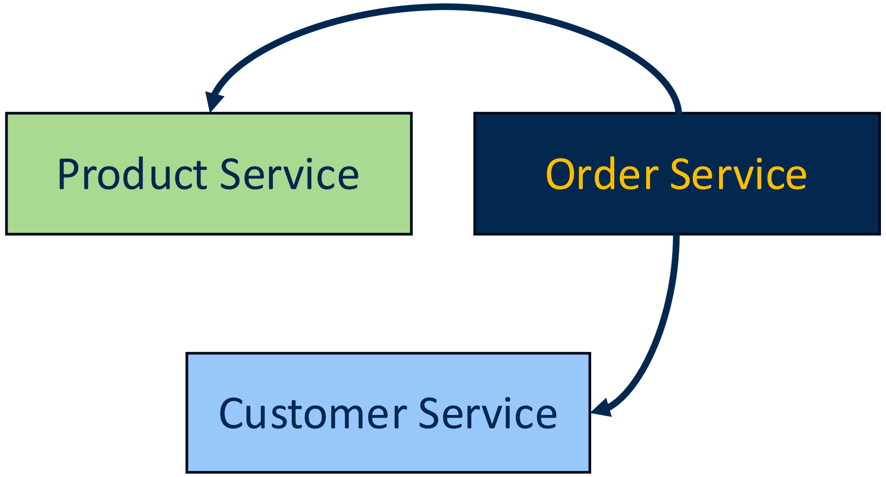
# Microservice design concerns

- Solution for eventual consistency is generally use-case dependent
- Reduce tight coupling
  - A microservice should not know the internal DB schema of another microservice
  - A microservice should not depend on another to be running to perform its task
- Limit network communication

# Network communication cost

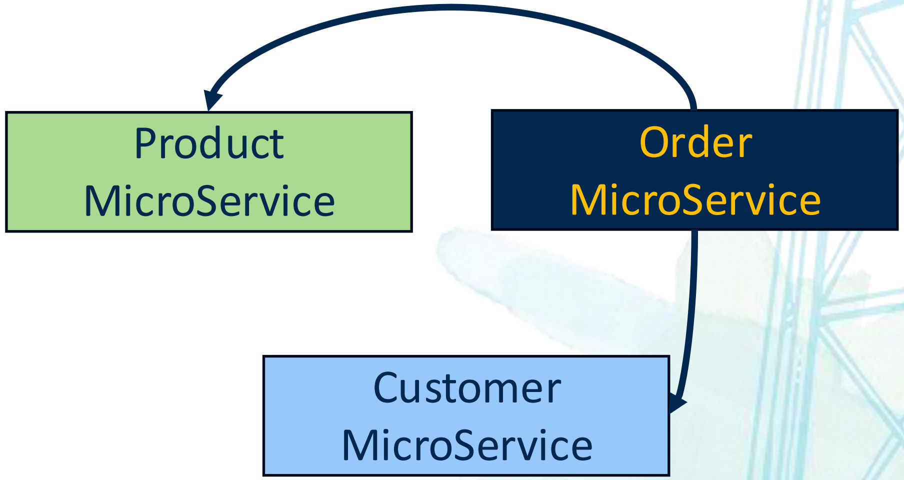
## Monolithic applications

***Function calls (few nanoseconds)***




## Microservices

***Network RPC calls (few microseconds)***

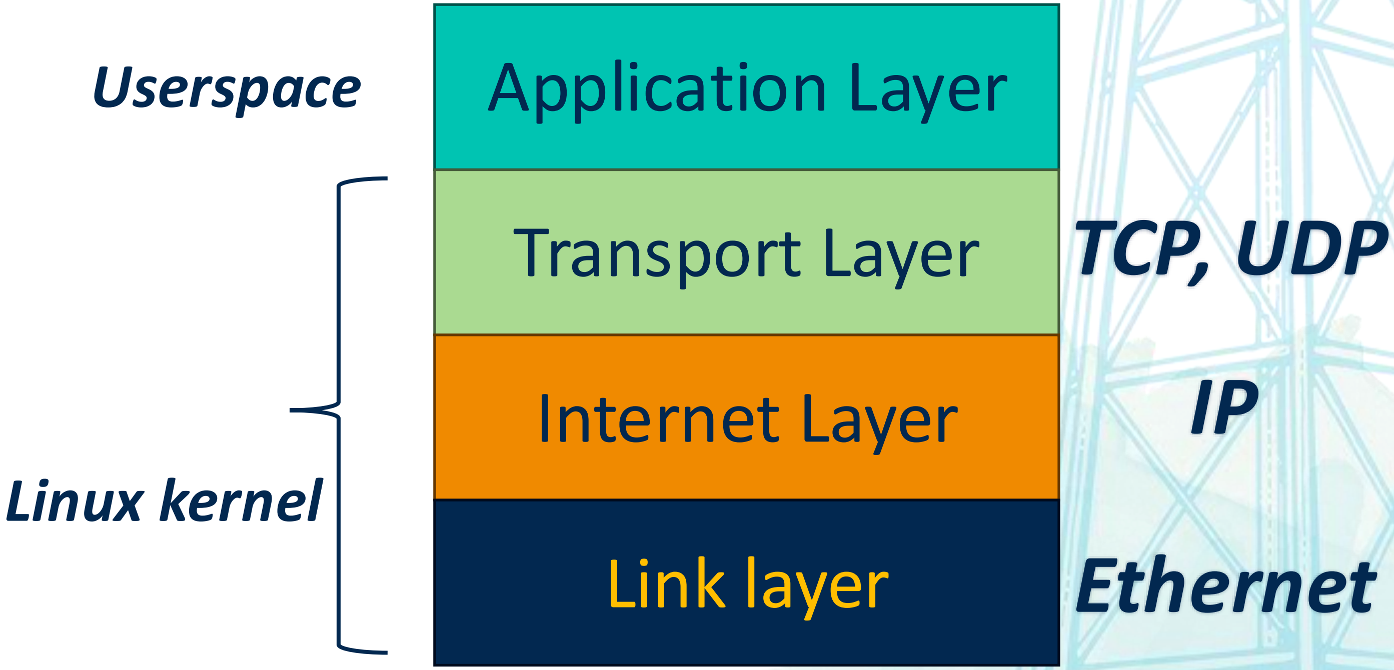


# Network overhead decomposition

- Network latency
  - Datacenters use fast network connections
    - Latest technology - Infiniband has ~1-2 microsecond latency, 400+ Gbps bandwidth
    - Network communication is still over ethernet in most data centers
  - Still not as fast as a local function call
- OS kernel overhead

# Kernel overhead for network comm. (Not in syllabus)

- The OS kernel contains the networking code
- TCP-IP is the most common networking stack
- It is organized in layers
- Every layer has a ***protocol***

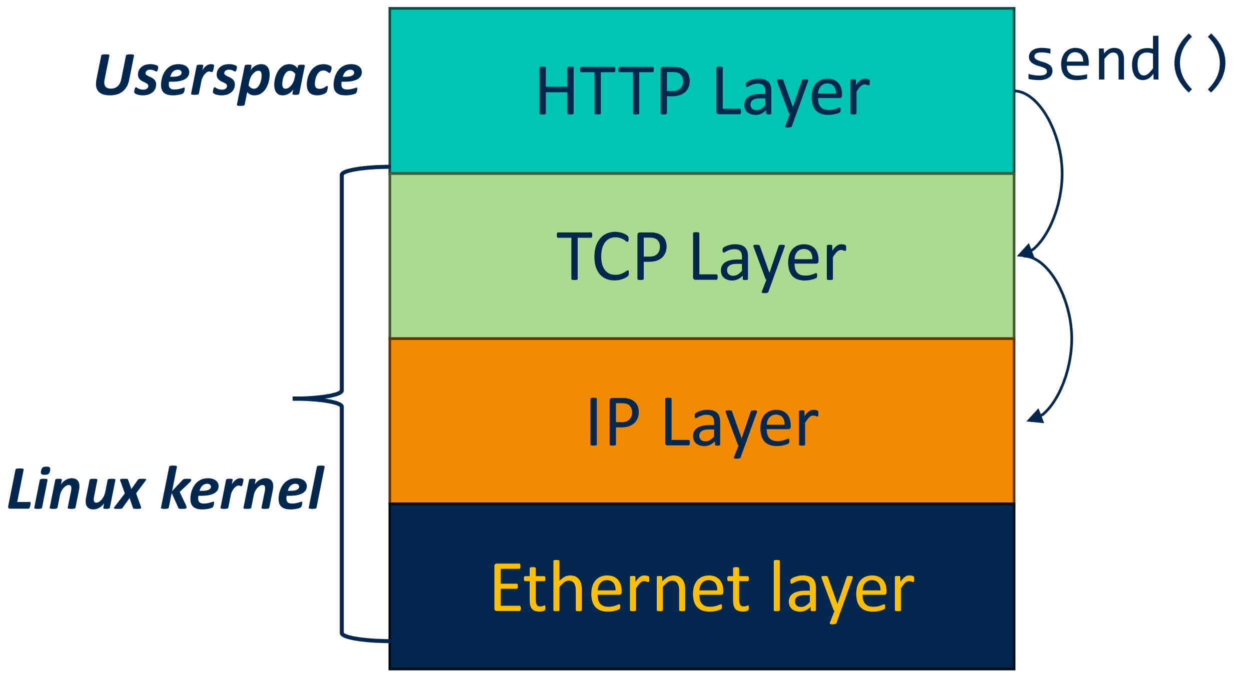


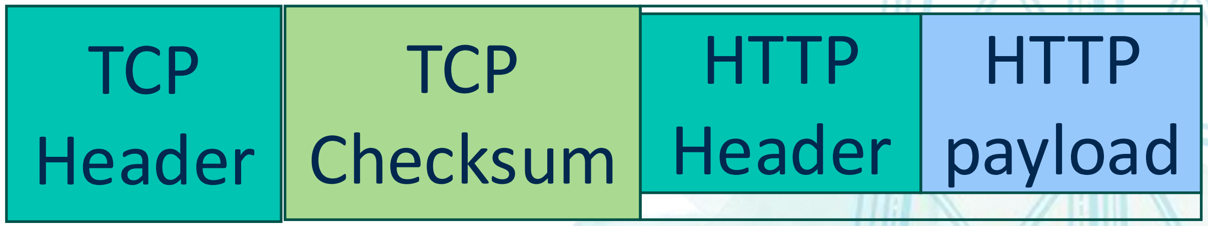
# Each layer has a protocol (Not in syllabus)

- Every layer/protocol has a fixed message format
  - Header
  - Payload
  - [optional] Checksum

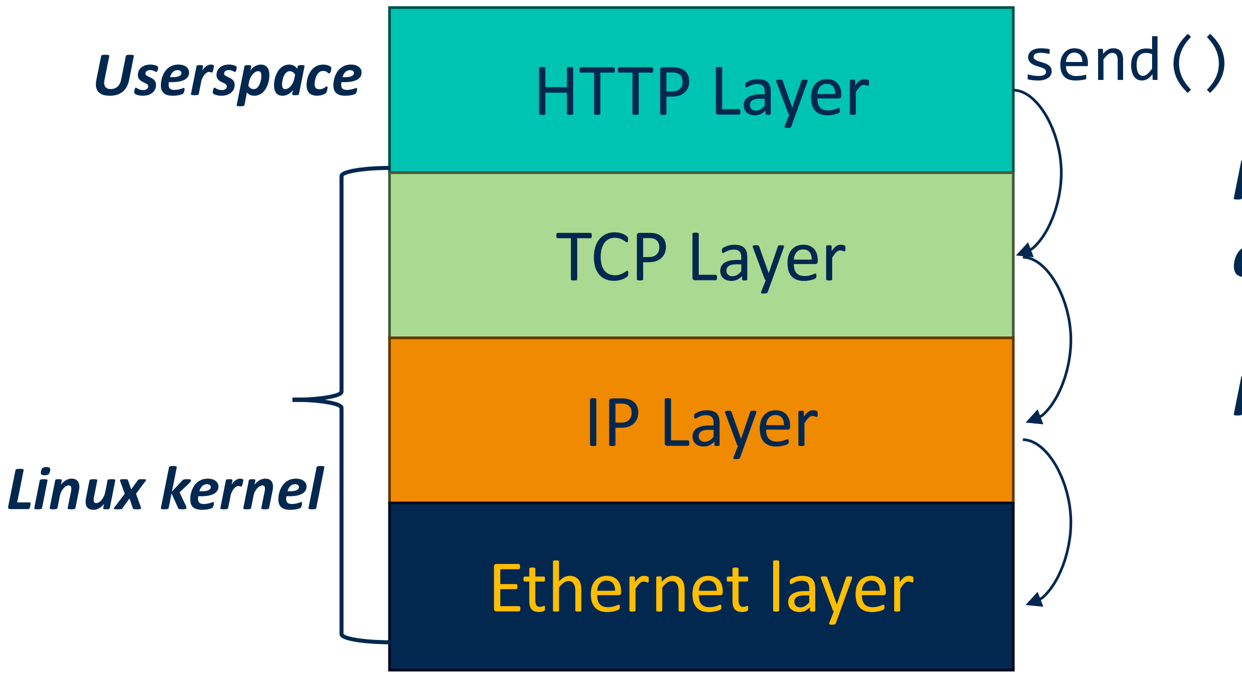

- As the packet traverses through the layers, packets are rewrapped

# Life of a packet (Not in syllabus)



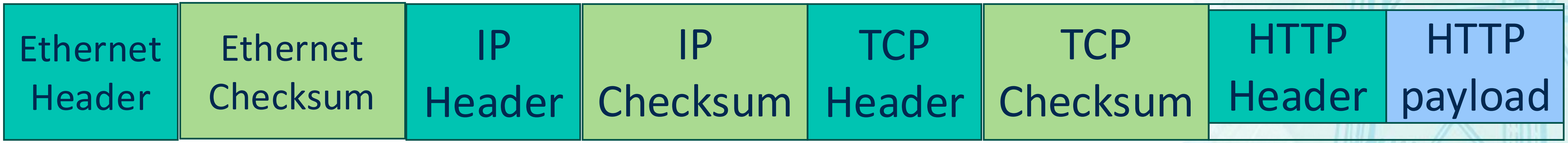



# Life of a packet (Not in syllabus)




***Packet encapsulation process + kernel context switch can take 100+ microseconds***

***Reading 6 will discuss alternative solutions***



# Microservice pros

- Stronger decoupling and lower interdependence
- Improved scalability
- Easer deployment

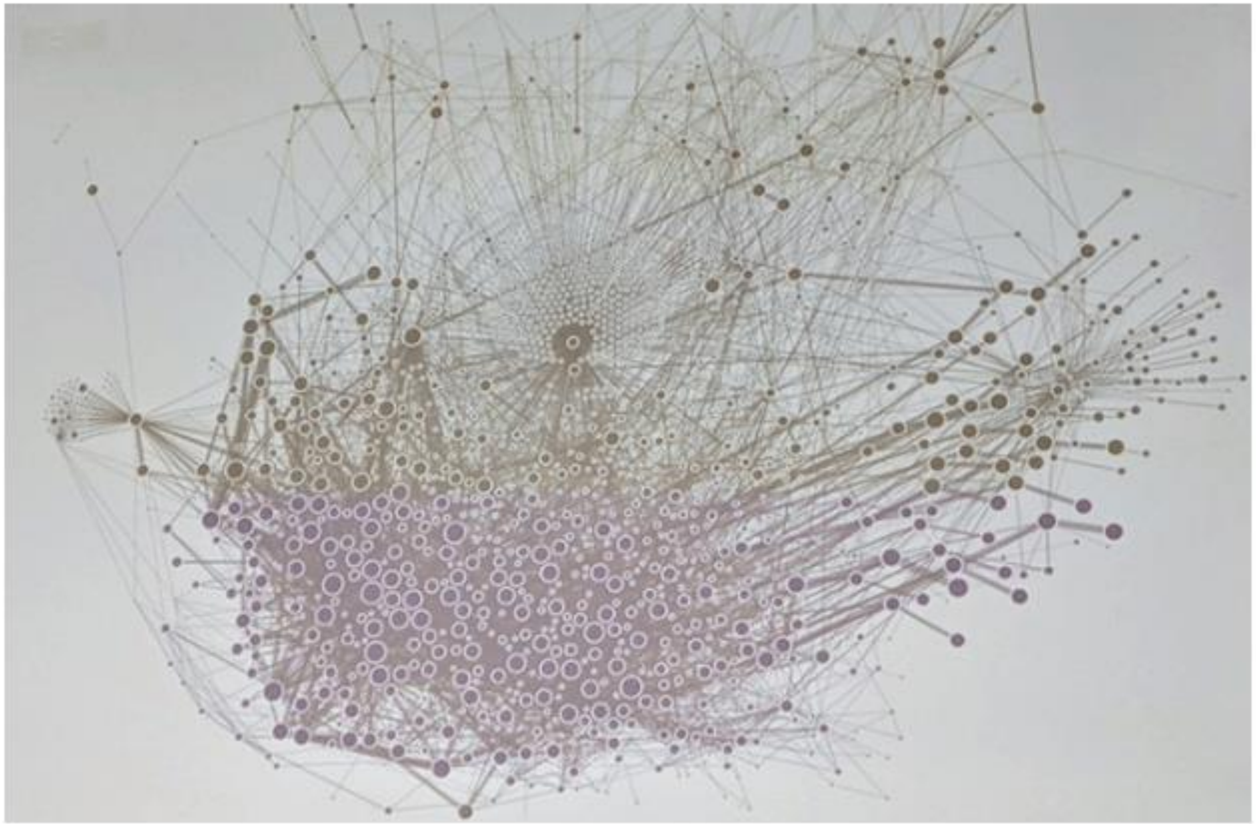
# Microservice cons

- Causes data denormalization
- Network overhead
- Higher complexity
- Debugging complex interactions is harder

# Latency vs throughput

- Latency – time taken for one operation
  - Measured in seconds, milliseconds, microseconds, etc
  - Service Level Objectives/Agreements (SLO/SLA)
    - Example: 95% of all requests should be served in under 2 ms
- Throughput – number of operations performed in unit time (requests/sec)
- Microservices increase latency compared to monolithic operation, but improve throughput

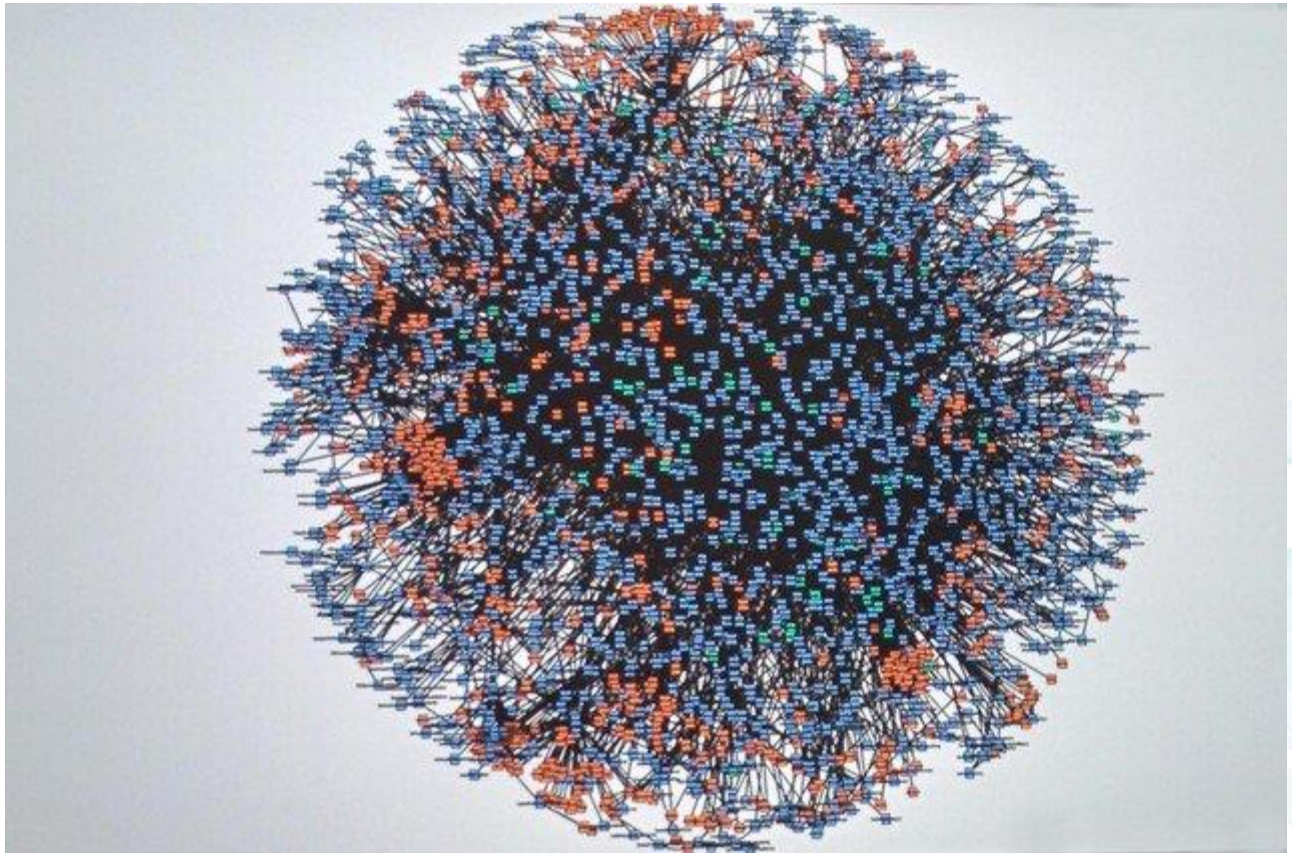
# Microservices at Uber (2019)



https://x.com/msuriar/status/1110244877424578560

# Microservices at Amazon (2008)

- Code-named “Deathstar”



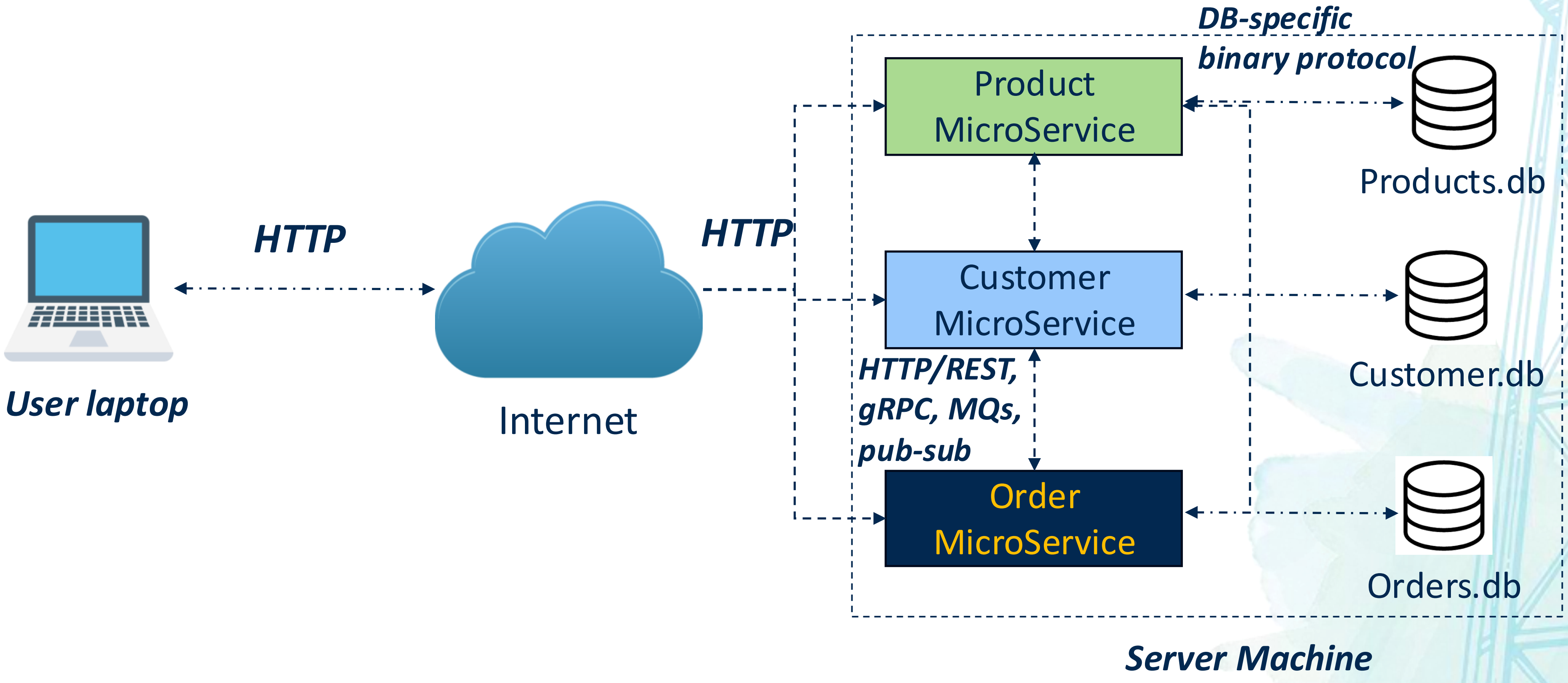
https://x.com/Werner/status/741673514567143424

# Data management and communication

- Once data and computation are split across services, two problems remain
  - How is data stored?
  - How services communicate?
- First, Spring Boot overview

# Spring Boot with REST API overview

# Microservices over HTTP



# HTTP GET and POST request

- GET request - Used to retrieve data from the server
  - `GET /index.html HTTP/1.1`

    ***URL***
  - `GET /index.html?id=ECS160&count=10`

    ***Request parameters***

# HTTP GET and POST request

- POST request - used to send data to the server

  **URL**

  ```
  POST /users HTTP/1.1
  Content-Type: application/json{
     "name": "John Doe",
     "email": john.doe@example.com
  }
  ```

  ***Post "body"***

UCDAVIS

# Spring Boot Overview

- Framework for creating RESTful microservices
- Reduces boilerplate configuration code
- Embedded server (Tomcat/Jetty)
- Simplifies microservice creation through annotations
- Built-in support for REST APIs

# RESTful microservices with Spring Boot

- Create classes that can act as REST endpoints
- Uses annotations to denote REST endpoint URLs
  - Allows complete decoupling from the boilerplate code
- Types of requests
  - `@GetMapping`, `@PostMapping`, `@PutMapping`, and so on… for all HTTP methods
- `@PathVariable` – extract variable from GET request
- @RequestBody – extract the post request body

```
class MyRequest {
    private String postDate;
    private String postContent;
    // .. Getters and setters
}

@RestController
@RequestMapping("/myservice")
public class MyController {
    @PostMapping("/sayhello")
    public String sayHello(@RequestBody MyRequest request) {
        // do something
        return "";
    }
}
```

**Effective URL: http://[serverip:port]/myservice/sayhello**

UCDAVIS

# Spring Boot Framework

- Uses reflection to first look up all classes with @RestController annotation
- Then automatically creates Servlets out of the methods annotated with @GetMapping, @PostMapping, etc.
- Uses reflection to parse the request parameters into class objects annotated with @RequestBody
- Generates the WAR file and launches the Apache Tomcat server
  - Simply execute `mvn spring-boot:run`

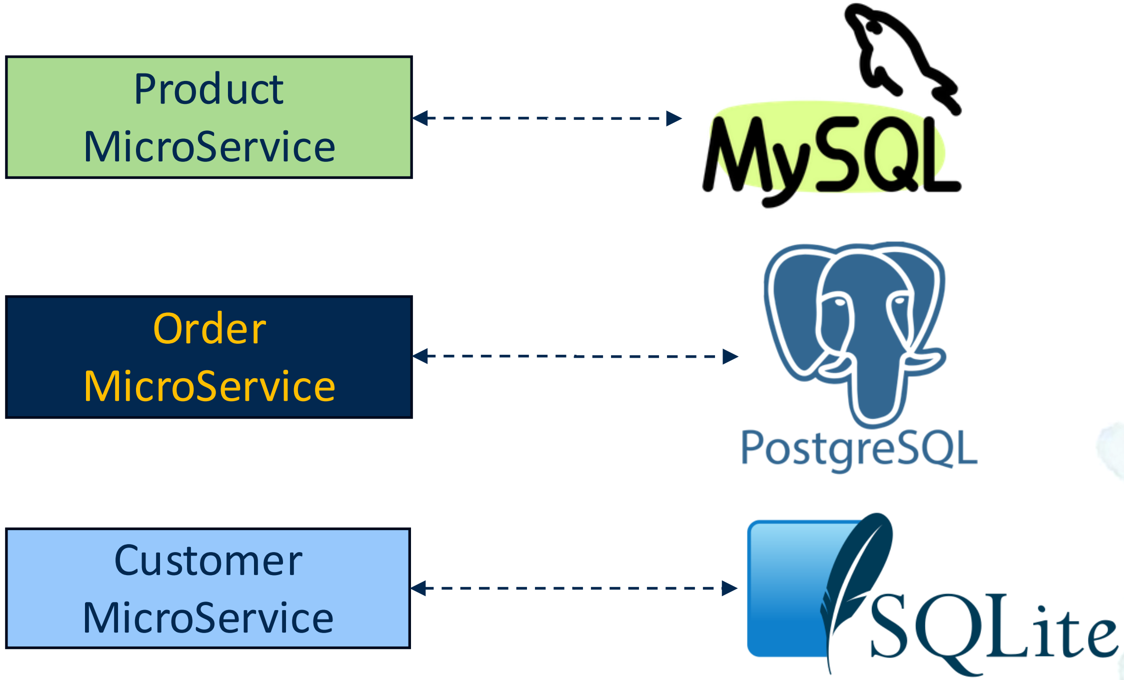
```
class MyRequest {
    private String postDate;
    private String postContent;
    // .. Getters and setters
}

@RestController
@RequestMapping("/myservice")
public class MyController {
    @PostMapping("/sayhello")
    public String sayHello(@RequestBody MyRequest
request) {
        return "";
    }
}
```

# Data management and communication

- Once data and computation are split across services, two problems remain
  - How is data stored?
  - How services communicate?

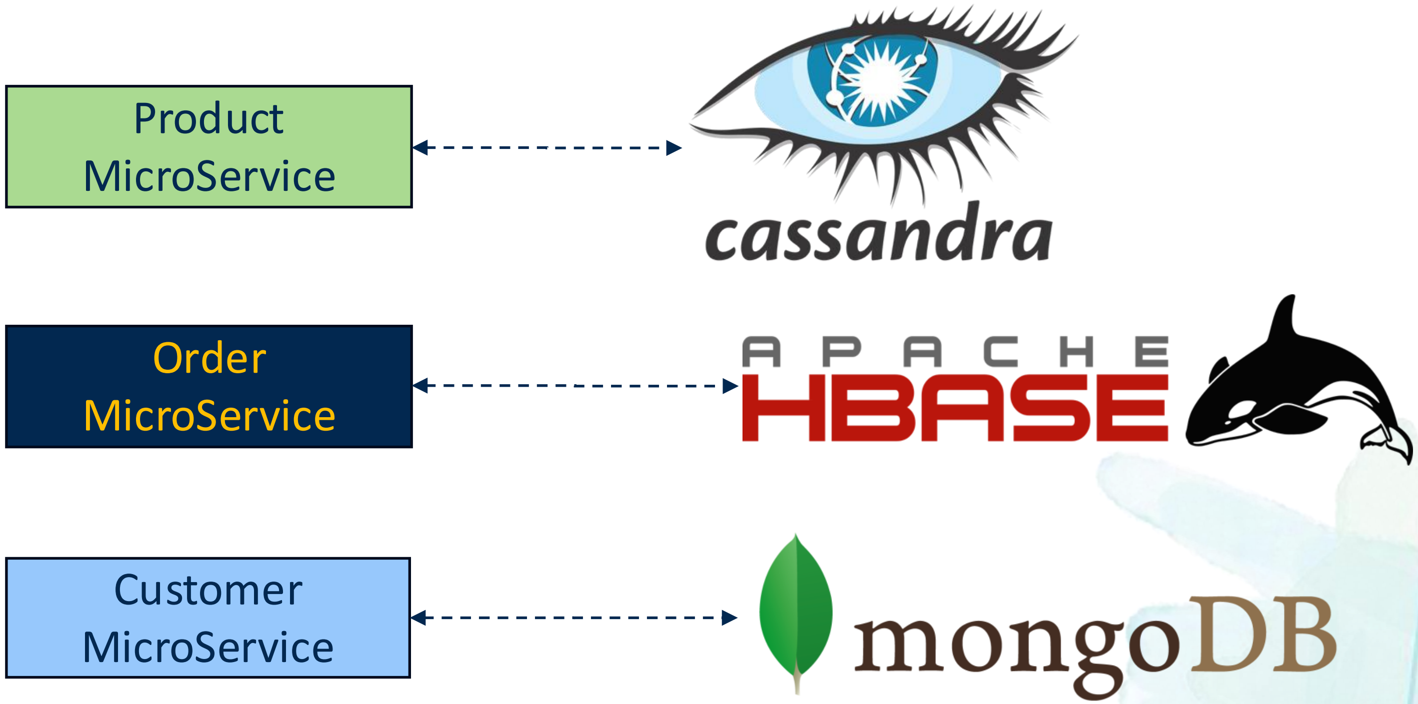
# Databases

# Microservices can have individual DBs




***MySQL, PostgreSQL, SQLite are relational databases***

# Microservices can have individual DBs




***Non-relational, NoSql databases***

# How to select which database to use?

MySQL

cassandra

PostgreSQL

APACHE HBASE

SQLite

mongoDB

redis

# Database choice dimensions

- Data model
  - Format of data user gives to the database
  - Examples - relational, document, graph, key-value
  - Shapes query semantics (joins, traversals, etc.)
    - Mechanism through which you can retrieve the data
- Storage engine
  - How data is physically stored and indexed
  - B-Trees, SSTables and LSM-Trees, Hash Indexes
  - Impacts performance tradeoffs (read vs. write performance, range scans, etc)

# Data models and query languages

- Relational databases (discussed earlier)
- Document model
- Key-value stores (discussed earlier in Redis discussion)

# Relational databases

- Stores data as tables
- Supports relations using foreign keys and joins
- Fixed schema

*Products tbl*

| ProductId | ProductName | ProductCount |
|---|---|---|
| 1001 | ABC | 10 |

*Customer tbl*

| CustomerId | CustomerName | CustomerAddr |
|---|---|---|
| 9001 | XYZ | CDEF |
| 9002 | UVW | ABCD |

*Order tbl*

| OrderId | CustomerId | ProductId |
|---|---|---|
| 110001 | 9001 | 1001 |

# Document model

- Data is a document!
  - JSON, XML
  - E.g. MongoDB
- The database maintains this document
  - Provides a query language to inspect the contents
- Flexible schema

# MongoDB BSON format

- BSON is binary-encoded JSON
- String that represents an object
- Objects consists of key-value pairs
- Values can be primitives, arrays, or nested JSON objects
- Naturally supports hierarchical and semi-structured data

***JSON object***

***Nested JSON object***

***JSON array***

```
[
  {
        "id": "usr123",
        "username": "coder_gal",
        "details": {
          "age": 28,
          "city": "San Francisco”
        },
        "skills": [
                "JavaScript",
                "Python",
                "JSON",
                "APIs"]
  }
]
```

UCDAVIS

# MongoDB

- MongoDB insert, find, update, delete operations

```
> db.createCollection("posts")

> db.posts.insertOne({
      title: "First post!",
      body: "Hello world!",
      likes: 3,
      category: "News",
      tags: ["news", "events"],
       author: "ABC"})

> db.posts.find( {category:
"news"} )
```

# MongoDB BSON format

- Fields are accessed using dot notation
  - `post.id`
  - `post.category`
- Arrays are accessed using index notation
  - `post.tags[0]`

```
> db.posts.findOne( {category: “news”} )

> db.posts.findOne( {category: “news”}).likes # prints 3

> db.posts.findOne( {category: “news”}).tags[0] # print news
```

# Key-value stores

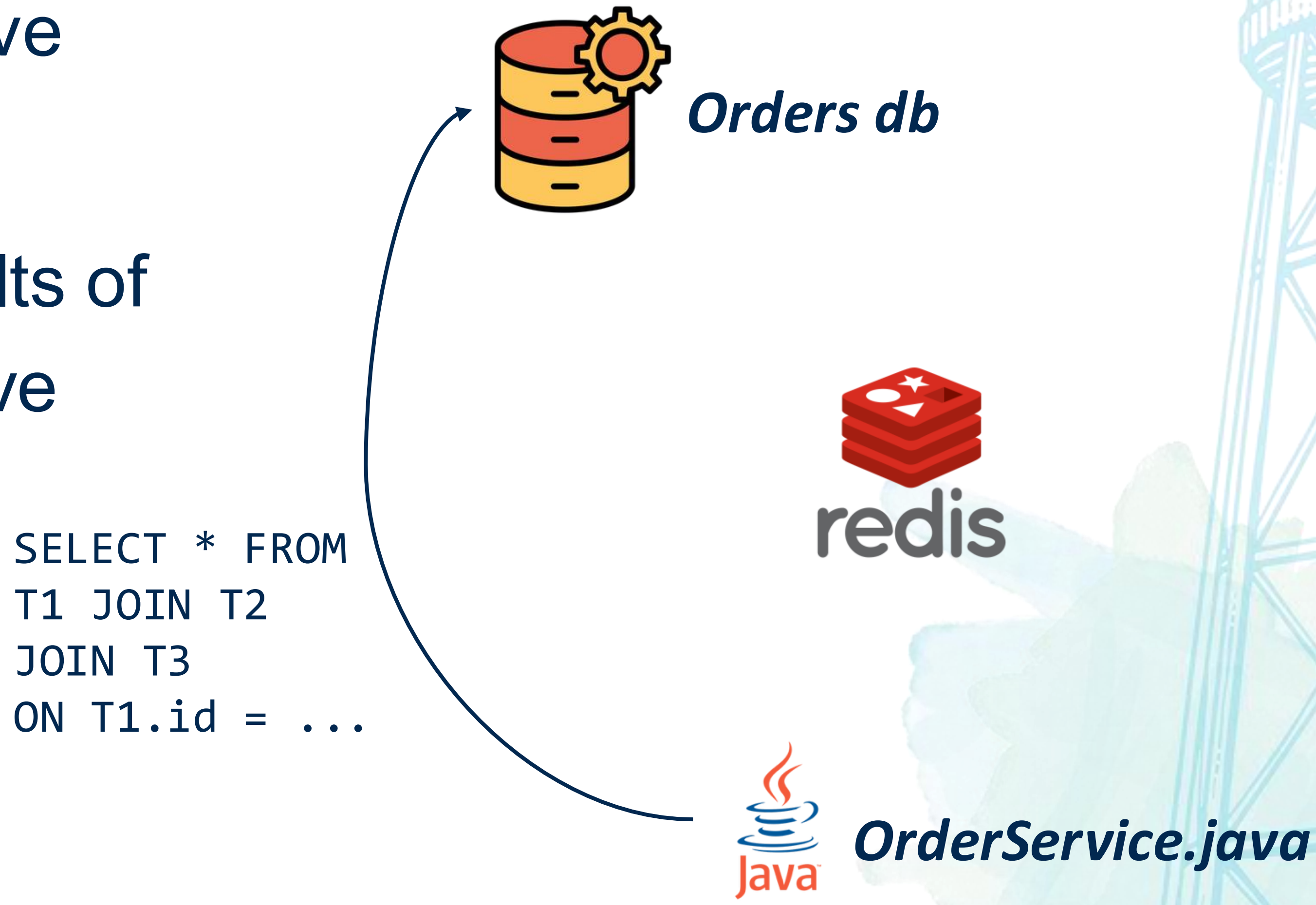
- Simple key-value pairs
- Redis, Memcached
- Typically operate in-memory only
- Usages
  - Cache expensive queries
  - Communication (PUBLISH and SUBSCRIBE primitives) (later)

| Key | Value | |
|---|---|---|
| 10279811 | **Field** | **Value** |
| | Name | ABC |
| | Age | 22 |
| | GPA | 3.8 |
| | Credits | 45 |
| 10279812 | **Field** | **Value** |
| | Name | DEF |
| | Age | 21 |
| | GPA | 3.9 |
| | Credits | 60 |

***Students Redis DB***

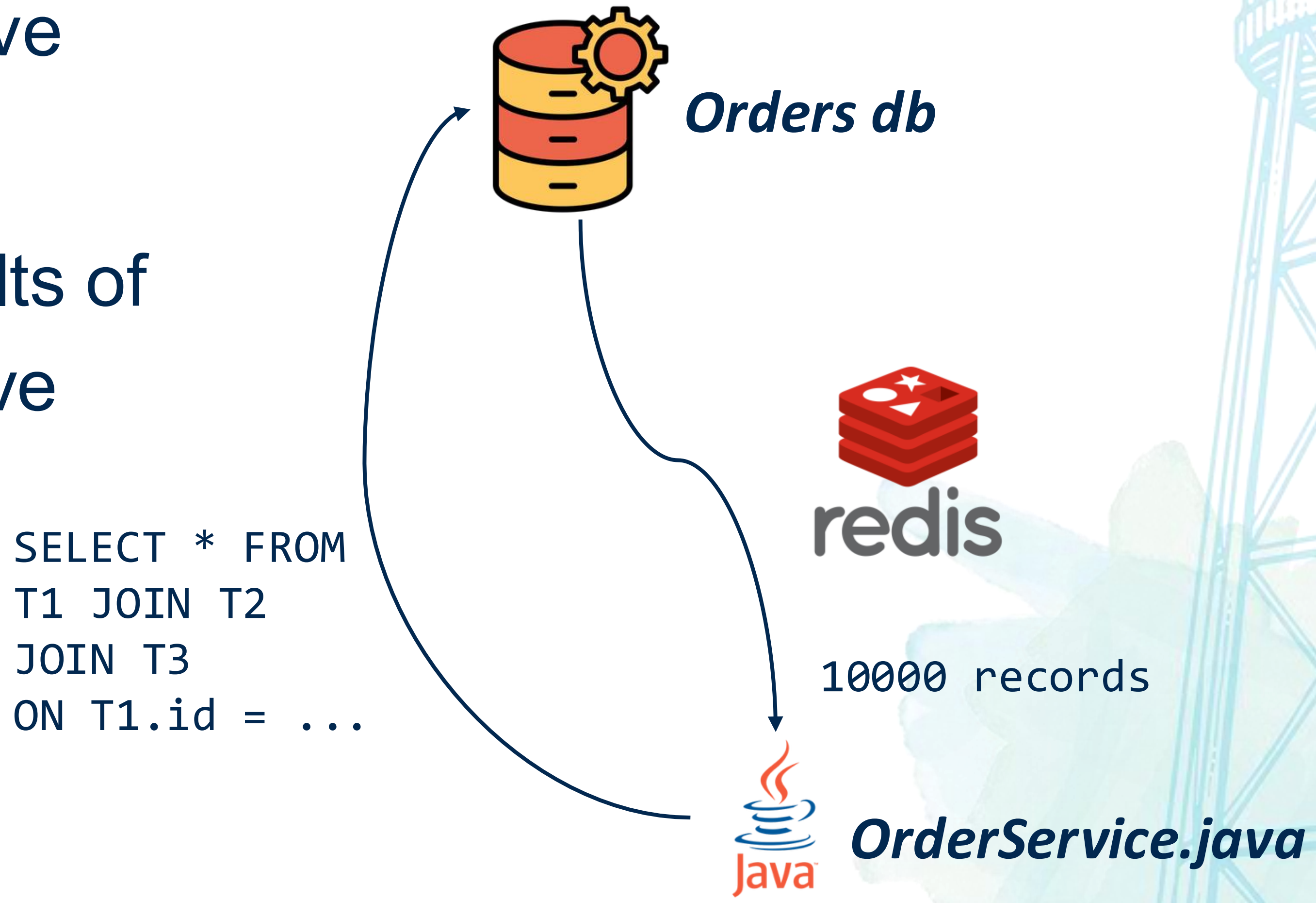
# Key-value stores

- Cache results of expensive operations
- For e.g. caching the results of API requests, or expensive database queries



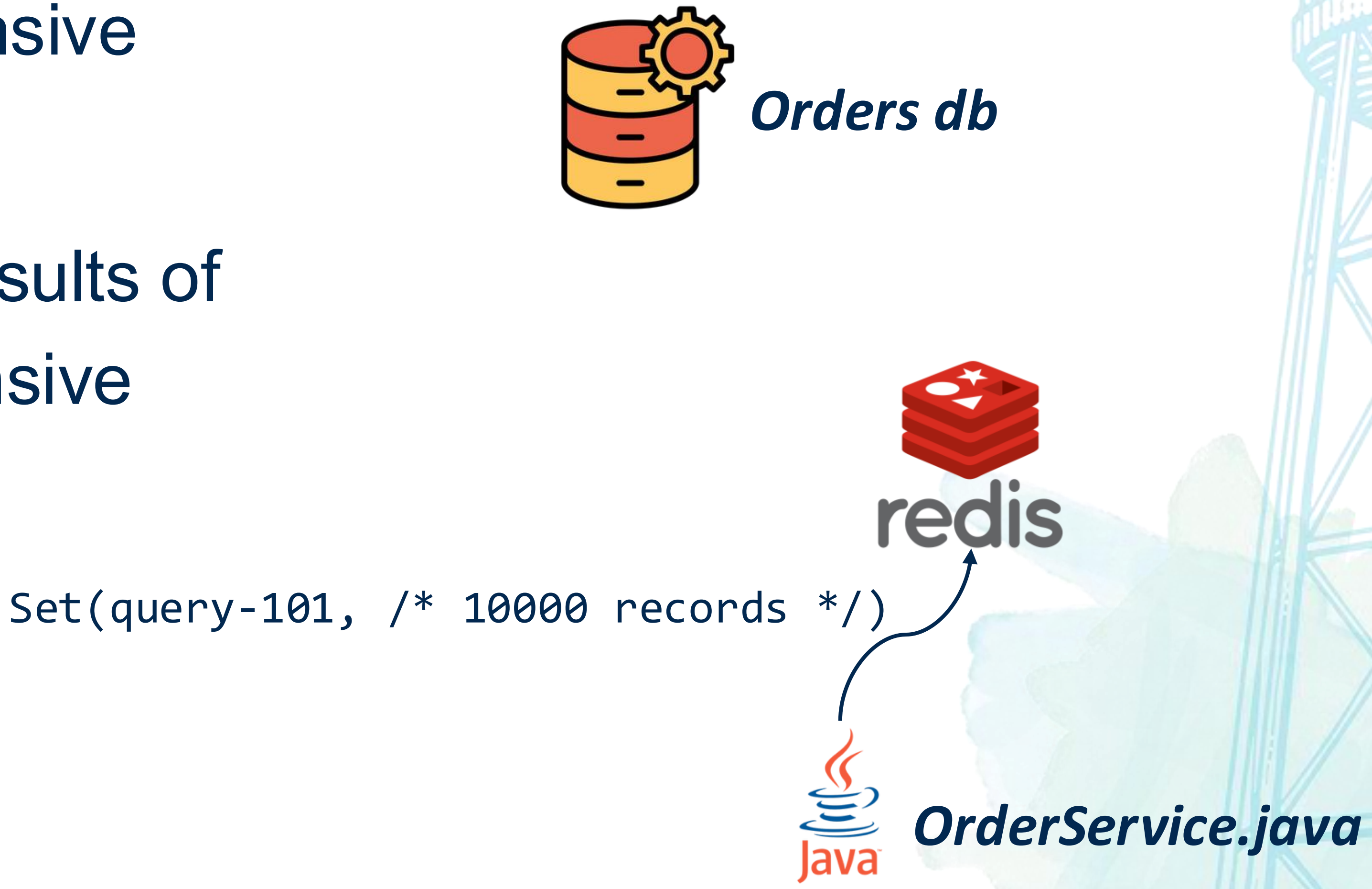
# Key-value stores

- Cache results of expensive operations
- For e.g. caching the results of API requests, or expensive database queries



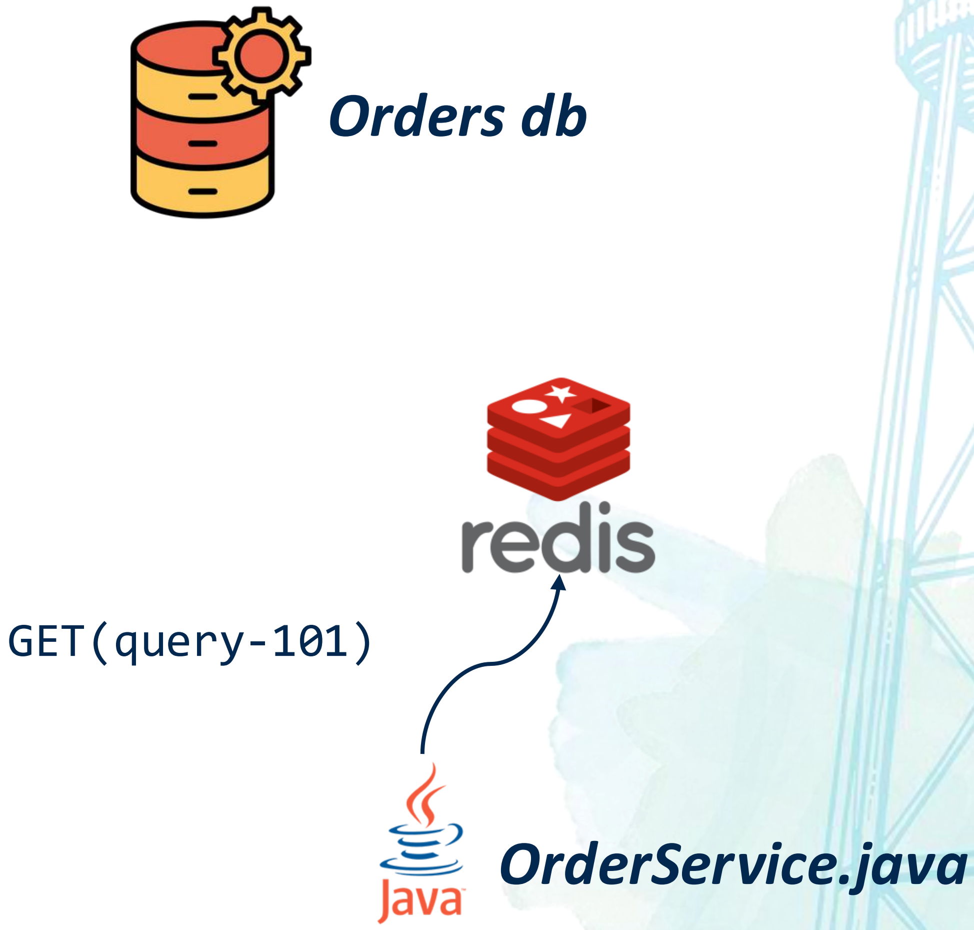
# Key-value stores

- Cache results of expensive operations
- For e.g. caching the results of API requests, or expensive database queries



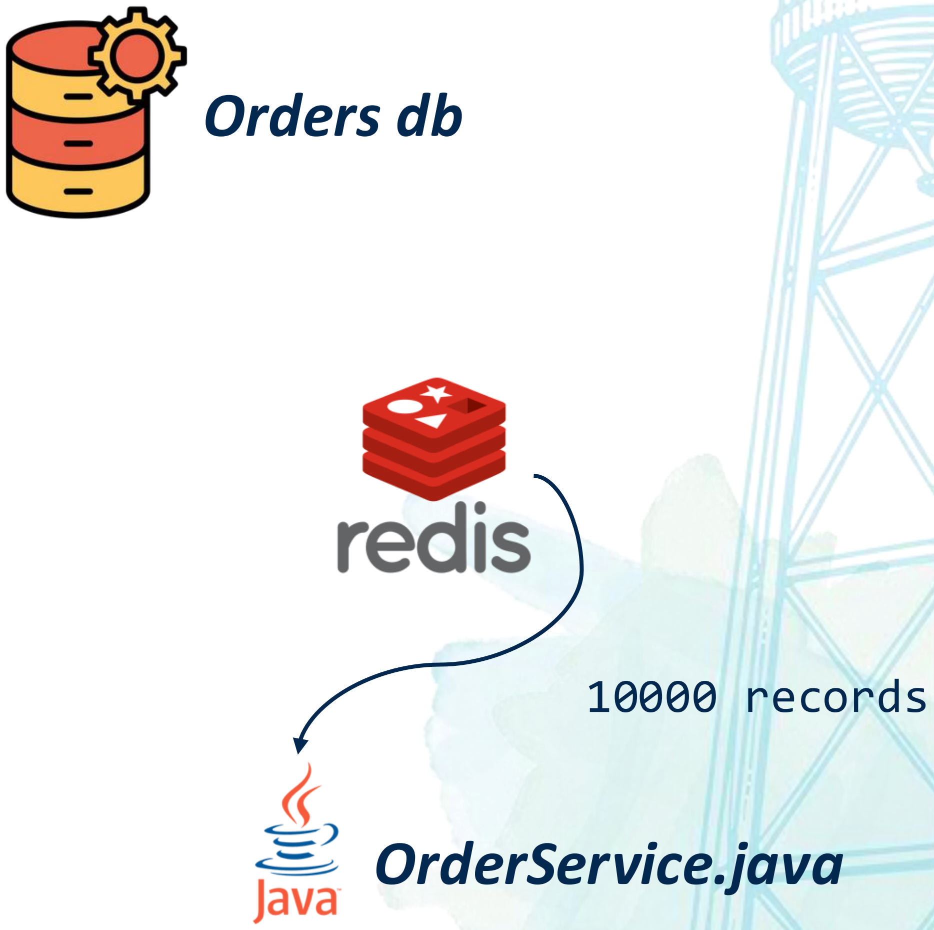
# Key-value stores

- Cache results of expensive operations
- For e.g. caching the results of API requests, or expensive database queries



# Key-value stores

- Cache results of expensive operations
- For e.g. caching the results of API requests, or expensive database queries



# Database choice dimensions

- Data model
  - Format of data user gives to the database
  - Examples - relational, document, graph, key-value
  - Shapes query semantics (joins, traversals, etc.)
    - Mechanism through which you can retrieve the data
- Storage engine
  - How data is physically stored and indexed
  - B-Trees, SSTables and LSM-Trees, Hash Indexes
  - Impacts performance tradeoffs (read vs. write performance, range scans, etc)

# Storage engines

- Log-structured storage engines
  - Bitcask (for Riak distributed system)
  - Apache Cassandra, LevelDB, RocksDB
- Page-oriented storage engines
  - Most relational databases – MySQL, Postgresql, etc.

# Database as an immutable log

- A log is an append-only data structure
- Example, store JSON as a log
- Access complexities
  - Insert/update is append is O(1)
  - Search is O(n)

```
{
    “title”: “ABC”,
    “tags”: [“S”, “T”],
    “author”: “X”
}
```

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | : | { | “ | t | i | t | l | e | “ | : |
| “ | A | B | C | “ | , | “ | t | a | g | s | “ |
| : | [ | ” | S | “ | , | “ | T | “ | ] | , | “ |
| a | u | t | h | o | r | “ | : | “ | X | “ | } |
| 0 | 2 | : | { | … | | | | | | | |

***Log***

UCDAVIS

# Hash indices

- Naive design: in-memory hash map stores key and log-offset as value
- O(1) insert, update, and search

| Key | Value |
|---|---|
| 01 | ■ |
| 03 | ■ |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | **1** | : | { | “ | t | i | t | l | e | “ | : |
| “ | A | B | C | “ | , | a | u | t | h | o | r |
| “ | : | “ | X | “ | } | \n | **0** | **3** | : | { | “ |
| t | i | t | l | e | “ | : | “ | X | Y | Z | “ |
| , | “ | a | u | t | h | o | r | “ | : | “ | Y |
| ” | } | | | | | | | | | | |

*Log*

# Hash indices – update operation

- Update operation consists of an append and an update of the hash index
  - Previous records immutable
- E.g. update(03)

| Key | Value |
|---|---|
| 01 | |
| 03 | |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | : | { | “ | t | i | t | l | e | “ | : |
| “ | A | B | C | “ | , | a | u | t | h | o | r |
| “ | : | “ | X | “ | } | \n | 0 | 3 | : | { | “ |
| t | i | t | l | e | “ | : | “ | X | Y | Z | “ |
| , | “ | a | u | t | h | o | r | “ | : | “ | Y |
| ” | } | 0 | 3 | : | { | “ | t | i | t | l | e |
| “ | : | “ | P | Q | R | “ | , | “ | a | u | t |
| h | o | r | “ | : | “ | Y | “ | } | | | |

*Log*

# Compaction of hash indices

- Divide log into segments
- Reduce disk usage by periodically removing duplicates
- Compaction generates a new segment
- Used by Bitcask DB

| Key | Value |
|---|---|
| 01 | |
| 03 | |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | : | { | “ | t | i | t | l | e | “ | : |
| “ | A | B | C | “ | , | a | u | t | h | o | r |
| “ | : | “ | X | “ | } | \n | 0 | 3 | : | { | “ |
| t | i | t | l | e | “ | : | “ | P | Q | R | “ |
| , | “ | a | u | t | h | o | r | “ | : | “ | Y |
| ” | } | | | | | | | | | | |

*Compacted Log Segment*

# Compaction of hash indices

- Compaction runs in a background thread
- Compaction can be expensive

| Key | Value |
|---|---|
| 01 | |
| 03 | |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | **1** | : | { | “ | t | i | t | l | e | “ | : |
| “ | A | B | C | “ | , | a | u | t | h | o | r |
| “ | : | “ | X | “ | } | \n | **0** | **3** | : | { | “ |
| t | i | t | l | e | “ | : | “ | P | Q | R | “ |
| , | “ | a | u | t | h | o | r | “ | : | “ | Y |
| ” | } | | | | | | | | | | |

*Log Segment*

# Limitations of approach

- High memory usage for large logs
  - A sparse hash index can overcome this limitation, but needs **sorting**
- Hash indices typically maintained only for the latest log segment
- Range queries are not natively supported
  - Find all records with id in range [02 – 20] must scan all records
  - **Sorting** the records can solve this limitation, too
- Compaction can be expensive
- Solution: SSTables (memtable) and LSM trees

# Log structured merge tree (LSM tree)

- Designed by Google as part of their distributed database BigTable
  - Chang, Fay, et al. "Bigtable: A distributed storage system for structured data." *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008)
- Adopted by many recent databases (distributed or single-machine)

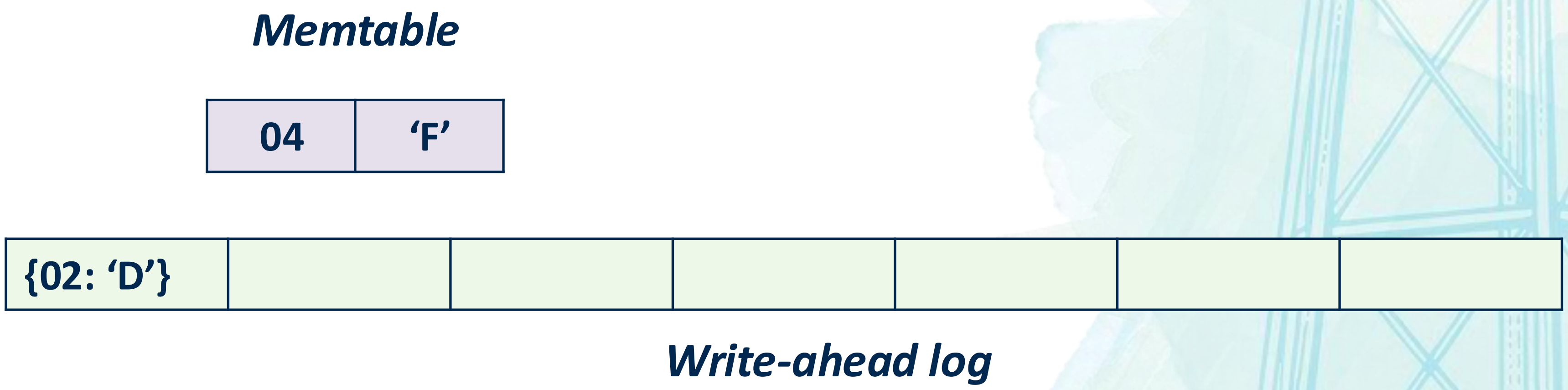
# Log structured merge tree (LSM tree)

- LSM tree components
  - On-disk write-ahead log (WAL)
  - In-memory sorted tree (memtable)
  - Immutable on-disk sorted string table (SSTable)
- **Core idea:** sort in-memory before storing on disk

# LSM tree operations

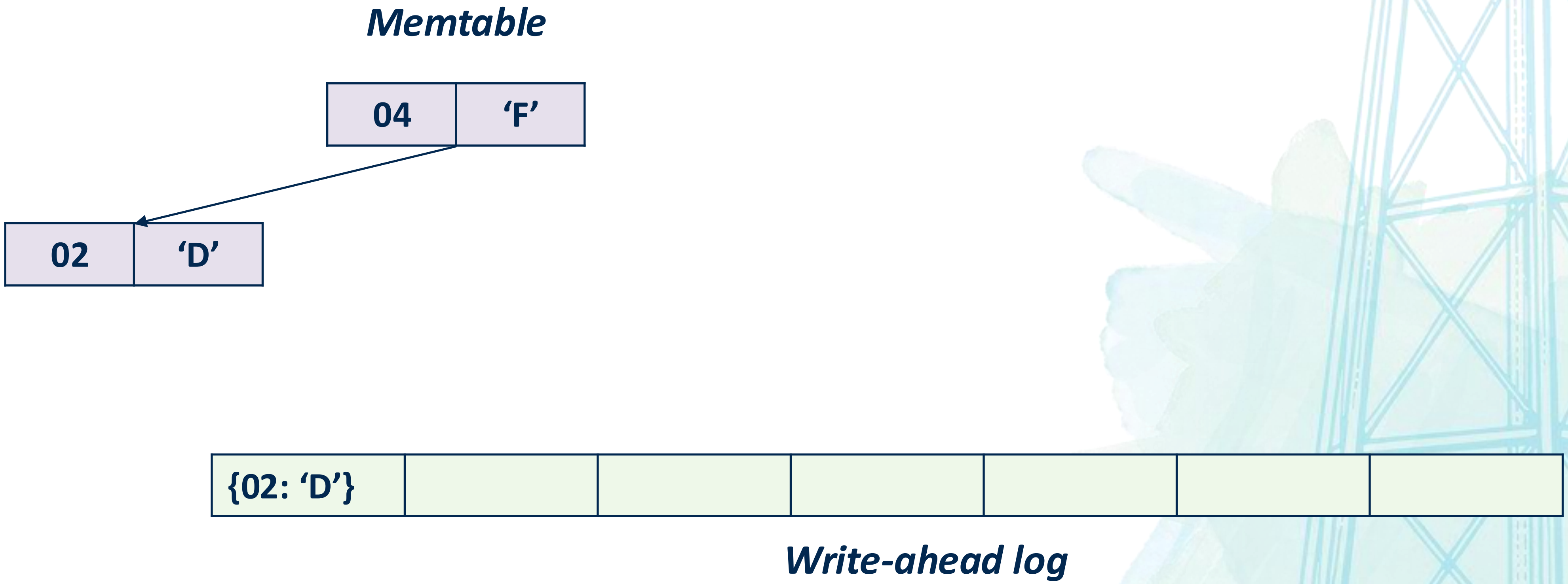
# Write-ahead log

- On-disk log that provides crash recovery abilities
  - System crashes can delete the in-memory sorted tree
  - Solution: write out the updates to a log on the disk before inserting into tree
- During crash recovery replay the write-ahead log

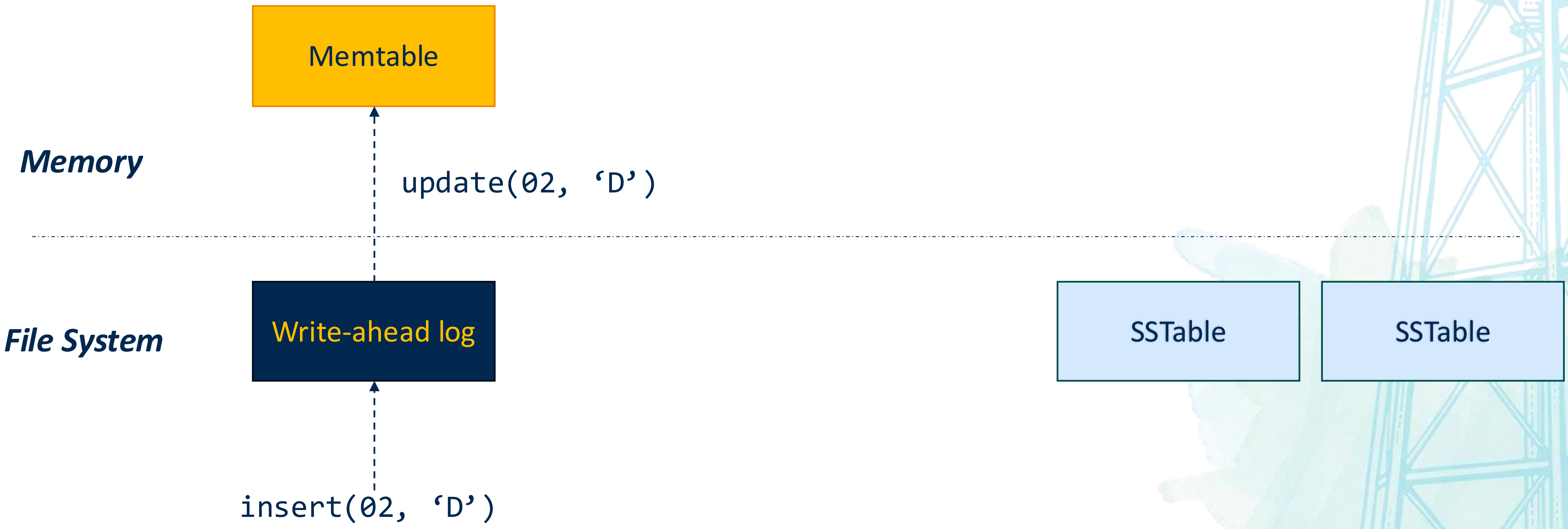


# Memtable

- Maintain a sorted in-memory tree (AVL or red-black tree)
- Writes update the in-memory tree

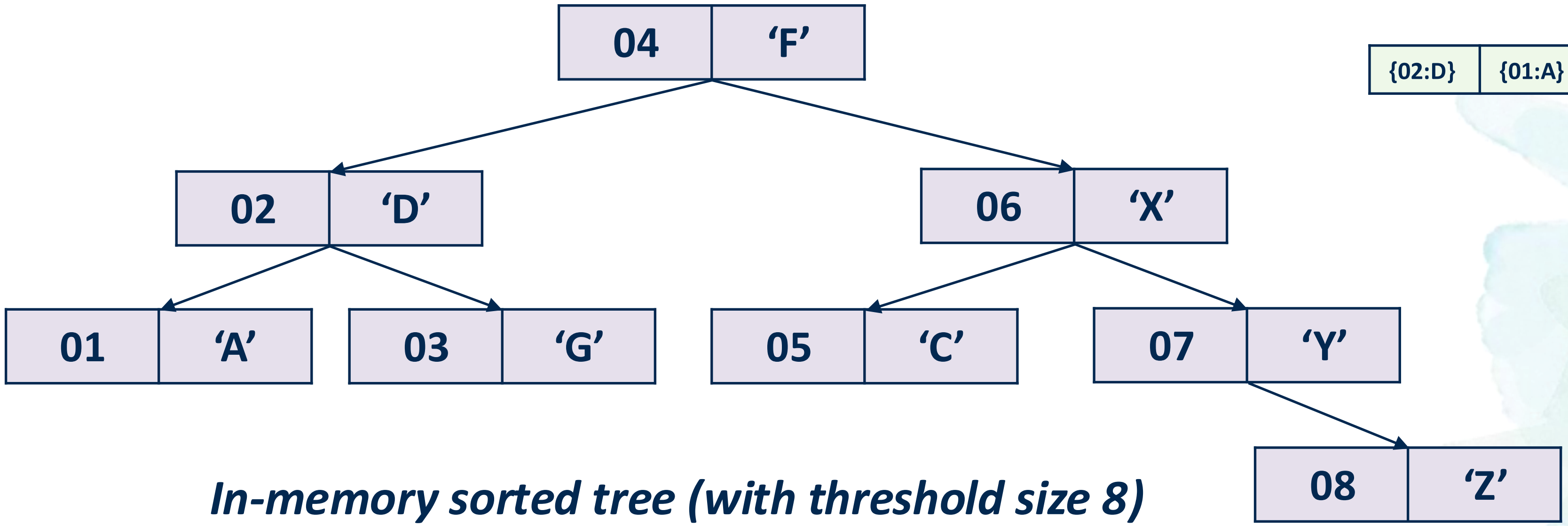


# LSM tree write operation



# Memtable

- Write contents to log segment (SSTable) when tree size reaches threshold
- Threshold is typically a few MB




*In-memory sorted tree (with threshold size 8)*


*Write-ahead log*

# Memtable

- Write sorted contents from Memtable to log segment (SSTable) when tree size reaches threshold

- Clear the WAL

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | : | { | ‘A’ | } | \n | 0 | 2 | : | { | ‘D’ |
| } | \n | 0 | 3 | : | { | ‘G’ | } | \n | 0 | 4 | : |
| { | ‘F’ | } | \n | 0 | 5 | : | { | ‘C’ | } | \n | 0 |
| 6 | : | { | ‘X’ | } | \n | 0 | 7 | : | { | ‘Y’ | } |
| 0 | 8 | : | { | ‘Z’ | } | | | | | | |

*Sorted segment file (on disk)*

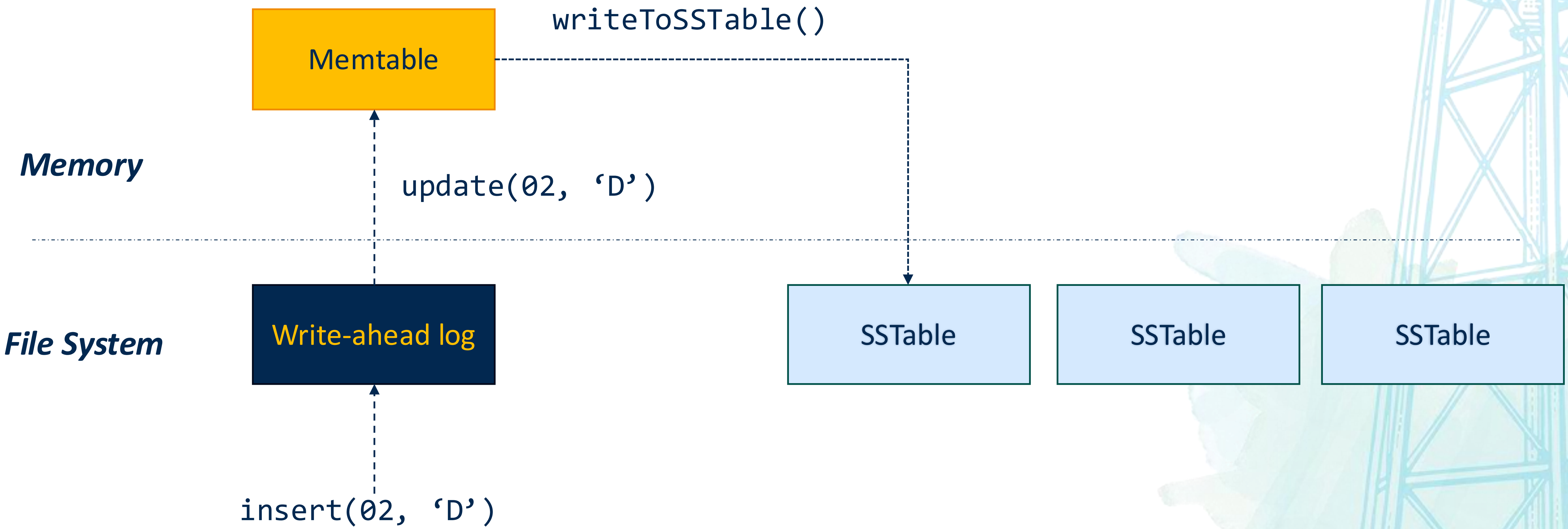
| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | |

*Write-ahead log*

# Memtable crash recovery

- Memtable lives in memory
  - System crashes/reboots lose its contents
- WAL provides crash recovery if system crashes before Memtable written to disk
- On reboot, replay the WAL records

# LSM tree write operation



# SSTables compaction

- Recall: SSTables are immutable
- Updates insert new records with same key
- Needs compaction to reduce on-disk storage size

*Segment 1*

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | : | { | ‘A’ | } | \n | 0 | 4 | : | { | ‘B’ |
| } | \n | 0 | 5 | : | { | ‘C’ | } | … | | | |

*Segment 2*

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | : | { | ‘D’ | } | \n | 0 | 3 | : | { | ‘E’ |
| } | \n | 0 | 4 | : | { | ‘F’ | } | … | | | |

*Segment 3*

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 3 | : | { | ‘G’ | } | \n | 0 | 4 | : | { | ‘H’ |
| } | | | | | | | | | | | |

# SSTables compaction

- Recall: SSTables are immutable
- Updates insert new records with same key
- Needs compaction to reduce on-disk storage size

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | : | { | ‘A’ | } | \n | 0 | 2 | : | { | ‘D’ |
| } | \n | 0 | 3 | : | { | ‘G’ | } | \n | 0 | 4 | : |
| { | ‘F’ | } | \n | 0 | 5 | : | { | ‘C’ | } | \n | |

***Merged Segment***

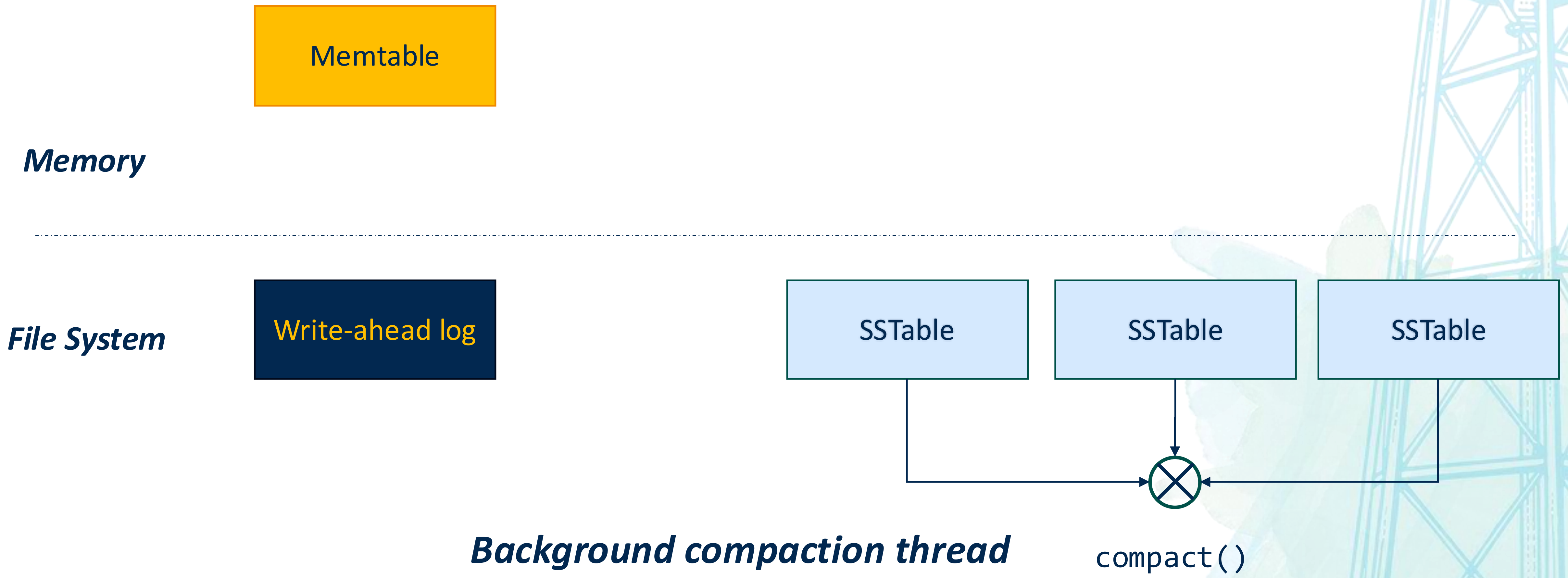
# SSTables compaction

- The records in each log segment is sorted by key
- Advantages: compaction is faster (essentially the ”merge phase” of a merge-sort algorithm)

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | : | { | ‘A’ | } | \n | 0 | 2 | : | { | ‘D’ |
| } | \n | 0 | 3 | : | { | ‘G’ | } | \n | 0 | 4 | : |
| { | ‘F’ | } | \n | 0 | 5 | : | { | ‘C’ | } | \n | |

***Merged Segment***

# LSM tree compaction operation



# LSM tree compaction operation

Memtable

***Memory***

***File System***

Write-ahead log

New Merged SSTable

***New merged SSTable can participate in further compactions***

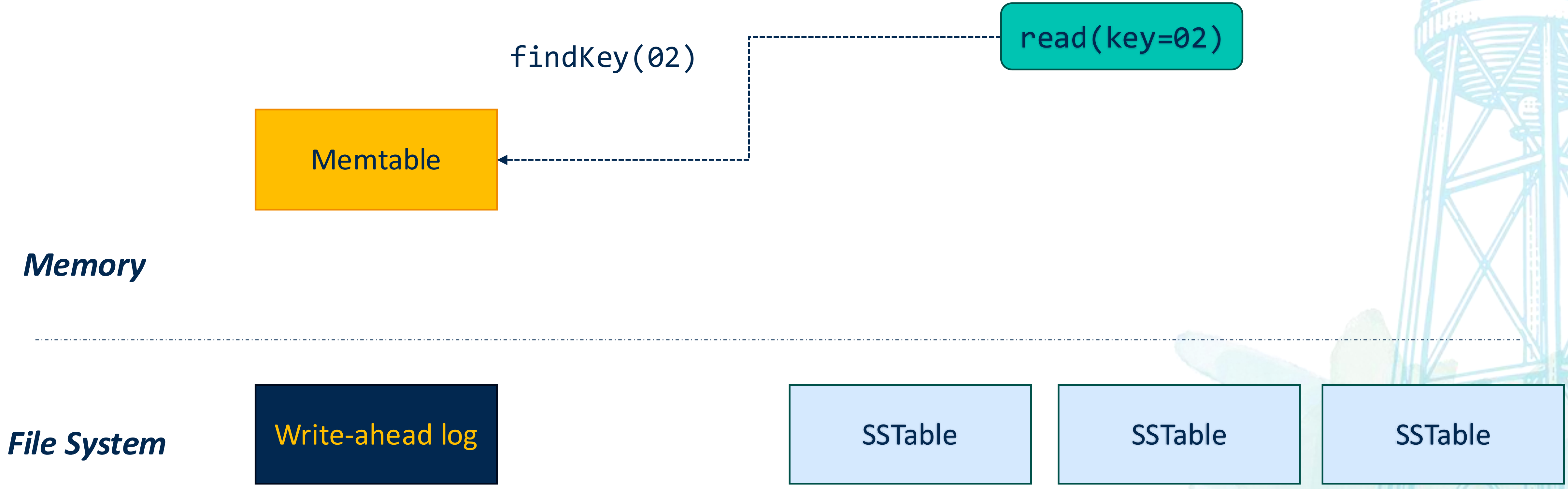
# Typical overheads of LSM write path

- Append to ***on-disk*** WAL has O(1) complexity
- Insert into ***in-memory*** memtable (AVL tree) has  O(log(n)) complexity
- Eventually, append to ***on-disk*** SSTable has O(1) complexity per-key
  - Bulk write amortizes the disk overhead
- ***On-disk*** log compaction and merging runs in a background thread
- Generally, writes are fast compared to alternatives
- Still, at least 3x write amplification

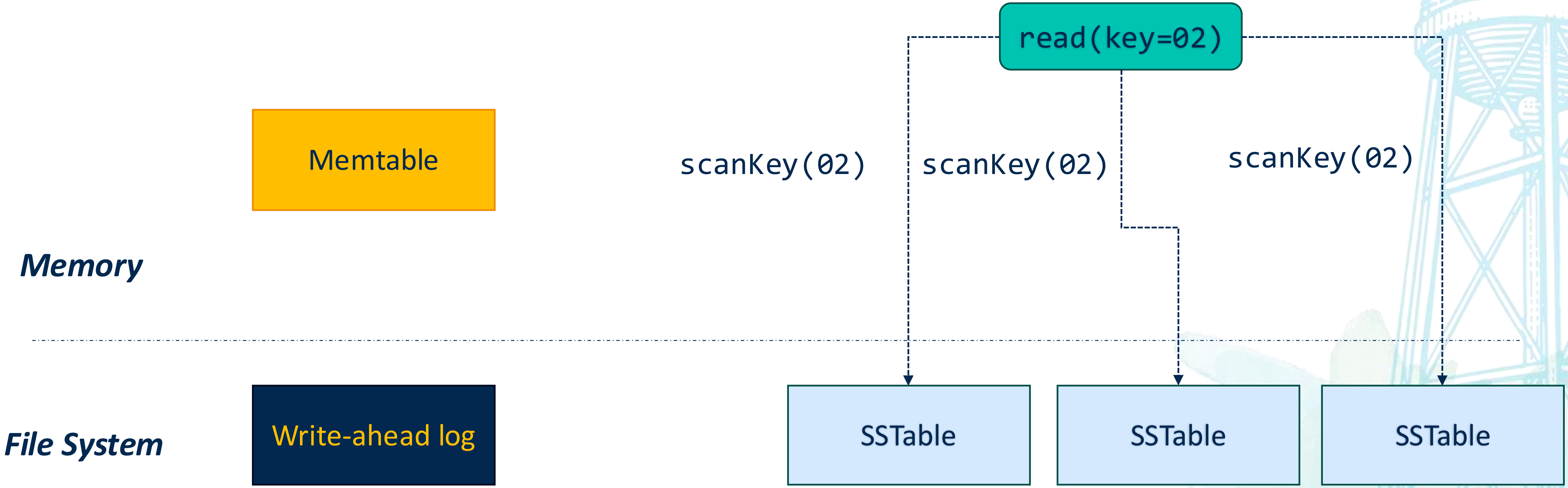
# LSM tree read operation

- Check if the key is in the ***in-memory*** memtable
  - Typically, O(log(n)) complexity
- If not, scan each of the ***on-disk*** SSTables has the key
  - Typically, O(n) complexity
- Generally, reads are slower than writes

# LSM tree read operation



# LSM tree read operation



# Block index for read optimization

- Each SSTable is divided into blocks
- Each SSTable has a block index which maps keys to block offsets inside the SSTable

***Block Index***

| Key | Offset |
|---|---|
| 01 | |
| 06 | |

***Block-0***

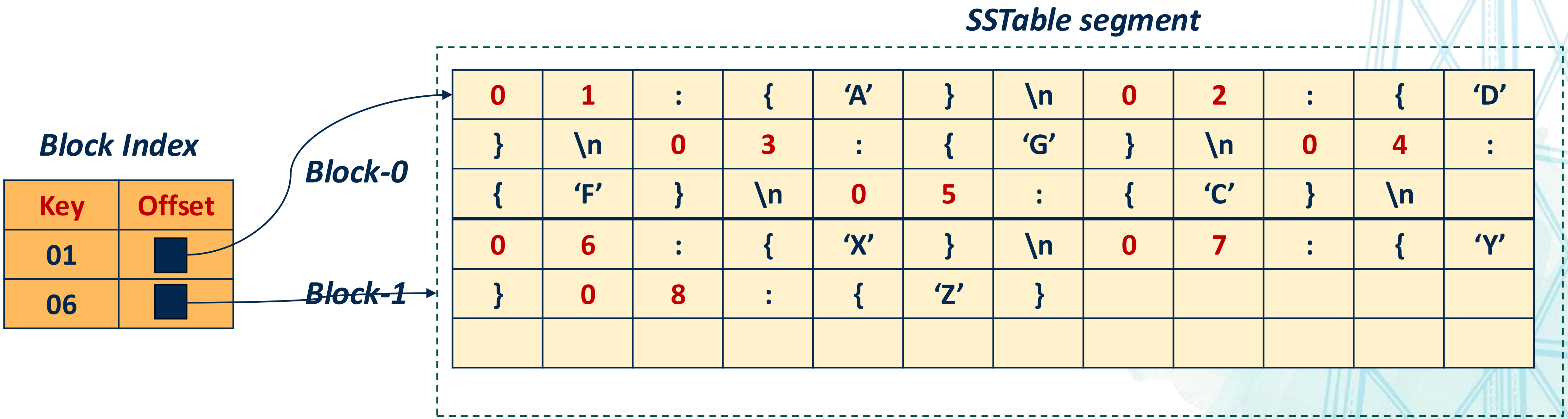
***Block-1***

***SSTable segment***

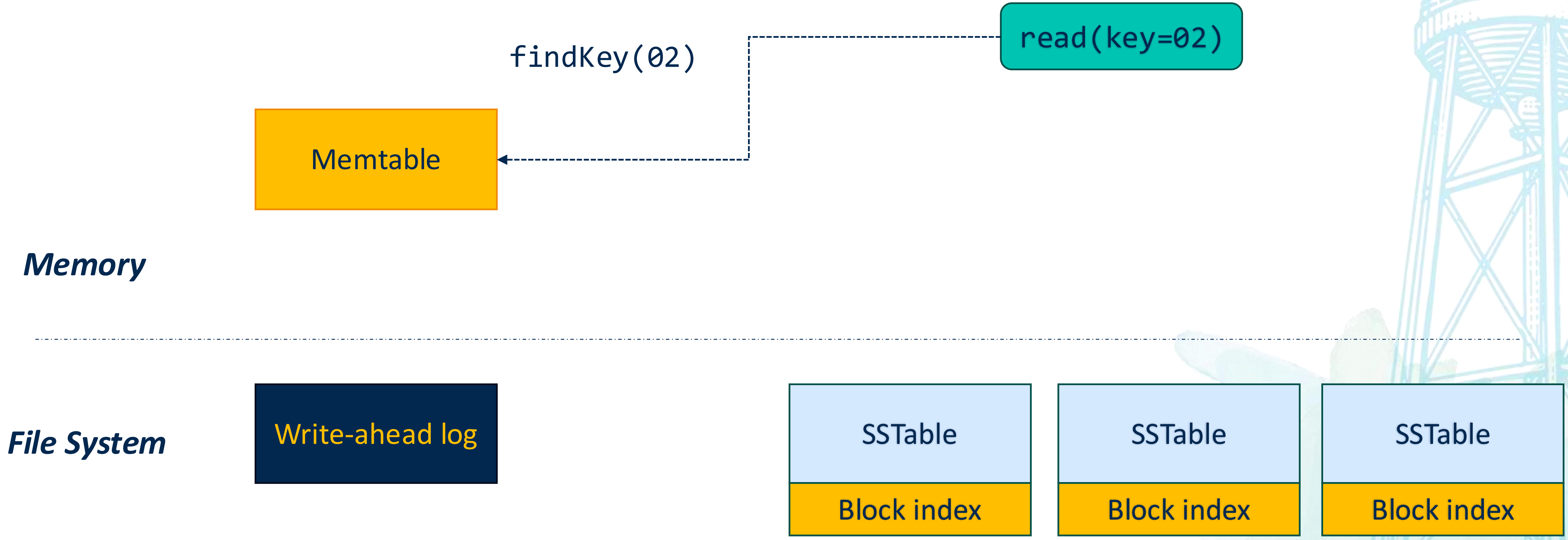
| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | : | { | ‘A’ | } | \n | 0 | 2 | : | { | ‘D’ |
| } | \n | 0 | 3 | : | { | ‘G’ | } | \n | 0 | 4 | : |
| { | ‘F’ | } | \n | 0 | 5 | : | { | ‘C’ | } | \n | |
| 0 | 6 | : | { | ‘X’ | } | \n | 0 | 7 | : | { | ‘Y’ |
| } | 0 | 8 | : | { | ‘Z’ | } | | | | | |
| | | | | | | | | | | | |

# Block index for read optimization

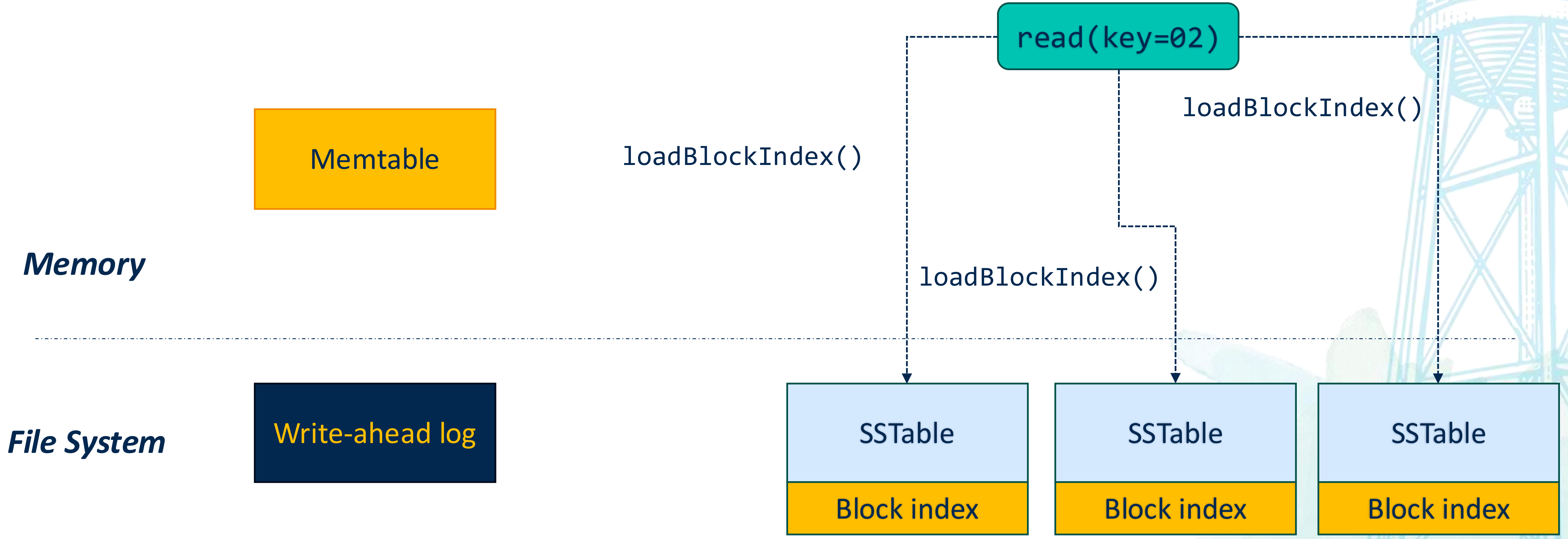
- When scanning SSTable for reads, load block index into memory
- Use block index to index into SSTable



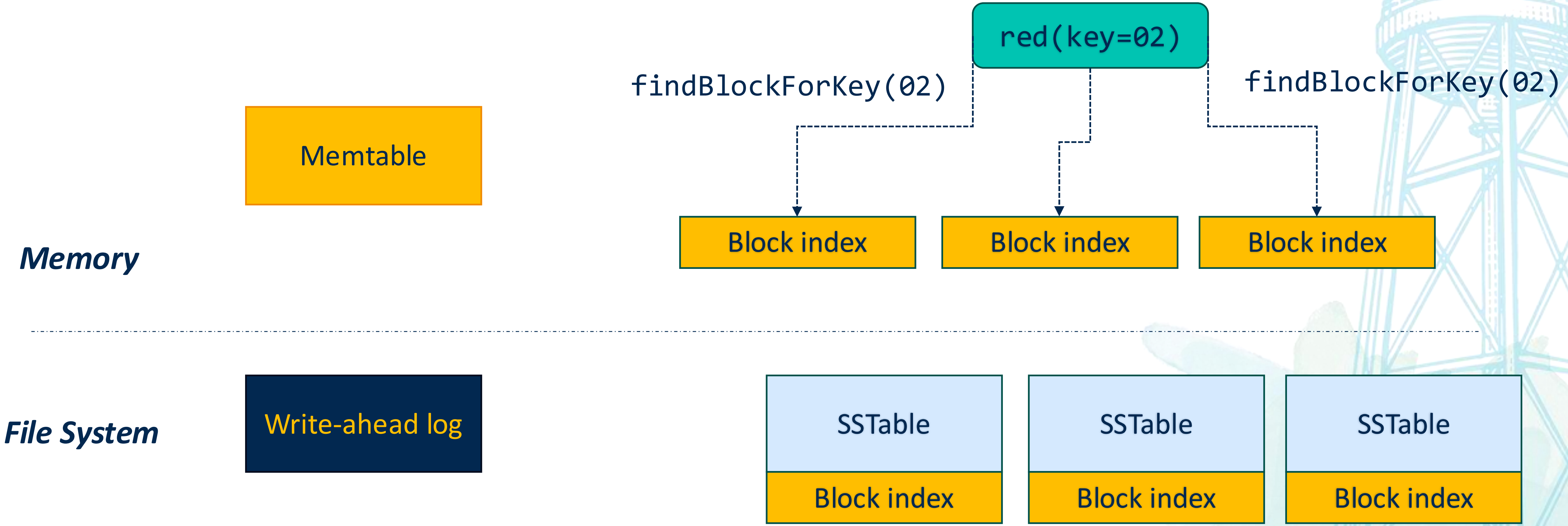
# LSM tree read path with block index



# LSM tree read path with block index

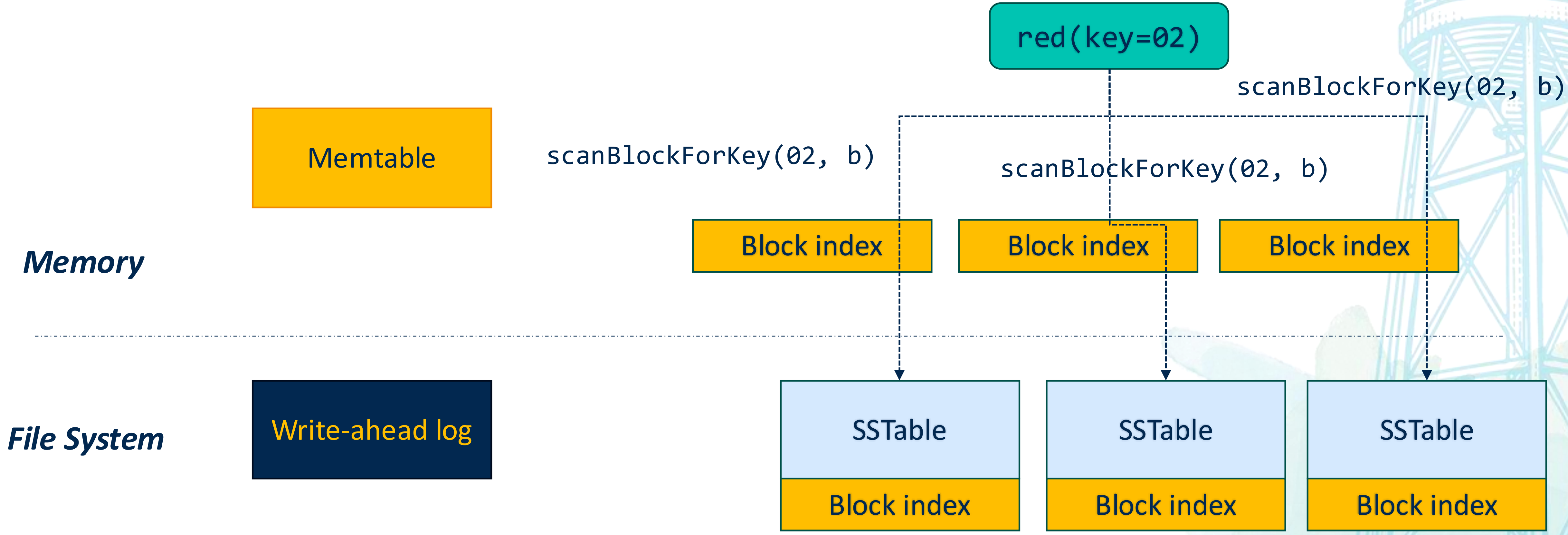


# LSM tree read path with block index




***Scan the block indices to find the corresponding SSTable blocks where the key could potentially be located***

# LSM tree read path with block index




***Scan only the selected blocks in each SSTable***

# Page-oriented storage engine (B+ trees)

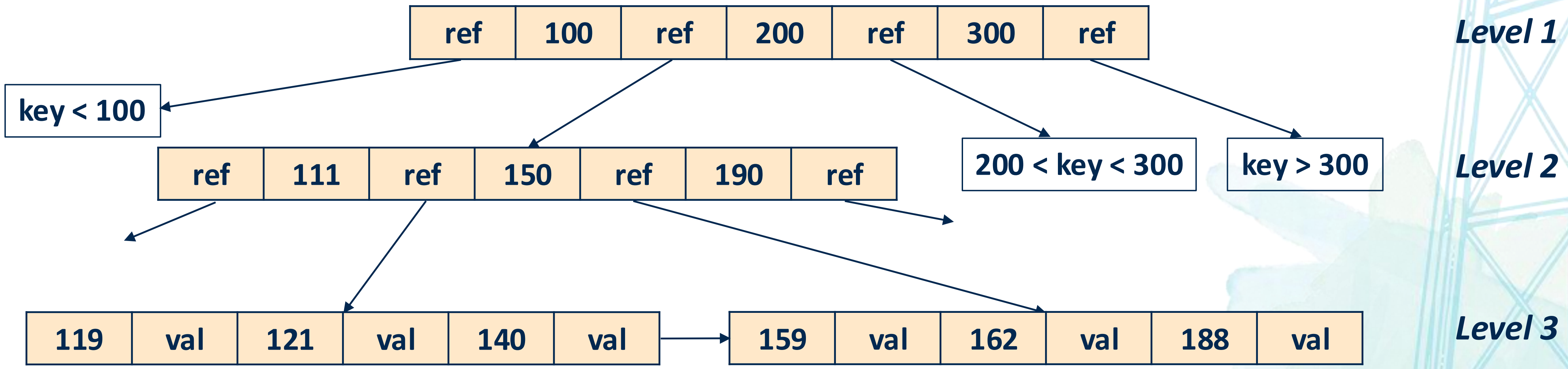
- Used to implement primary key indexes in traditional relational databases
- On-disk data structure that keeps key-value pairs sorted by key
- Supports random accesses
- Writes are performed directly on this on-disk sorted tree, unlike in LSM trees which operated on the in-memory Memtable

# B+ tree

- Recall: B+ trees are ”inspired” by binary search tree
  - B+ trees have multiple branches
- Each node has a max and minimum number of keys
  - Branching factor – how many children each internal node has
- Data only lives in the leaf nodes, intermediary nodes help navigation
- Leaf nodes maintain a linked list for better range scans
- Each node is stored on a “page” of 4 KB size

# B+ tree

- Example B+ tree with three levels and branching factor of 4



# B+ tree read path

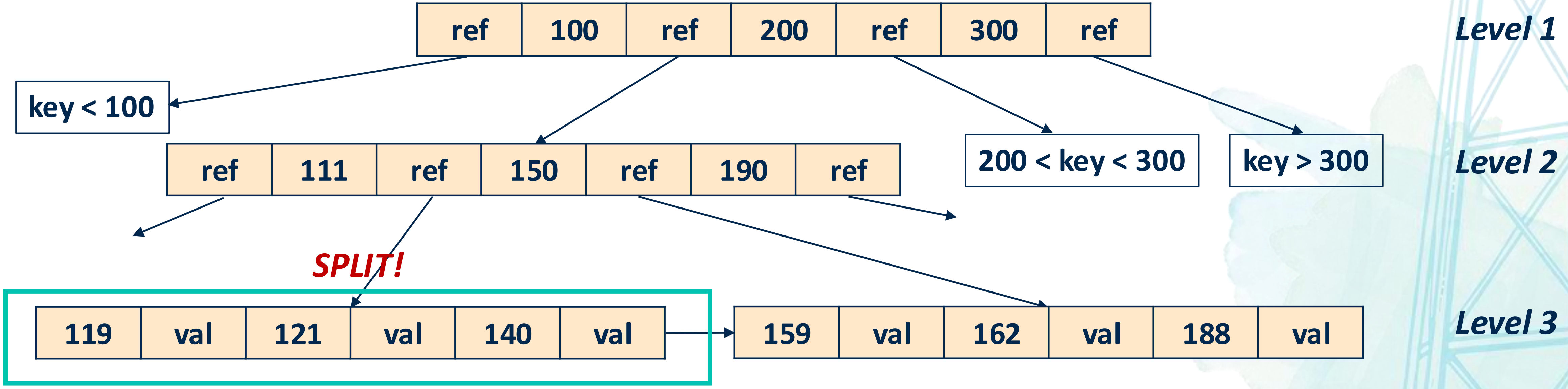
- Traverse the tree following the branches based on the key value
  - Start at the root
  - At each node: search the sorted keys (e.g., binary search) to find the correct child pointer
  - Follow the pointer to the next node and repeat until you reach the leaf
- Complexity is O(log(n))

# B+ tree write path

- Recall – every B+ tree has a max number of keys per node
- If insertion exceeds this max number, the node must be split and tree rebalanced
- Node splitting and tree balancing are expensive operations **performed on disk**

# B+ tree node splitting

- Assume max number of keys per node is 3
- insert(152)



# LSM tree vs B+ tree

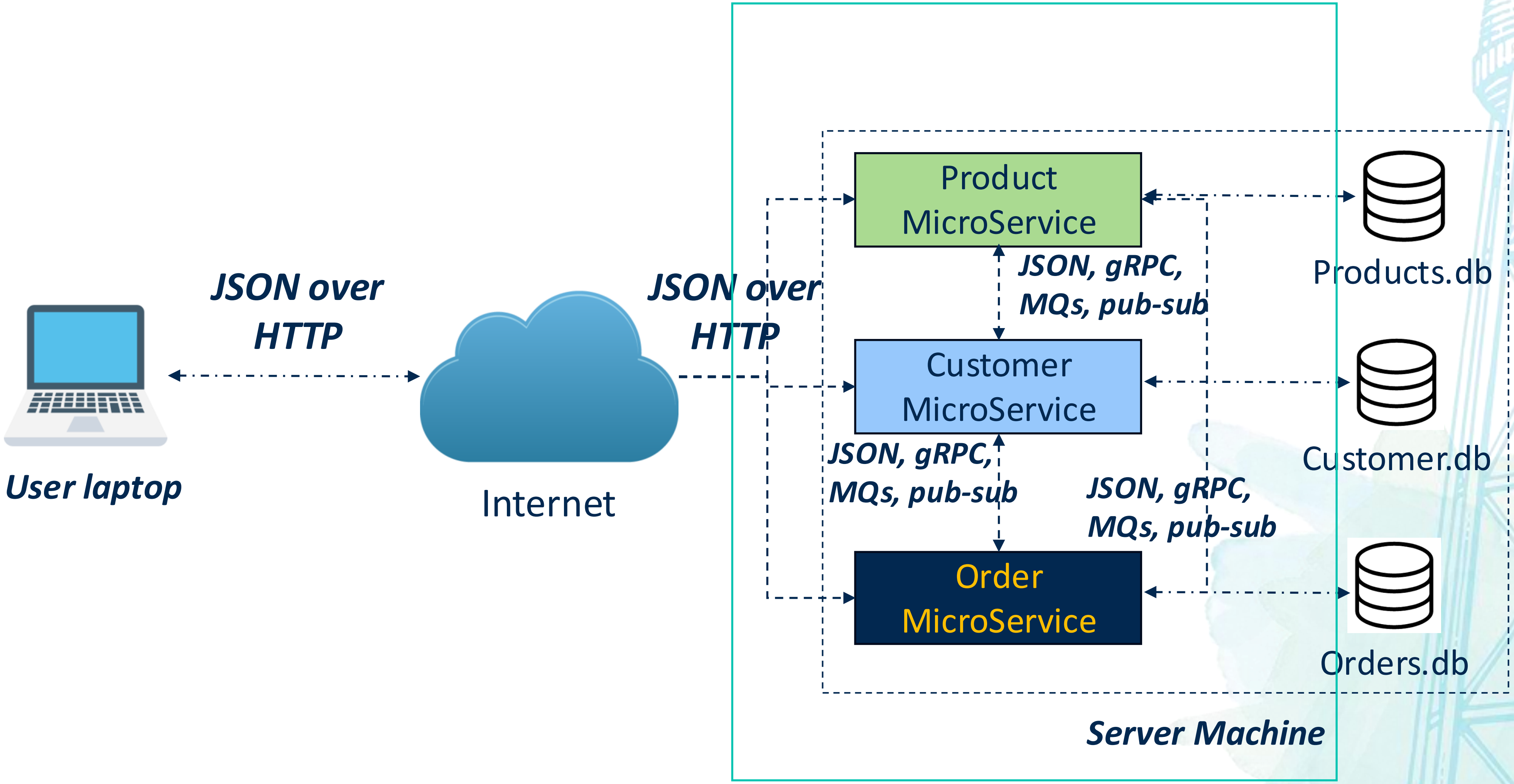
- LSM trees typically have higher write throughput
- LSM trees can compress better, B+ trees can cause fragmentation
- B-trees typically have better read performance
  - The key can exist at only one place, unlike LSM trees which involve scanning multiple SSTables
- No clear winner – requires empirical testing on a per use-case basis

# Summary

- Decomposing monolithic apps into microservices allows per-microservice database selection
- Read-heavy and write-heavy microservices can choose which db to use according to its needs

# Communication styles

# Intra-service communication

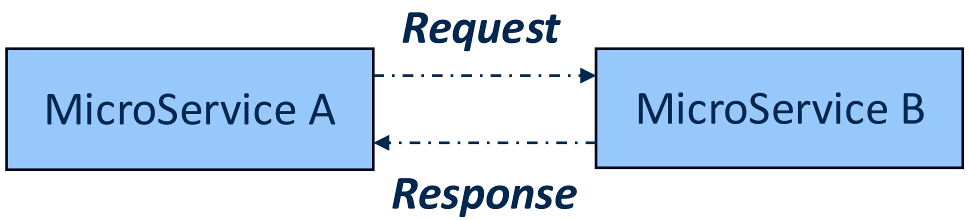


# Communication choice dimensions

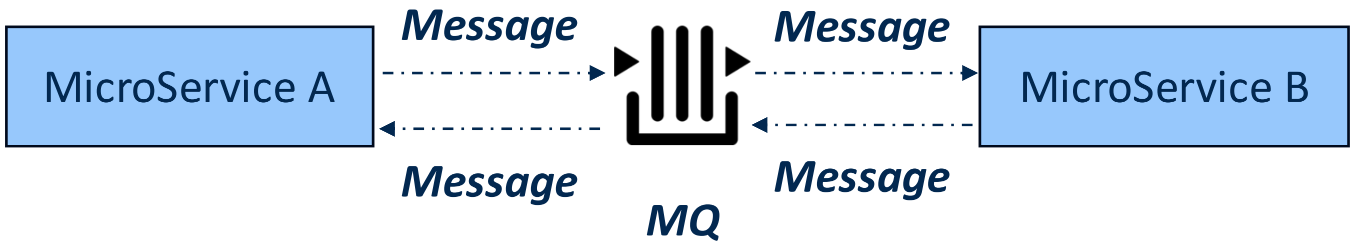
- Synchronous request/response (JSON over HTTP, gRPC)
- Asynchronous request/response (message queues - RabbitMQ)
- Event-driven publish/subscribe models

# Communication choices

- Synchronous request/response (JSON over HTTP, gRPC)

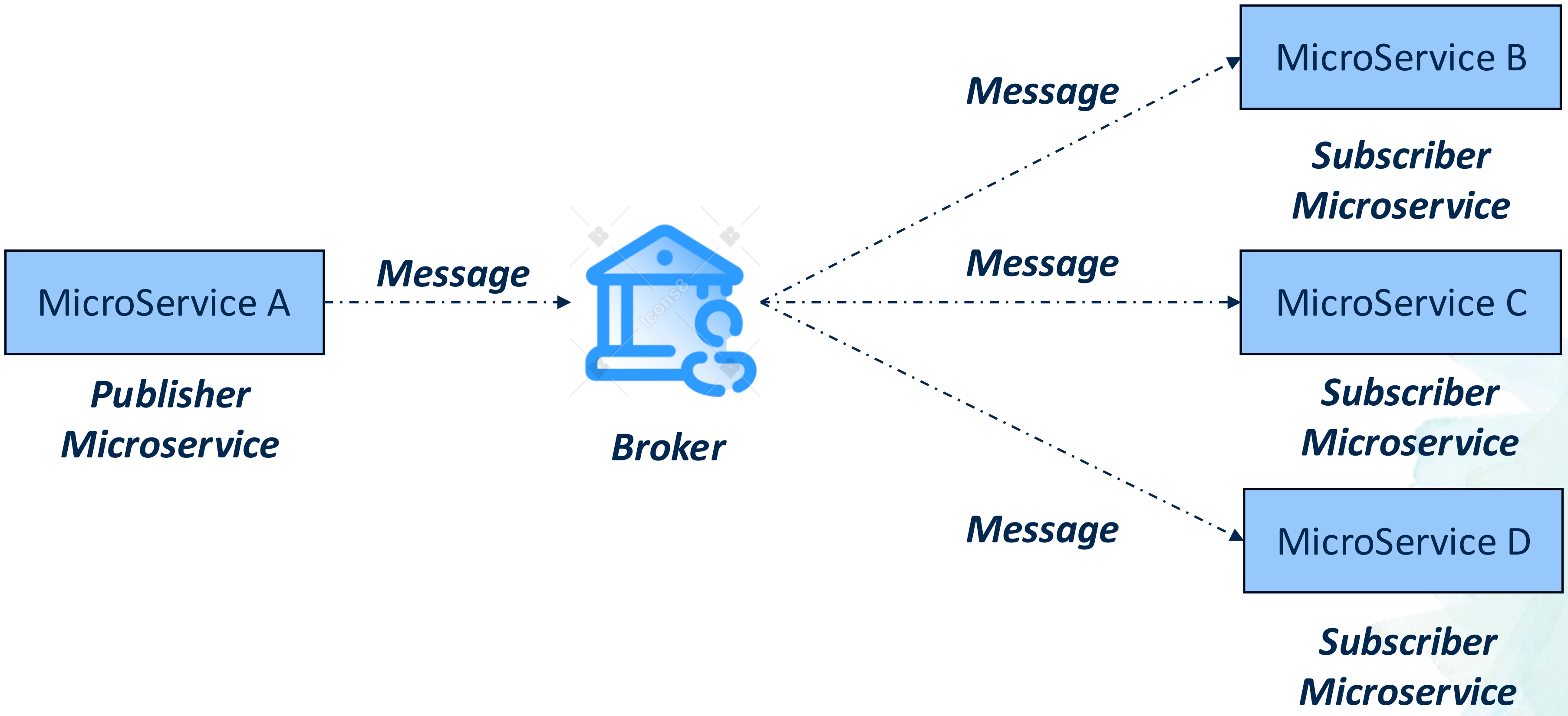



- Asynchronous request/response (message queues - RabbitMQ)



# Communication choices

- Event-driven publish/subscribe models

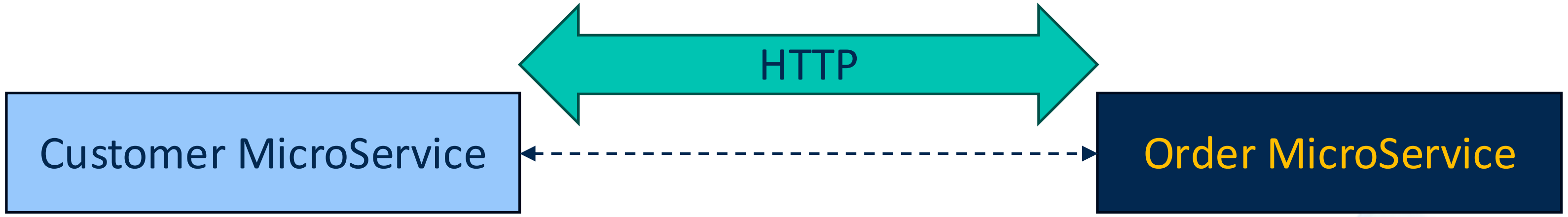


# Synchronous request/response

- Designed to make a remote call feel like a local function call
- Same programming model as local function calls
  - Caller sends a request, waits for a response
  - Typically, it is a blocking call
- JSON over HTTP, gRPC
- Note: the "synchronous" here refers to the communication paradigm or interaction pattern, not how the programmer *uses* the paradigm
  - Programmer can use async programming primitives to communicate with synchronous communication substrates

# JSON over HTTP

```
{
  "id": 123,
   "name": "John Doe",
   "email": john.doe@example.com
}
```



# REST APIs

- Application/microservice exposes an URL
- Uses HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources
- Commonly paired with JSON for data exchange

```
// GET Request to Fetch a User denoted by ID

> GET https://api.github.com/users/123

// Response

{
    "id": "123",
    "name": "John Doe",
    "email": john.doe@example.com

}
```

# HTTP JSON client

- Example Java HTTP client communicating with GitHub API endpoint

```
HttpClient client = HttpClient.newHttpClient();

String jsonBody =
    {"title":"Found a bug",
     "body":"Steps to reproduce..."};

HttpRequest req = HttpRequest.newBuilder()
    .uri(URI.create(
    "http://api.github.com/repos/ecs160/issues"))
    .header("Content-Type", "application/json”).
    .POST(jsonBody).build();

HttpResponse<String> resp = client.send(req, ..);

System.out.println("Status: " +
resp.statusCode());

System.out.println("Body: " + resp.body());
```

# Async HTTP JSON client (Not in syllabus)

- Can use language-level async primitives with synchronous communication mechanisms
- Python, Rust, and other languages provide async I/O through coroutines
- Java provides ”futures”

```
import asyncio

async def fetch(url: str) -> str:
    # Simulate network I/O
    await asyncio.sleep(1)
    return f"fetched: {url}"

async def main():
    # Start multiple coroutines concurrently
    tasks = [asyncio.create_task(fetch(u)) for u in
["a", "b", "c"]]

    # Do other work while they run
    for i in range(3):
        print("doing other work", i)
        some_other_long_running_task()

    # Await results (does not block the OS thread)
    results = await asyncio.gather(*tasks)
    print(results)

asyncio.run(main())
```

# JSON implications

- JSON message contains the schema
- What happens if the receiver receives a message with the `email` field missing?
  - Receiver can still make sense of the rest of the fields
  - Receiver can ”handle” missing or extra fields

```
// expected
{
    "id": "123",
    "name": "John Doe",
    "email": john.doe@example.com
}

// received
{
    "id": "123",
    "name": "John Doe",
}
```

# JSON implications

- JSON is text-based
- Text-based protocols are less efficient than binary protocols

```
{
    "id": "123",
    "name": "John",
}
```

***Text-based representation***

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| { | “ | i | d | “ | : | 1 | 2 | 3 | , | “ | n |
| a | m | e | “ | : | “ | J | O | H | N | “ | } |

***Binary-based representation***

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 123 | | | | J | O | H | N |

* Assume each box is 8 bits, chars are 8 bits, integers are 32 bits

# JSON implications

- Protocol efficiency: low
- Type safety (schema adherence): low
- What about ease of debugging?
  - Can view exactly the message contents on the wire
- Ease of debugging: high

# gRPC

https://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html

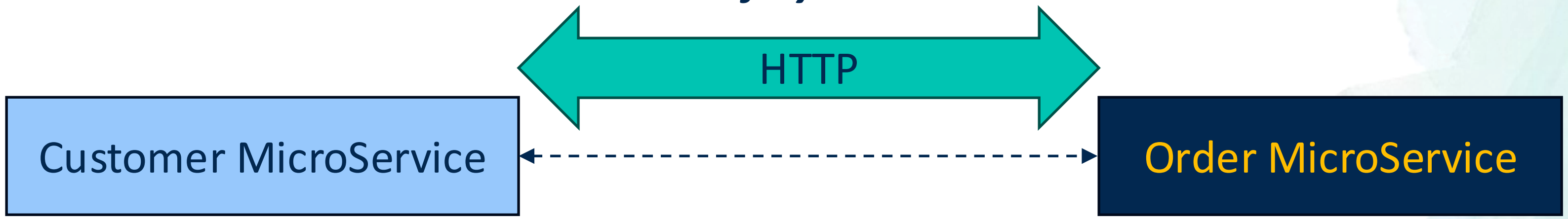
## Google Remote Procedure Call

```
Person(
  userName: "Martin",
  favouriteNumber: 1337,
  interests: ["daydreaming", "hacking"]
)
```

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0a | 4d | 61 | 72 | 74 | 69 | 6e | 10 | b9 | 0a | 1a | 0b |
| 64 | 61 | 61 | 79 | 64 | 72 | 65 | 61 | 6d | 69 | 6e | 67 |
| 1a | 07 | 68 | 61 | 63 | 6b | 6e | 67 | | | | |

protobuf
Protocol Buffers

***Protobuf byte stream***



# Protobuf

- Google’s binary encoding format
- User must define schema in a `.proto` file
- `protoc` compiler generates typed data classes and service stubs across many languages
  - C++, C#, Java, Kotlin, Objective-C, PHP, Python, Ruby

# Protobuf

- Define schema in .proto file
- Protobuf defines standard data types
- Tags are the field's numeric identifier used in the binary format
- Fields are encoded as `(tag, wire-type, value)`

```
Person(
  userName: "Martin",
  favouriteNumber: 1337,
  interests: ["daydreaming", "hacking"]
)
```

**Tag**

```
// myservice.proto
message Person {
    required string user_name       = 1;
    optional int64  favourite_number = 2;
    repeated string interests       = 3;
}
```

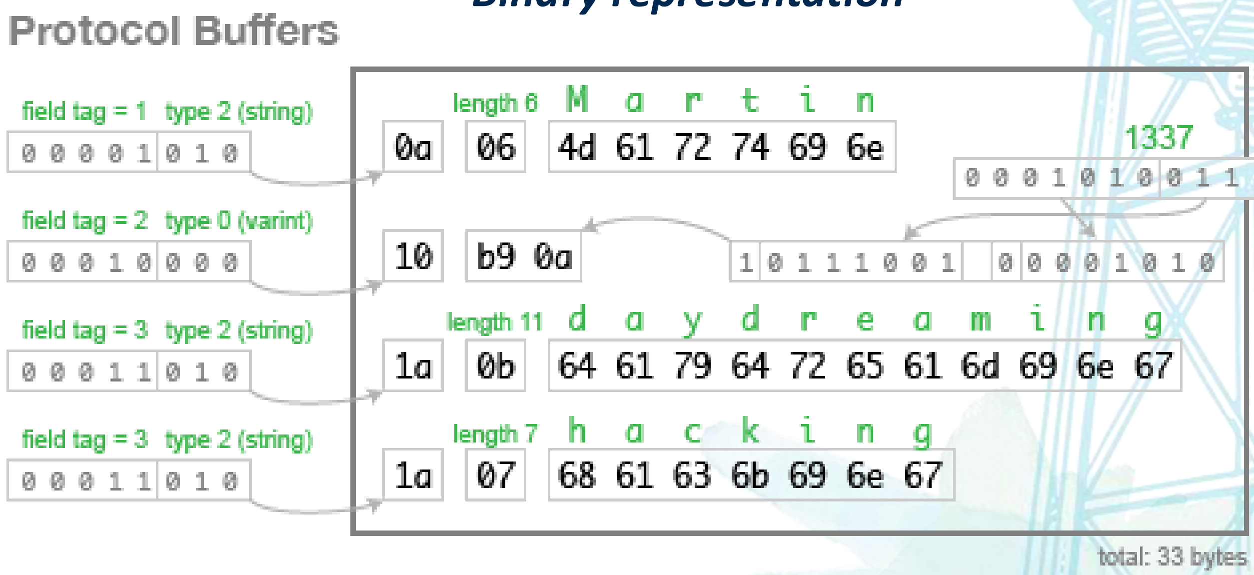
# Protobuf binary format

https://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html

```
// myservice.proto
message Person {
    required string user_name       = 1;
    optional int64  favourite_number = 2;
    repeated string interests       = 3;
}
```

```
Person(
  userName: "Martin",
  favouriteNumber: 1337,
  interests: ["daydreaming", "hacking"]
)
```

***Binary representation***

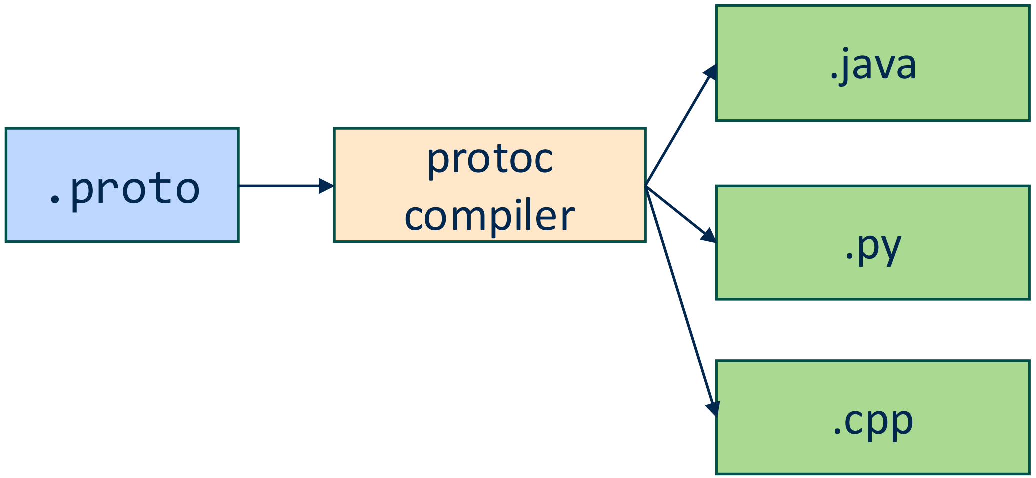



***Binary data on the wire***

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0a | 06 | 4d | 61 | 72 | 74 | 69 | 6e | 10 | b9 | 0a | 1a |
| 0b | 64 | 61 | 61 | 79 | 64 | 72 | 65 | 61 | 6d | 69 | 6e |
| 67 | 1a | 07 | 68 | 61 | 63 | 6b | 6e | 67 | | | |

# Codegen with protoc

- Invoke protoc to generate classes and service stubs
- `protoc -I . myservice.proto --java_out=./gen`




```
{
  "userName": "Martin",
   "favouriteNumber": 1337,
   "interests": ["daydreaming", "hacking"]
}
```

```
// Proto schema
message Person {
    required string user_name        = 1;
    optional int64  favourite_number = 2;
    repeated string interests        = 3;
}
```

# Java codegen with protoc

- protoc generates Java code
  - `Person` class
  - Code to build the binary representation (serialization)
  - Parse a `Person` object from the binary representation (deserialization)
- Classes can be used by both the server and client

```
// PSEUDOCODE (protobuf version 2)

public final class Person {
  // -- snip (fields)
  // -- snip (getters and setters)

  // Serialization
  public byte[] toByteArray() { ... }
  public void writeTo(OutputStream out) { ... }
  public int getSerializedSize() { ...}

  // --- Deserialization (Des) ---
  public Person parseFrom(byte[] data) { ... }
  public Person parseFrom(InputStream in) { ... }

}
```

# Generating gRPC endpoints

- Define the interface for service endpoints in `.proto` file
- The method name (`SayHello`) is the endpoint
  - Note: no URLs – gRPC is not HTTP
- Run `protoc` with `--grpc-java_out=gen`

```
// myservice.proto
syntax = "proto3";

service Greeter {
  rpc SayHello(HelloRequest) returns (HelloReply);
}

message HelloRequest { string name = 1; }
message HelloReply   { string msg  = 1; }
```

# Generating gRPC endpoints

- Define the service endpoints in `.proto` file
- Run `protoc` with `--grpc-java_out=gen`

```
// Server code generated by protoc

public final class GreeterGrpc {
  static final String SERVICE_NAME = "demo.Greeter";

  public abstract static class GreeterImplBase
    implements io.grpc.BindableService {

    public void sayHello(HelloRequest req,
        io.grpc.stub.StreamObserver<HelloReply>
resp) {
        resp.onError(... EXCEPTION ...);
    }
  }
}
```

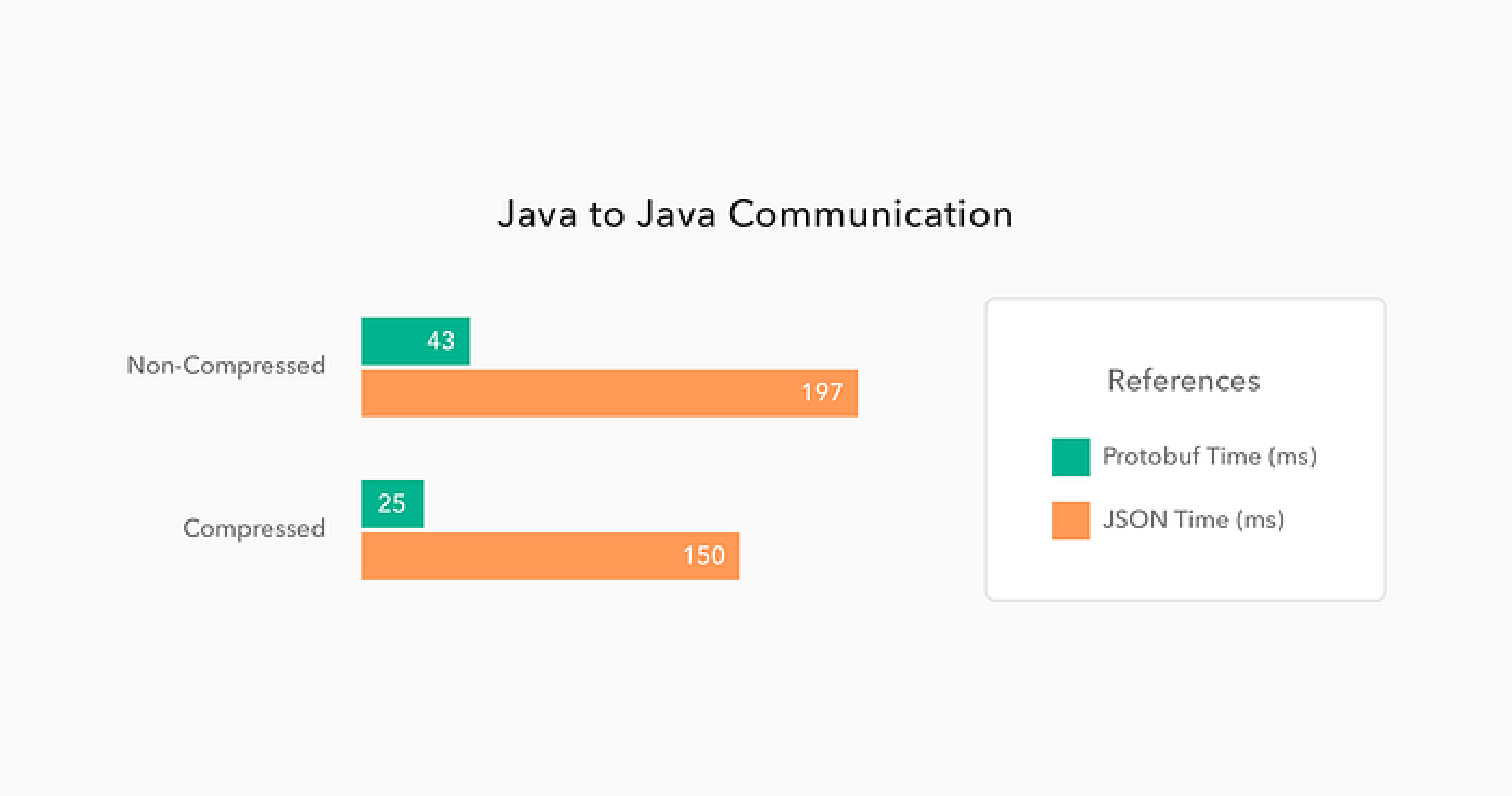
# Generating gRPC endpoints

- Define the service endpoints in `.proto` file
- Run `protoc` with `--grpc-java_out=gen`
- Extend `GreeterImplBase` to provide functionality

```
public final class GreeterImpl extends
GreeterGrpc.GreeterImplBase {

  @Override
  public void sayHello(
    HelloRequest req,
    io.grpc.stub.StreamObserver<HelloReply> resp) {
        resp.onNext(
        HelloReply.newBuilder().setMsg("hi " +
                req.getName()).build());
        resp.onCompleted();
  }
}
```

# gRPC efficiency

https://itnext.io/a-minimalist-guide-to-grpc-e4d556293422

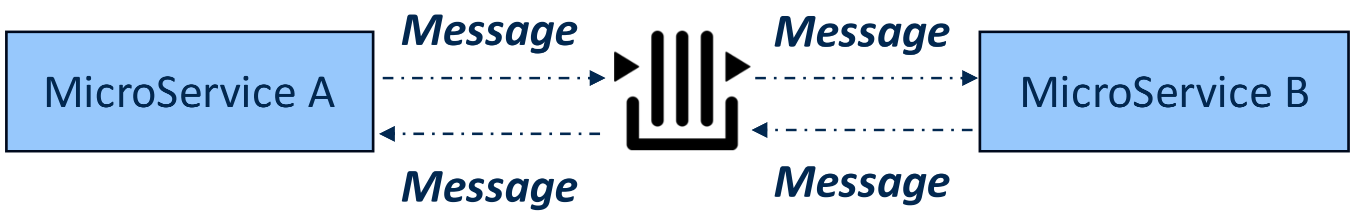


# gRPC implications

- Protocol efficiency: high
- Type safety (schema adherence): high
  - If a field is missing, the ser-des logic will detect it and raise an Exception
- What about ease of debugging?
  - Ease of debugging: low

# Message queuing (MQ)

- Asynchronous communication model
  - Messages sent to a queue and processed by consumers independently of the producer
  - Messages are saved on disk until the consumer acks consumption
- Stronger decoupling

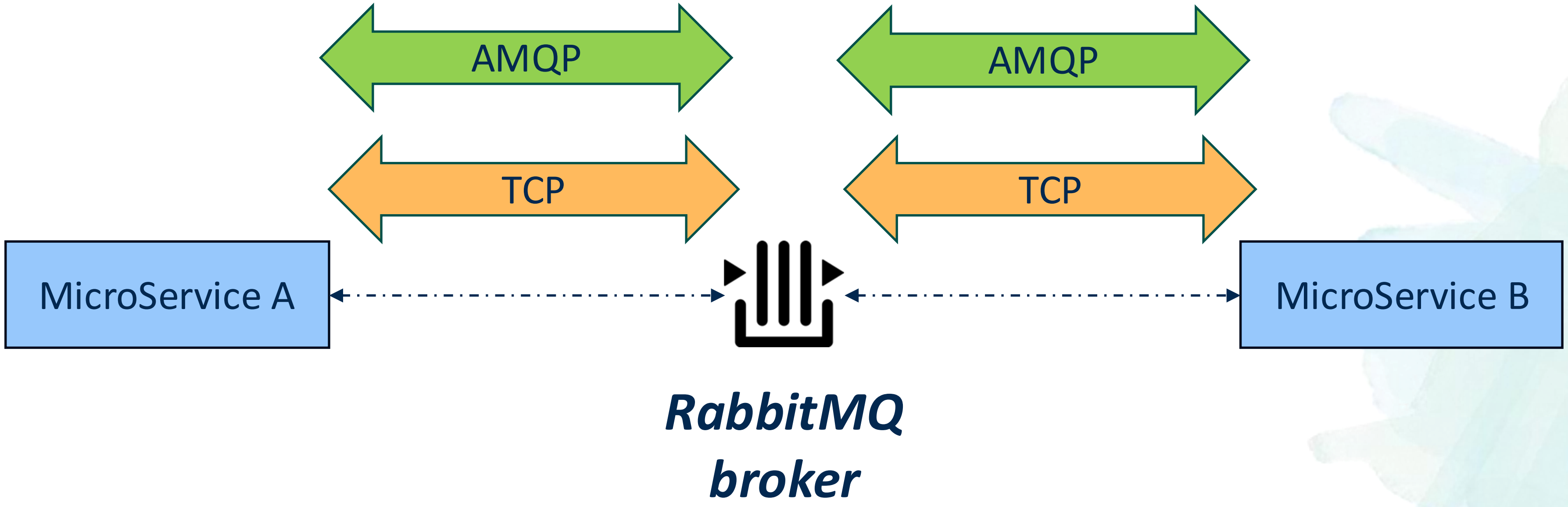


# Message queuing (MQ)

- Application must be aware of asynchronicity
- Message sent not necessarily message delivered
- Publisher can request acks
  - When the message exchange has received the message and saved it durably

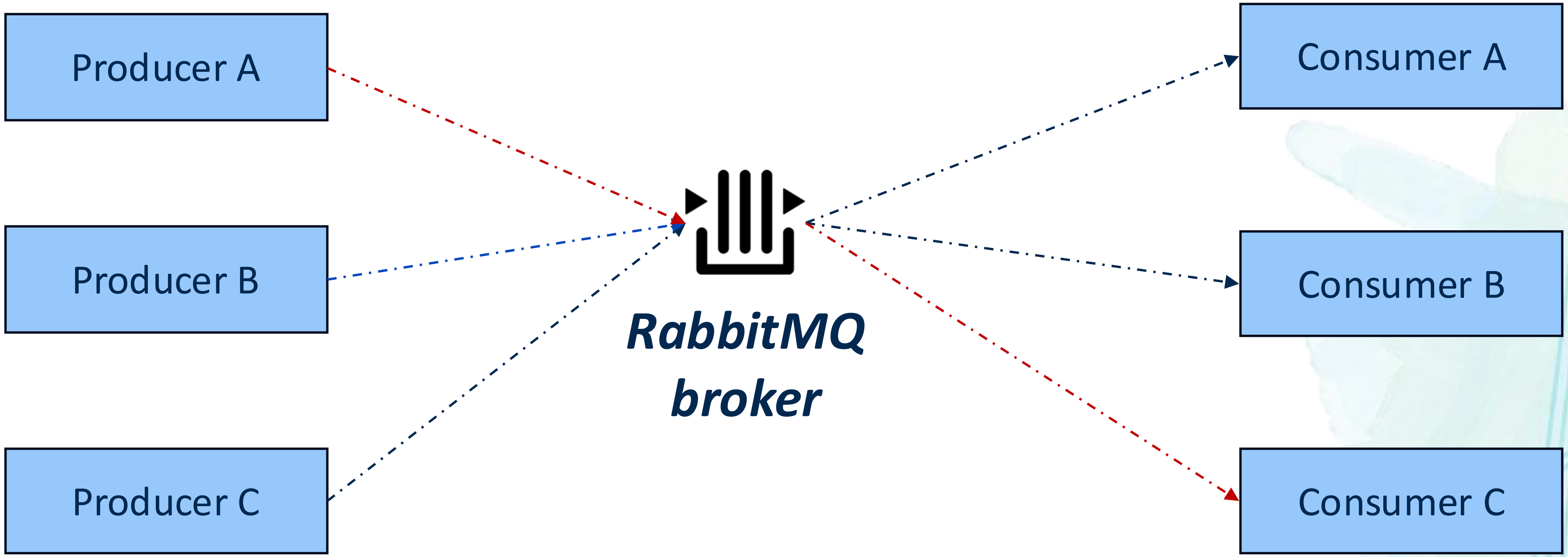
# RabbitMQ

- Advanced Message Queuing Protocol (AMQP)
  - Binary protocol running on TCP



# AMQP model

- The broker receives messages from publishers and route them to subscribers
- Each route is typically an on-disk queue

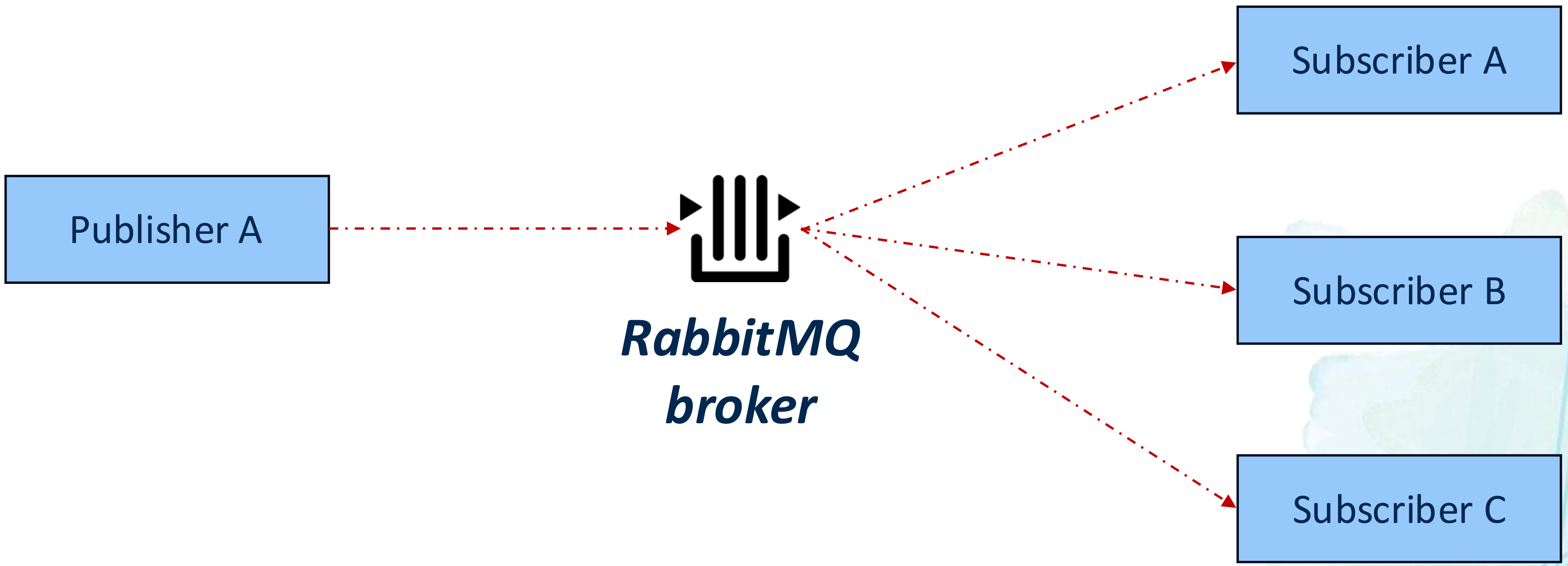


# Pub-sub architecture

- Asynchronous messaging pattern where **publishers** send messages to a central **message broker** or **topic**, and **subscribers** receive messages based on their subscriptions
- Broadcasting: messages can be sent to multiple subscribers
- Same frameworks often can act as both MQ or Pub-Sub depending on configuration

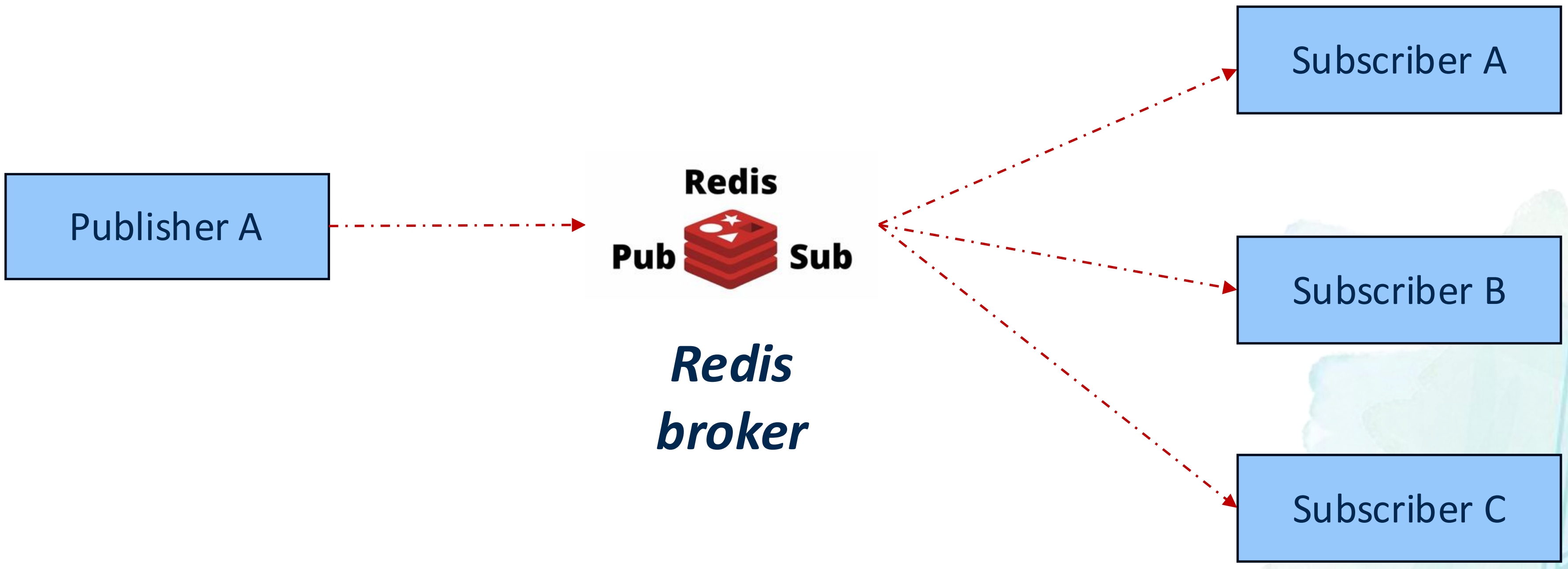
# RabbitMQ as message broker

RabbitMQ can bind same queue to multiple subscribers



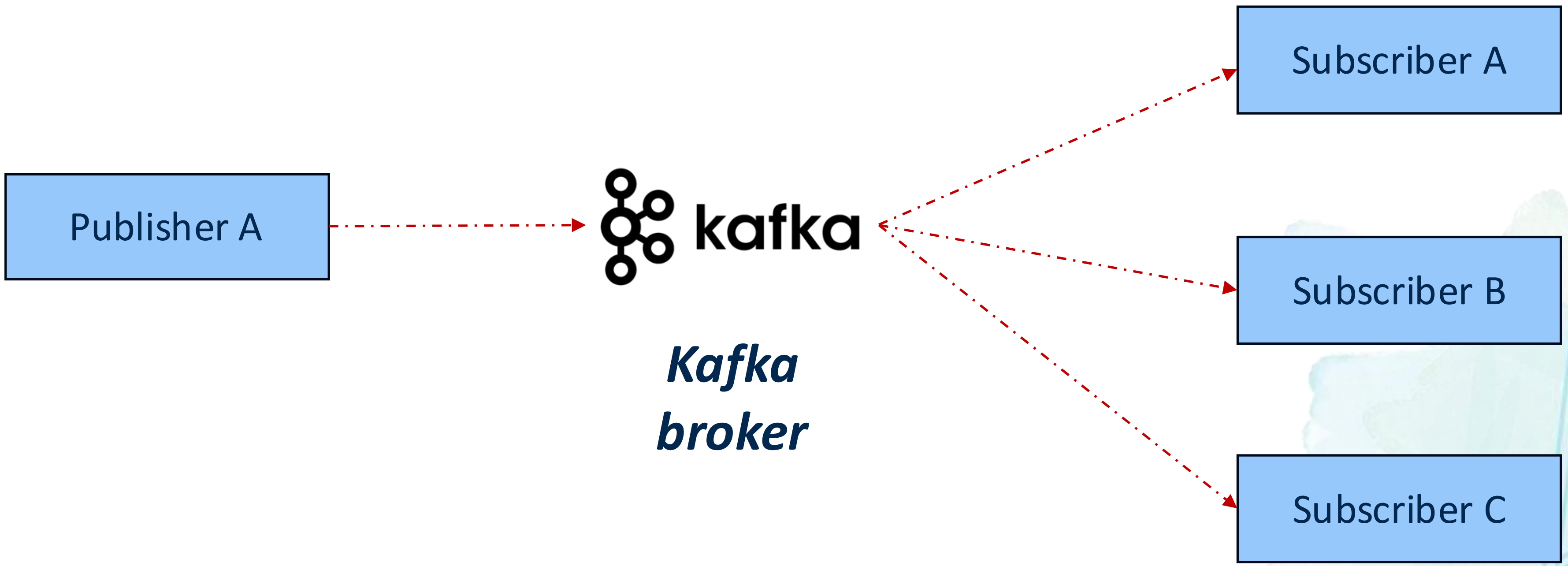
# Redis as message broker

Redis exposes “channel” that multiple subscribers can bind to



# Kafka as message broker

[More in next module]



# Message queues implications

- Strong decoupling
  - Consumer does not have to be running
  - Message persisted in the message queue
- Potentially increased latency
  - Increased number of hops
  - Overhead for persisting the message

# Summary

- Microservices improve throughput
- Microservices allow each service to have its own database
  - Data models – relational, document model, key-value store
  - Storage engine – log-based and B-tree based
- Microservices can communicate
  - Synchronously (JSON over HTTP, gRPC)
  - Asynchronously (Message queues)
  - Event-driven pub/sub (Kafka – next module)