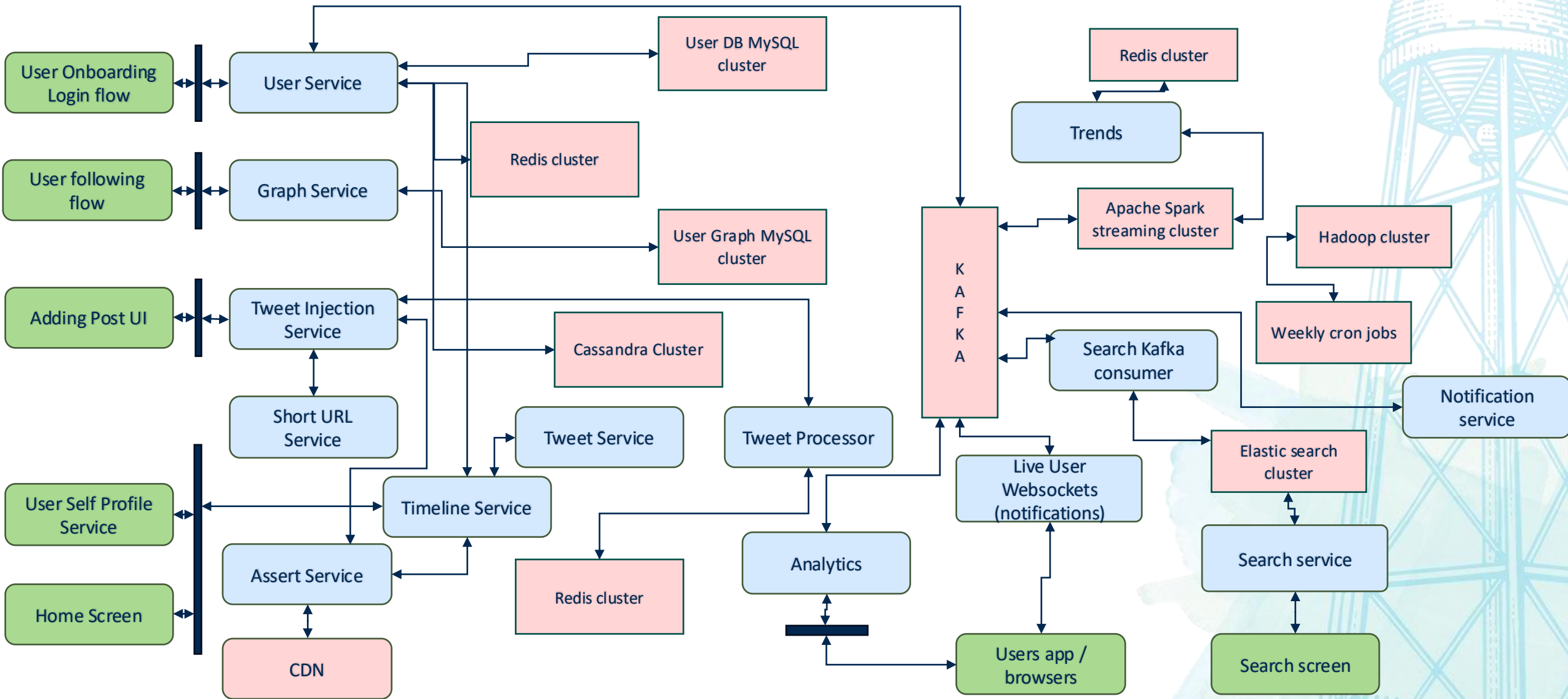


# Modern software architectures

Tapti Palit



# Modern software architecture



# 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?

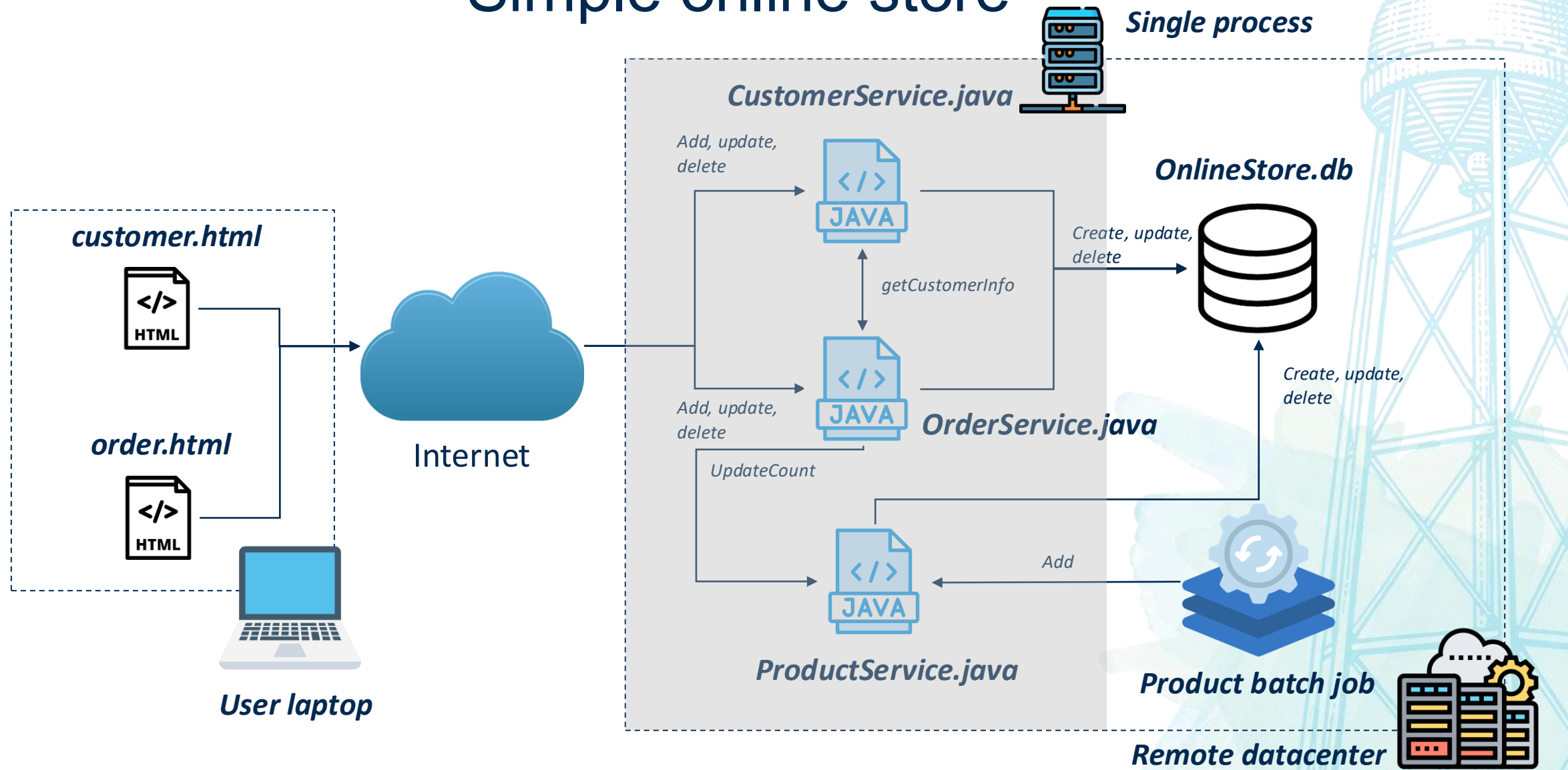


# Monolithic software architecture

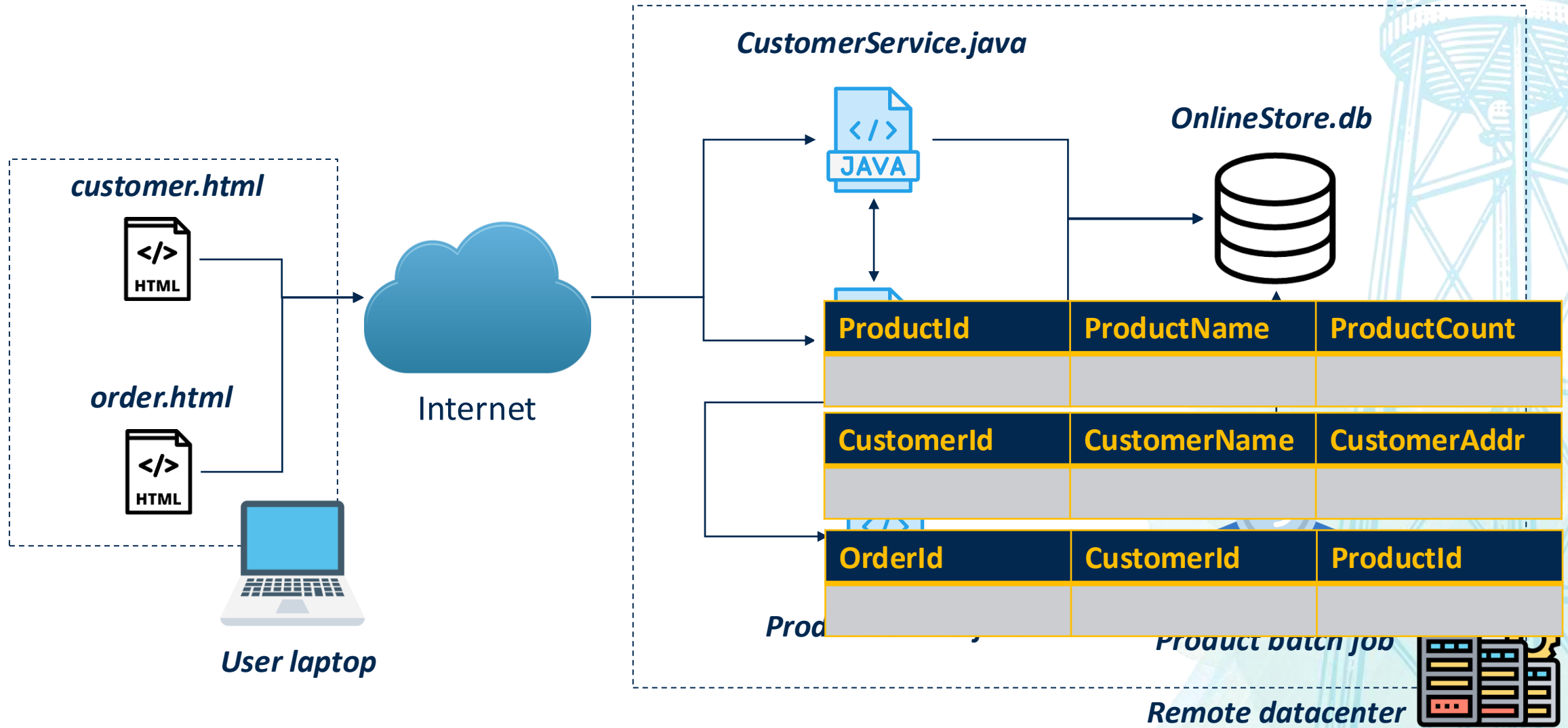
- Widely used in early, mid 2000s
- All software components are part of the same application
- All software components run on the same machine, in same process
- Typically developed in the same language stack



# Simple online store



# Simple online store



# 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*

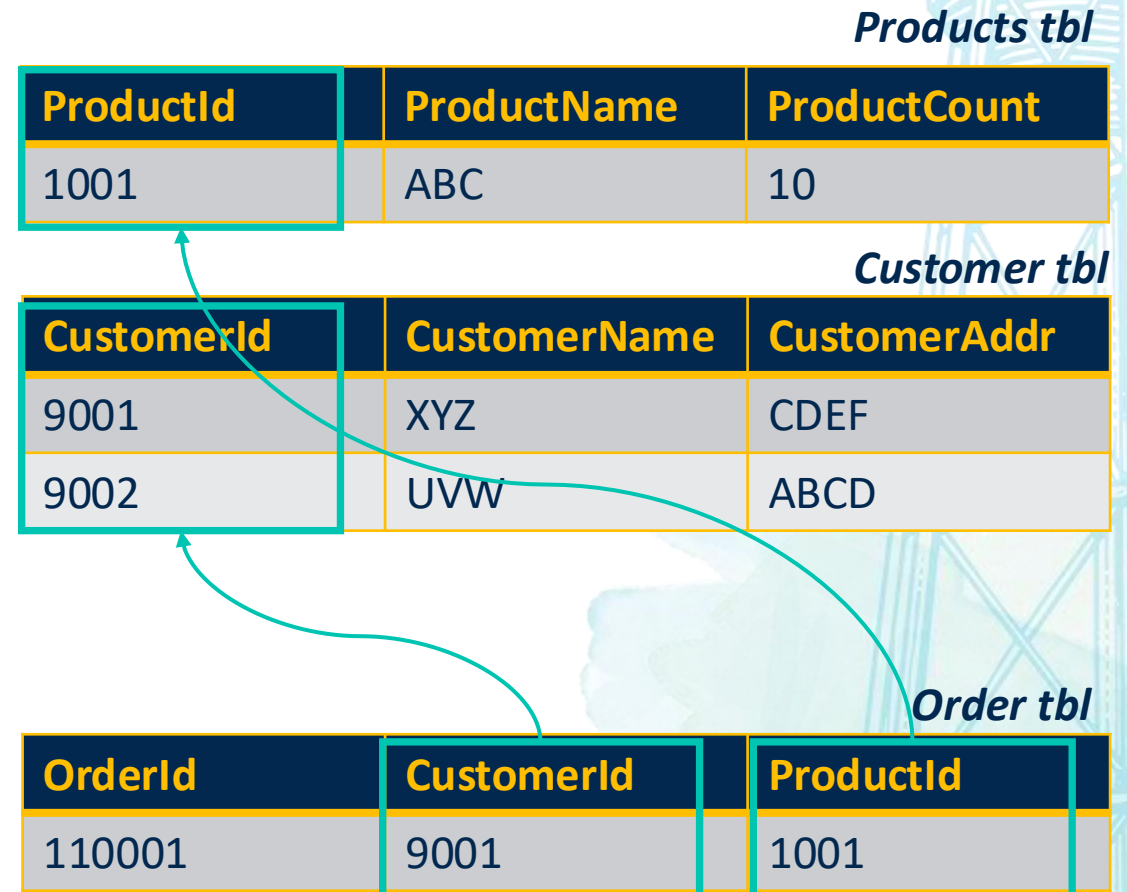
CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF

*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

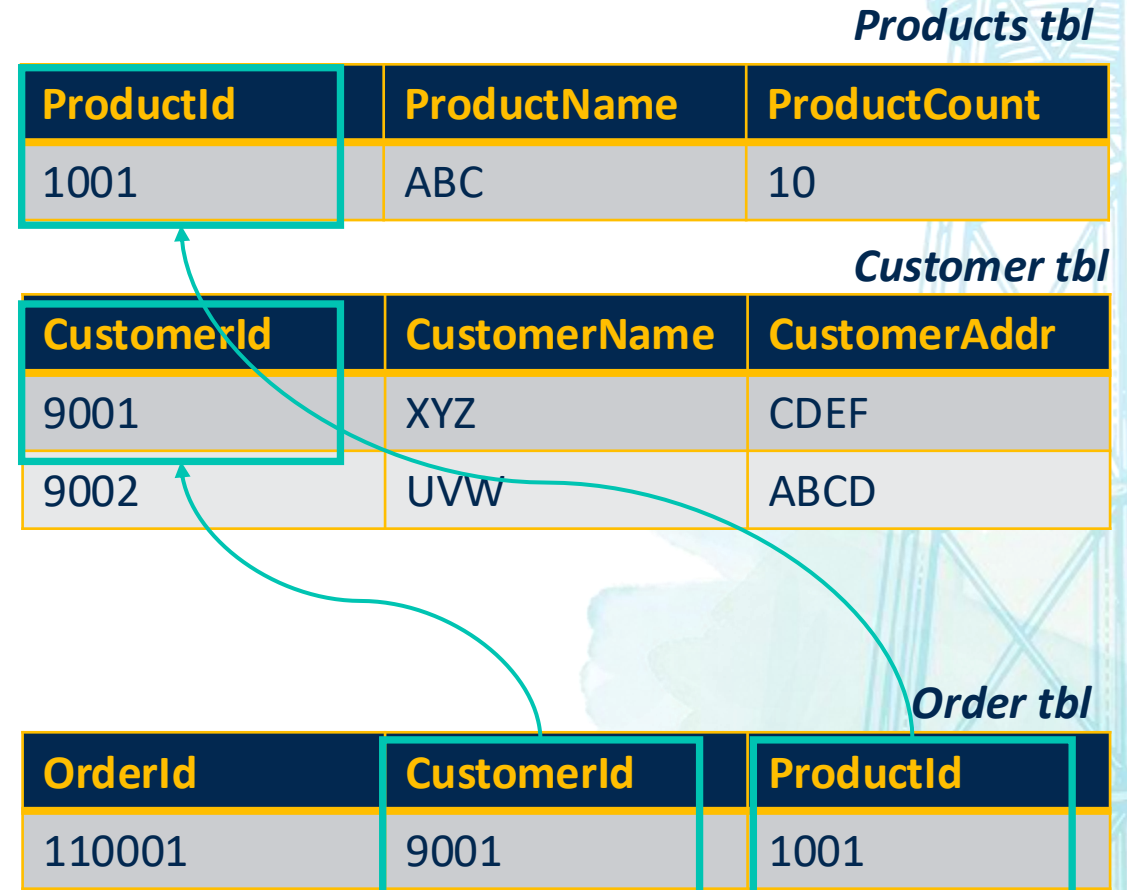


# Structured query language (SQL)

- 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



# SQL joins

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*

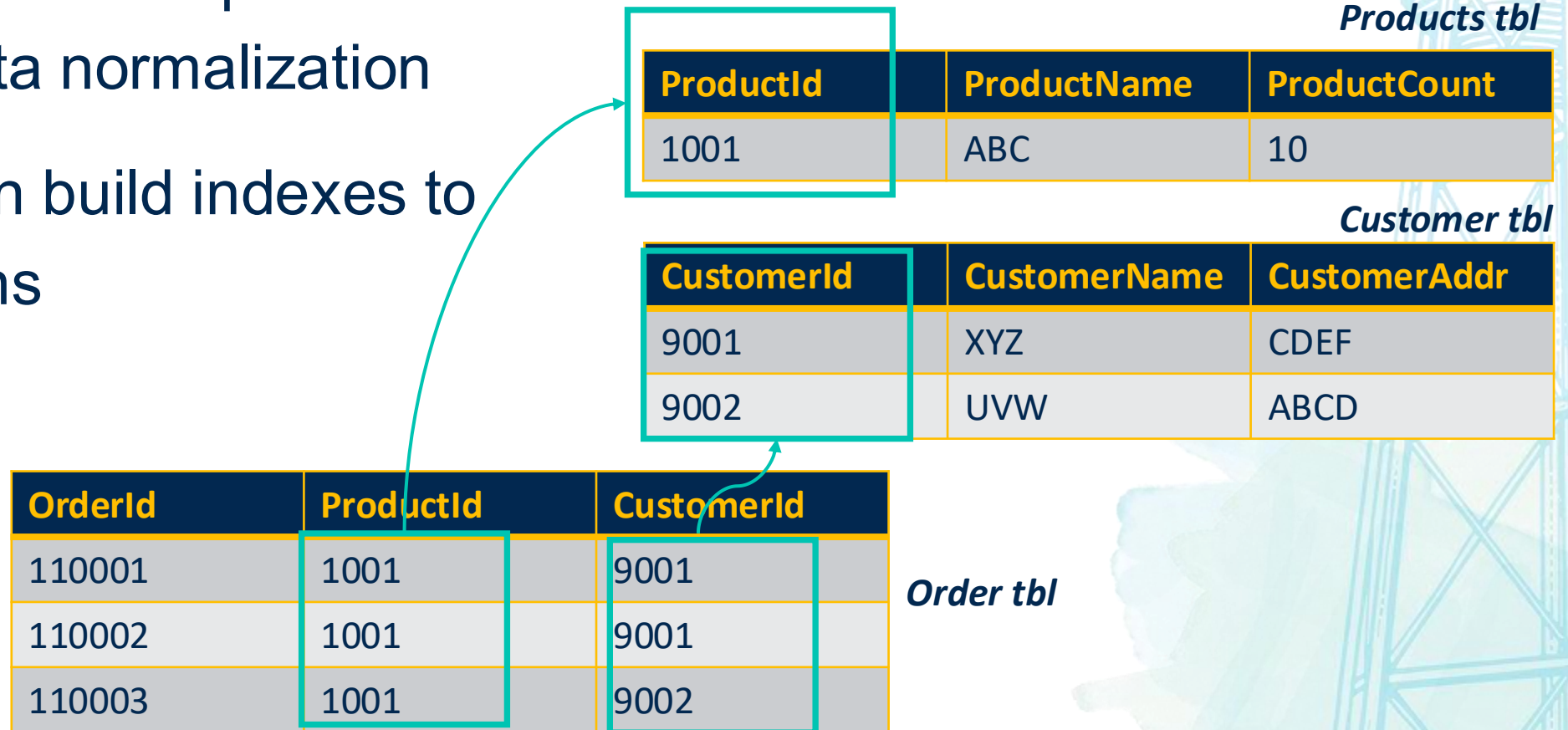
CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	ABCD

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



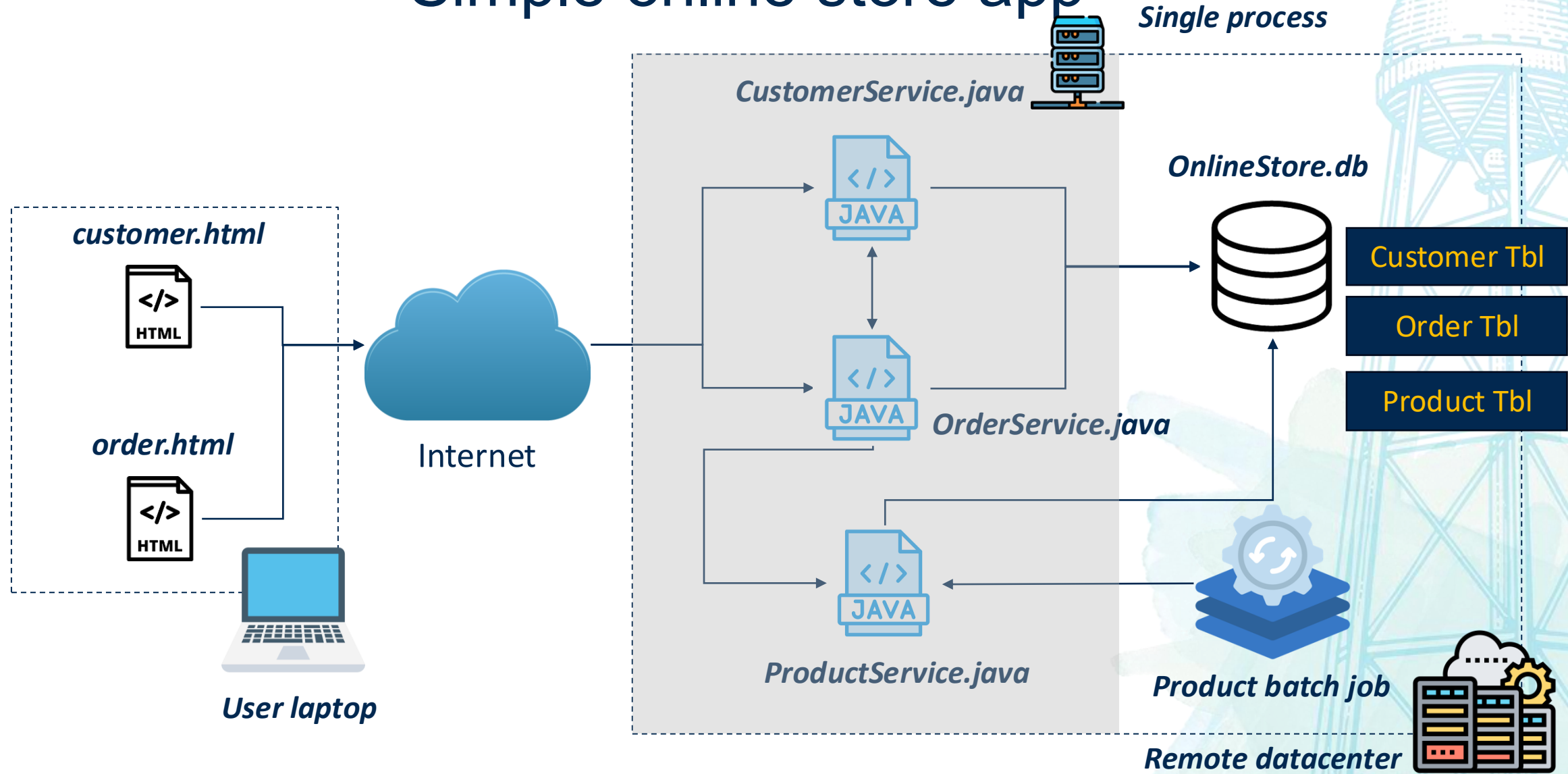
# Relational database implications

- 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

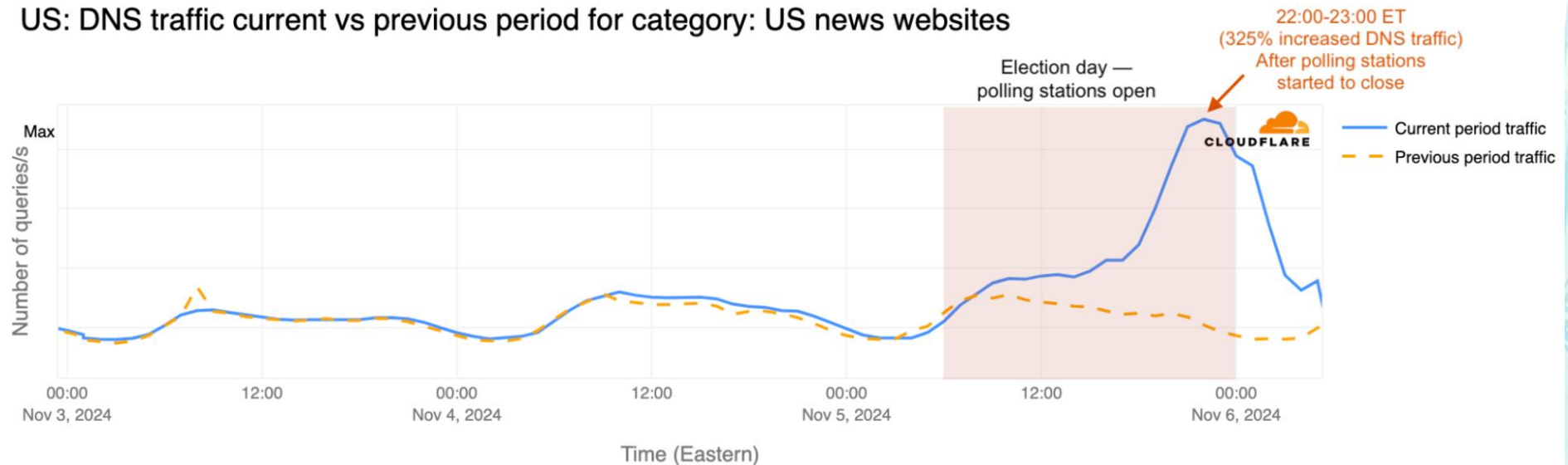


# Scalability

- 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/>

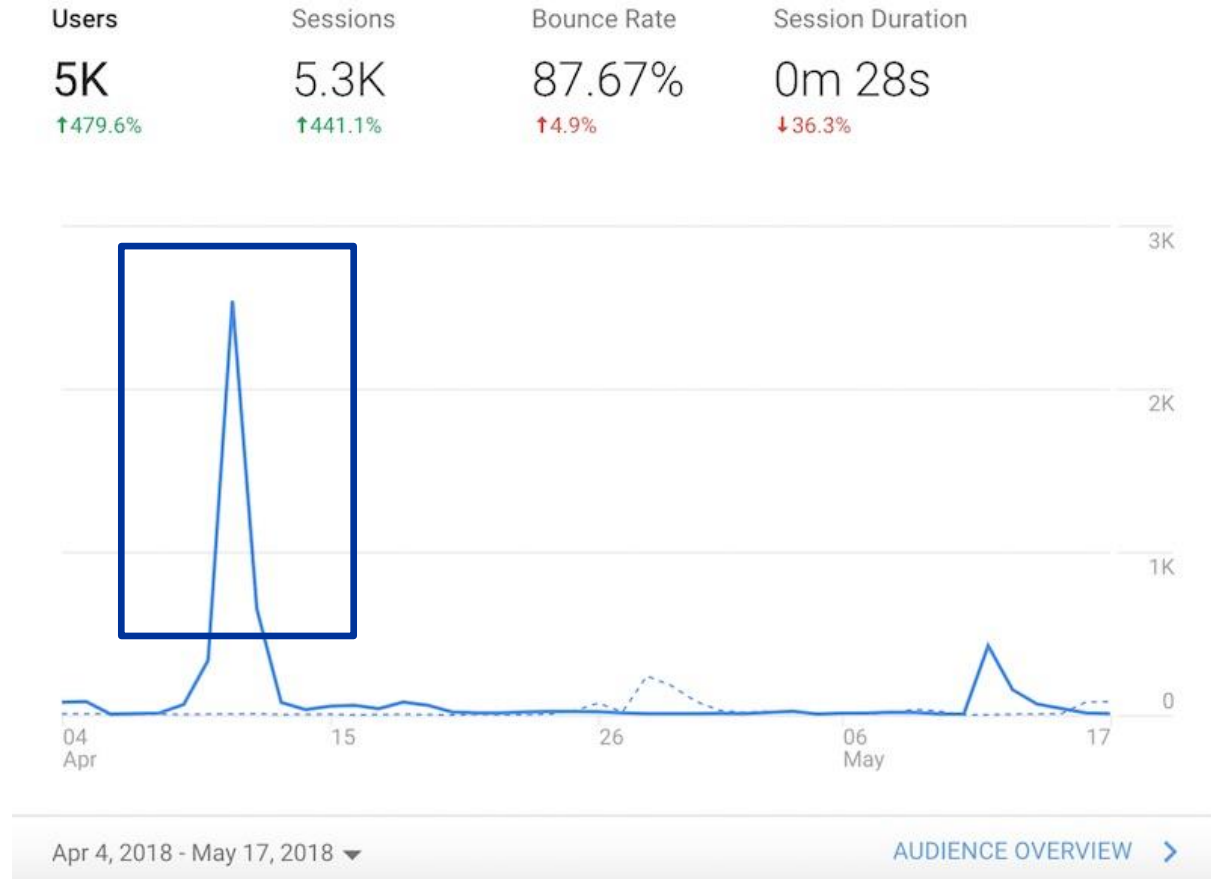
US: DNS traffic current vs previous period for category: US news websites



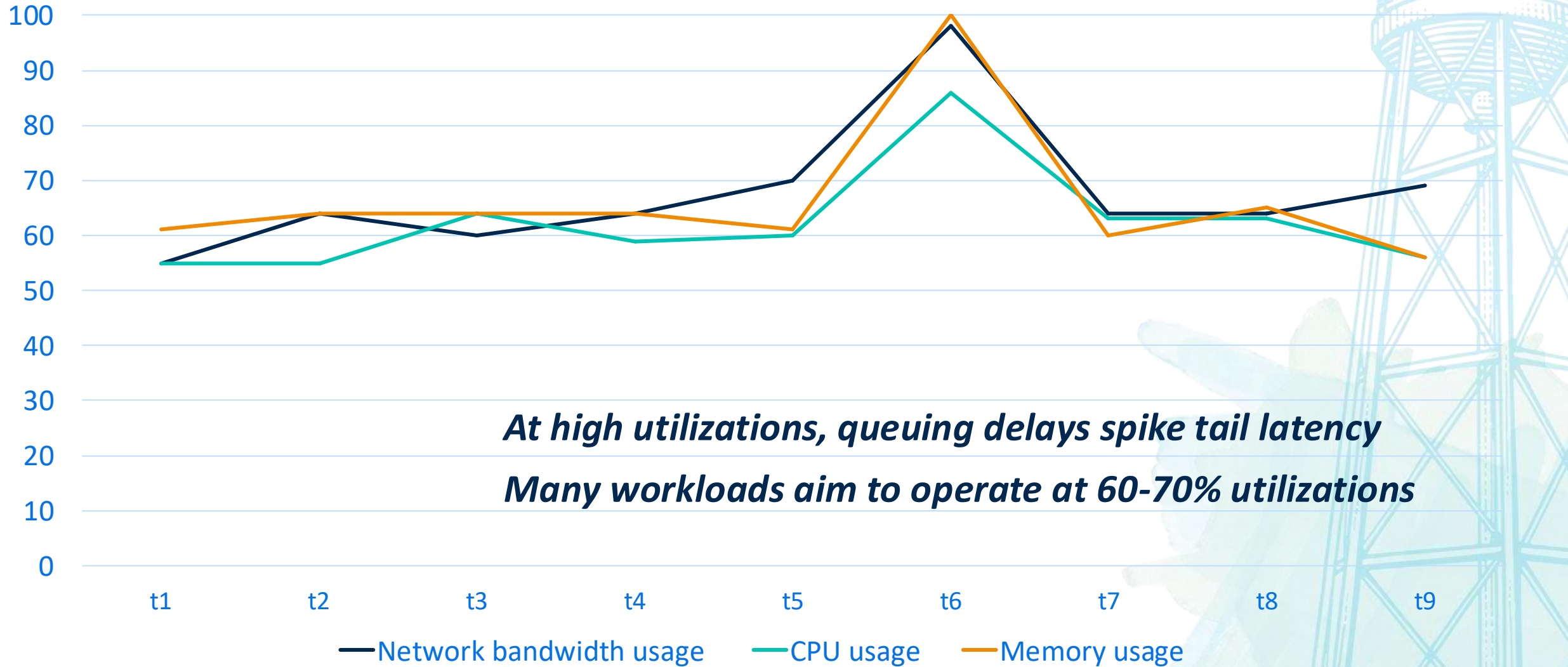
# Scalability

- 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/>

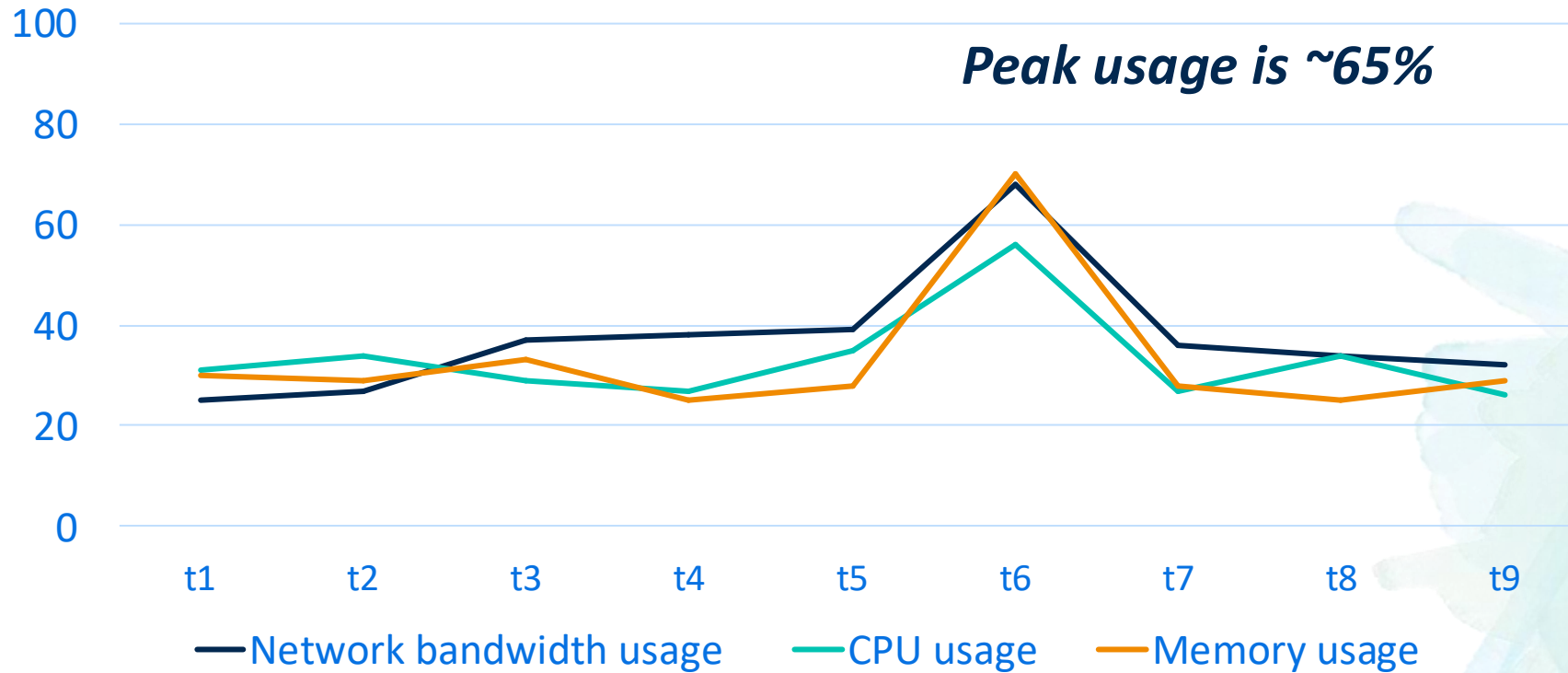


# Traffic spike to online store



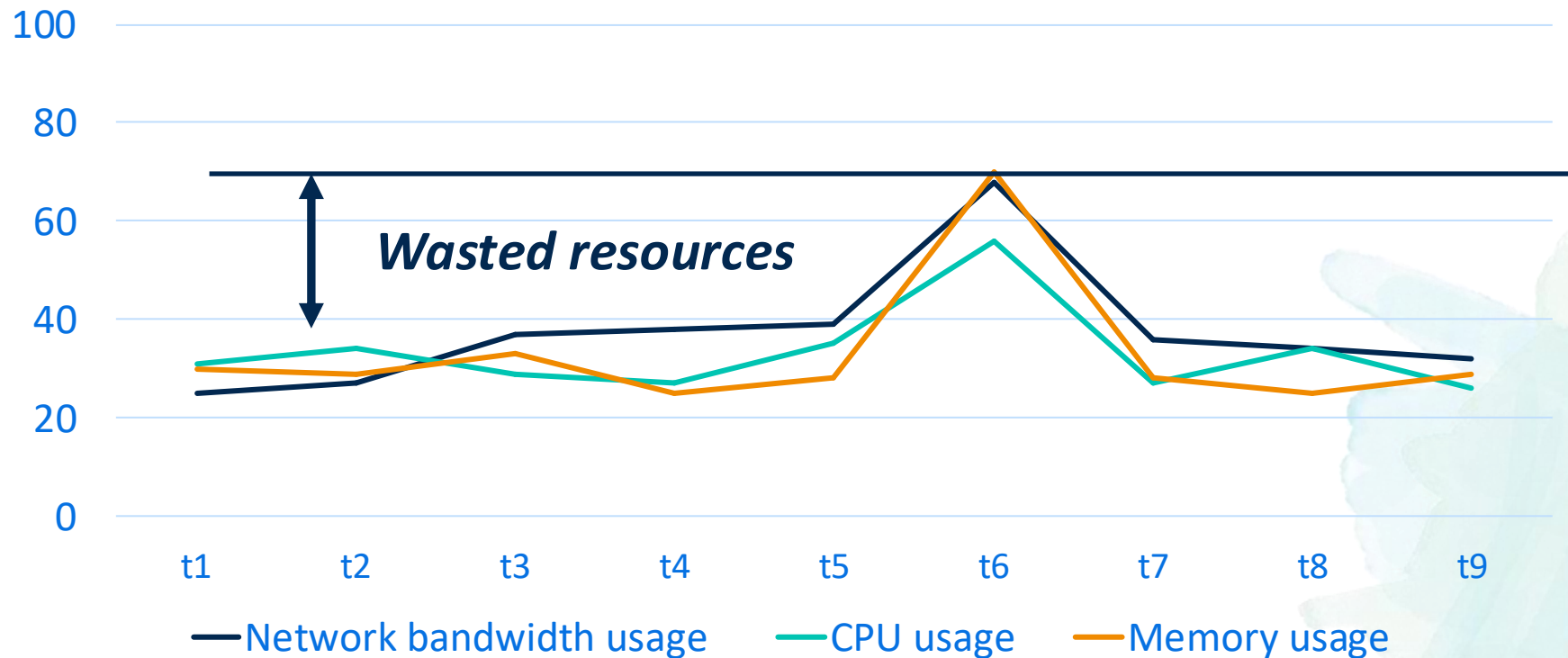
# Vertical scaling

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



# Vertical scaling limitations

Resource wastage when requests are not bursty

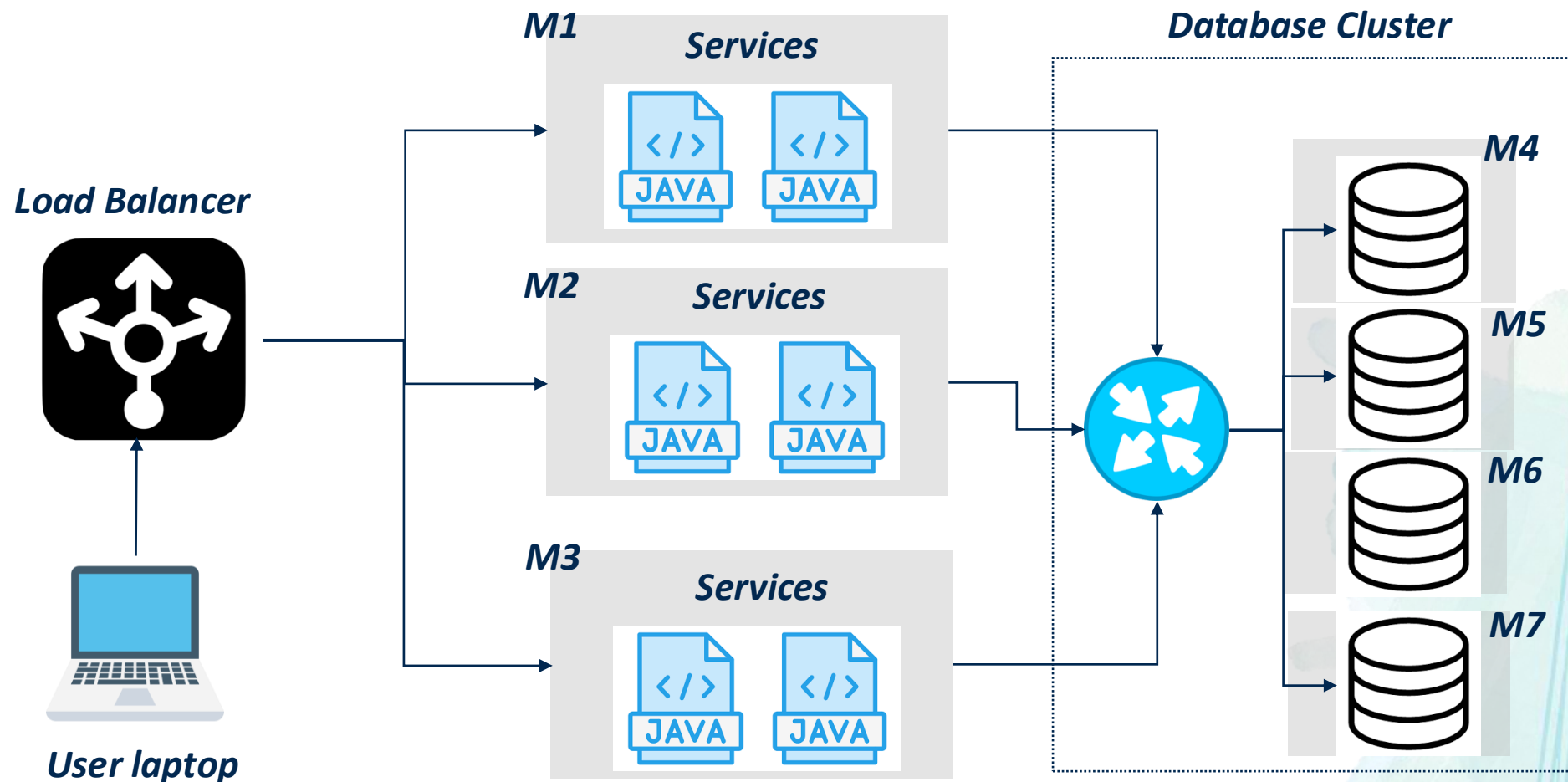


# Vertical scaling limitations

- 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

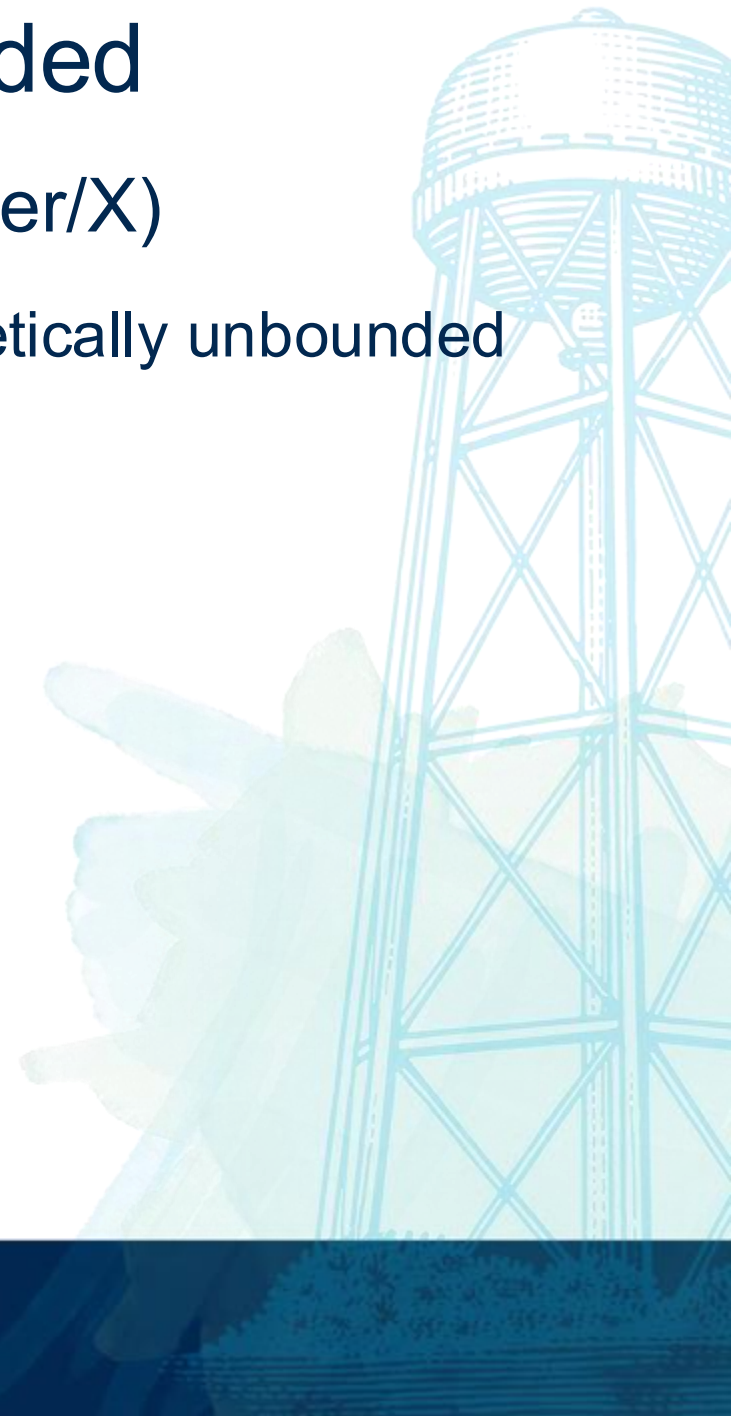


# Load balancer

- 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

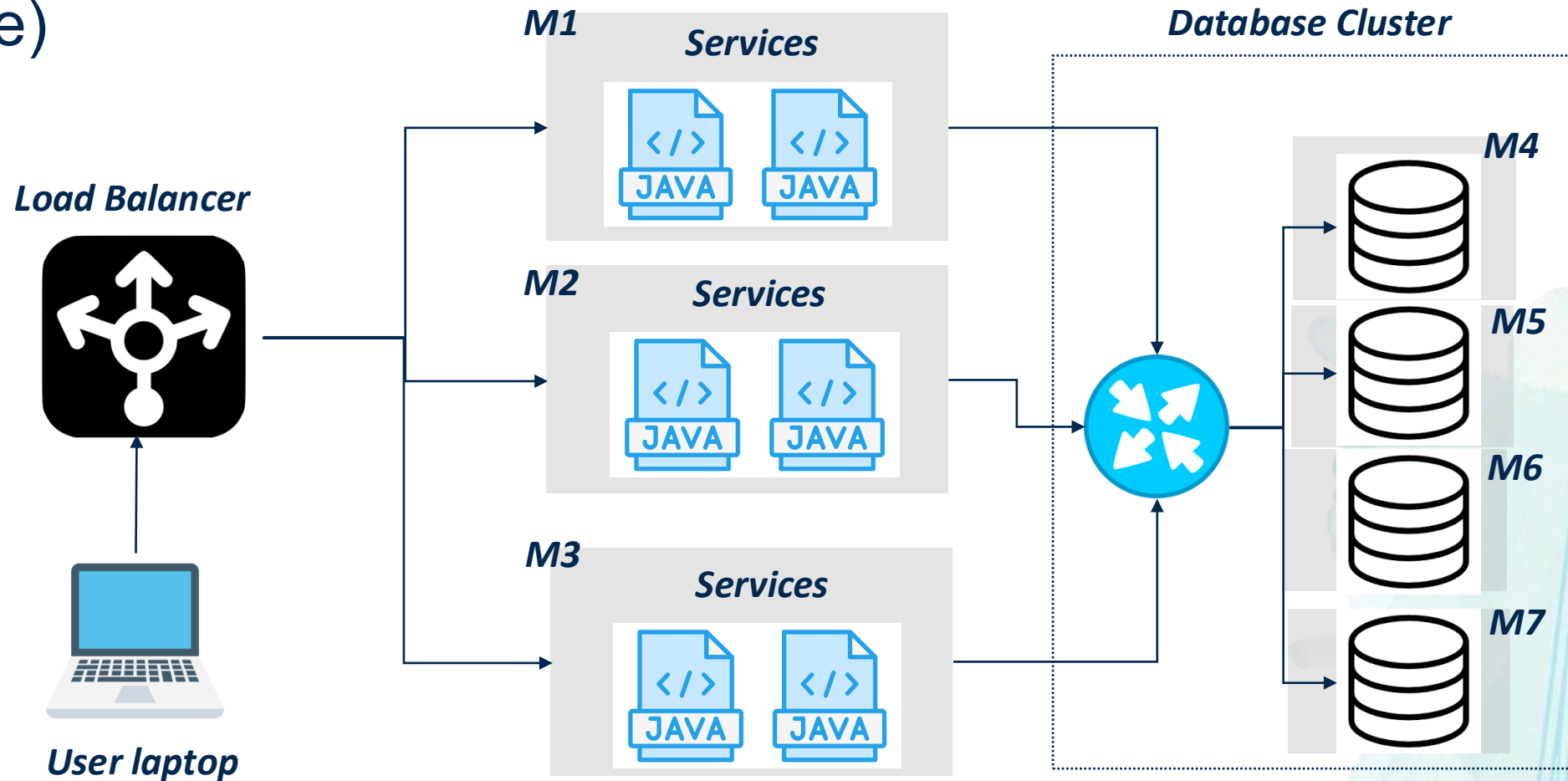
# Horizontal scaling is unbounded

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



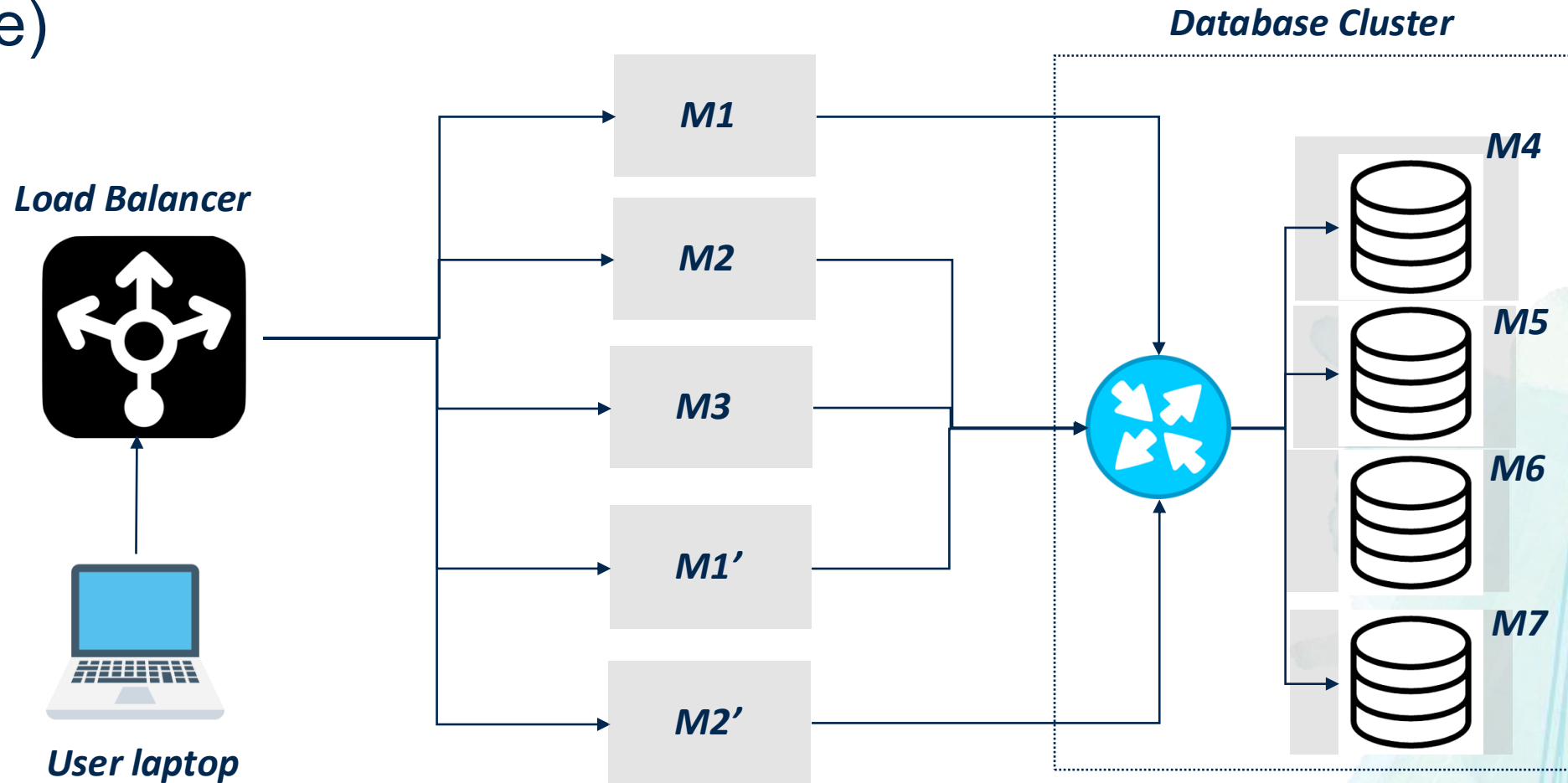
# Horizontal scaling is necessary for autoscaling

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



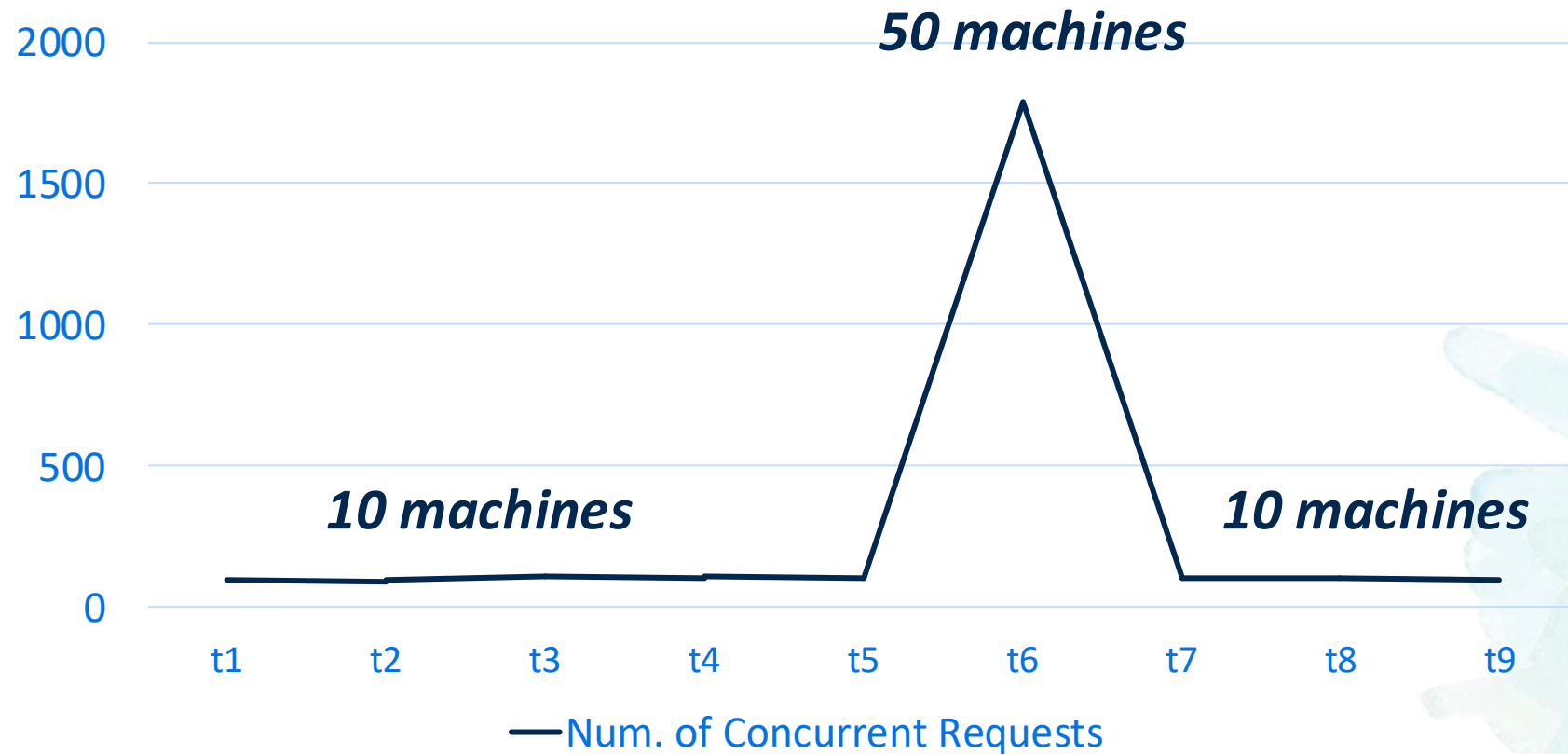
# Horizontal scaling is necessary for autoscaling

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

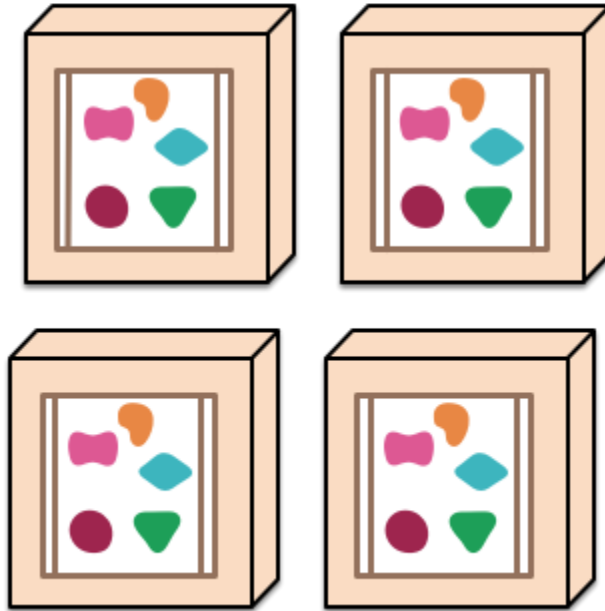
<https://martinfowler.com/articles/microservices.html>

# Microservices overview

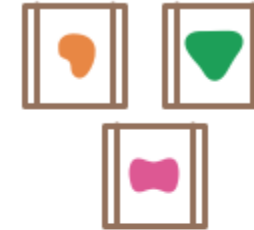
*A monolithic application puts all its functionality into a single process...*



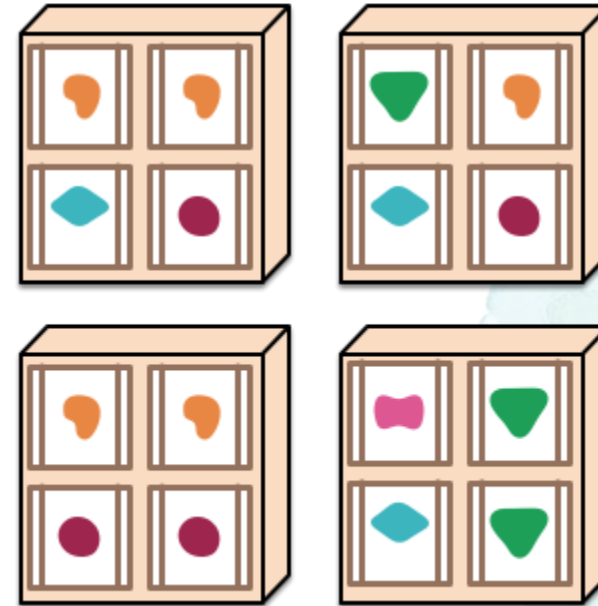
*... and scales by replicating the monolith on multiple servers*



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

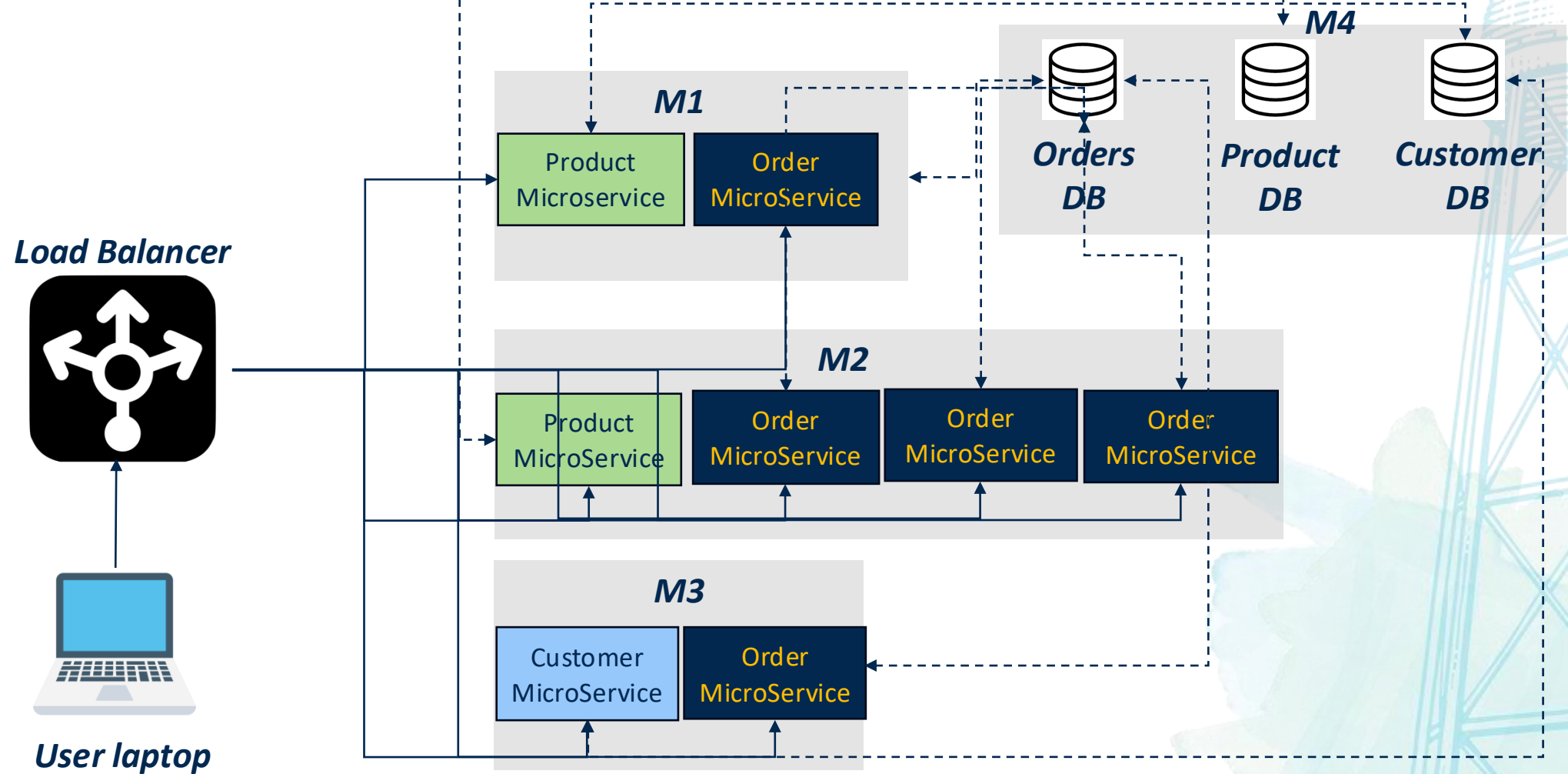


*... and scales by distributing these services across servers, replicating as needed.*

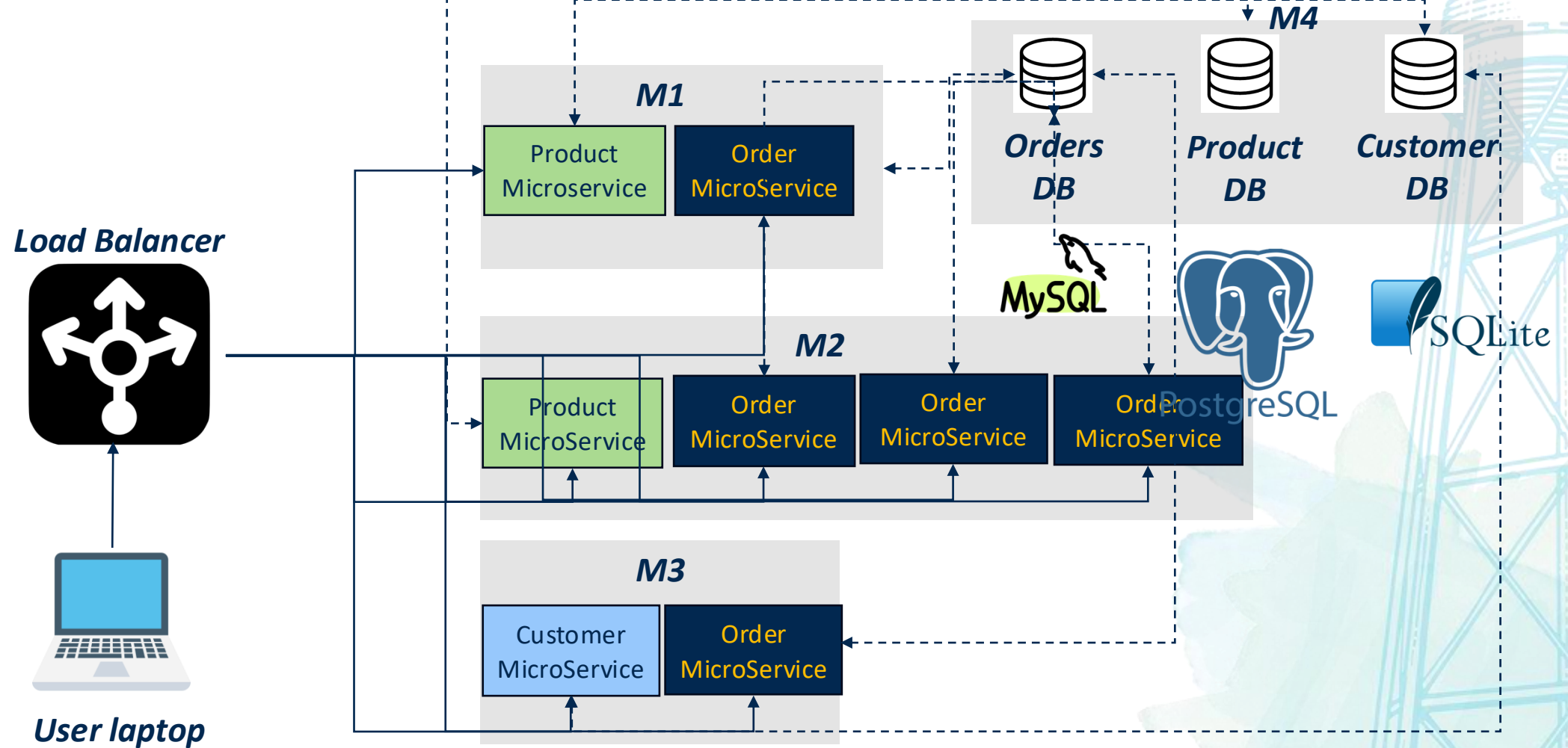


<https://martinfowler.com/articles/microservices.html>

# Microservice architecture for online store



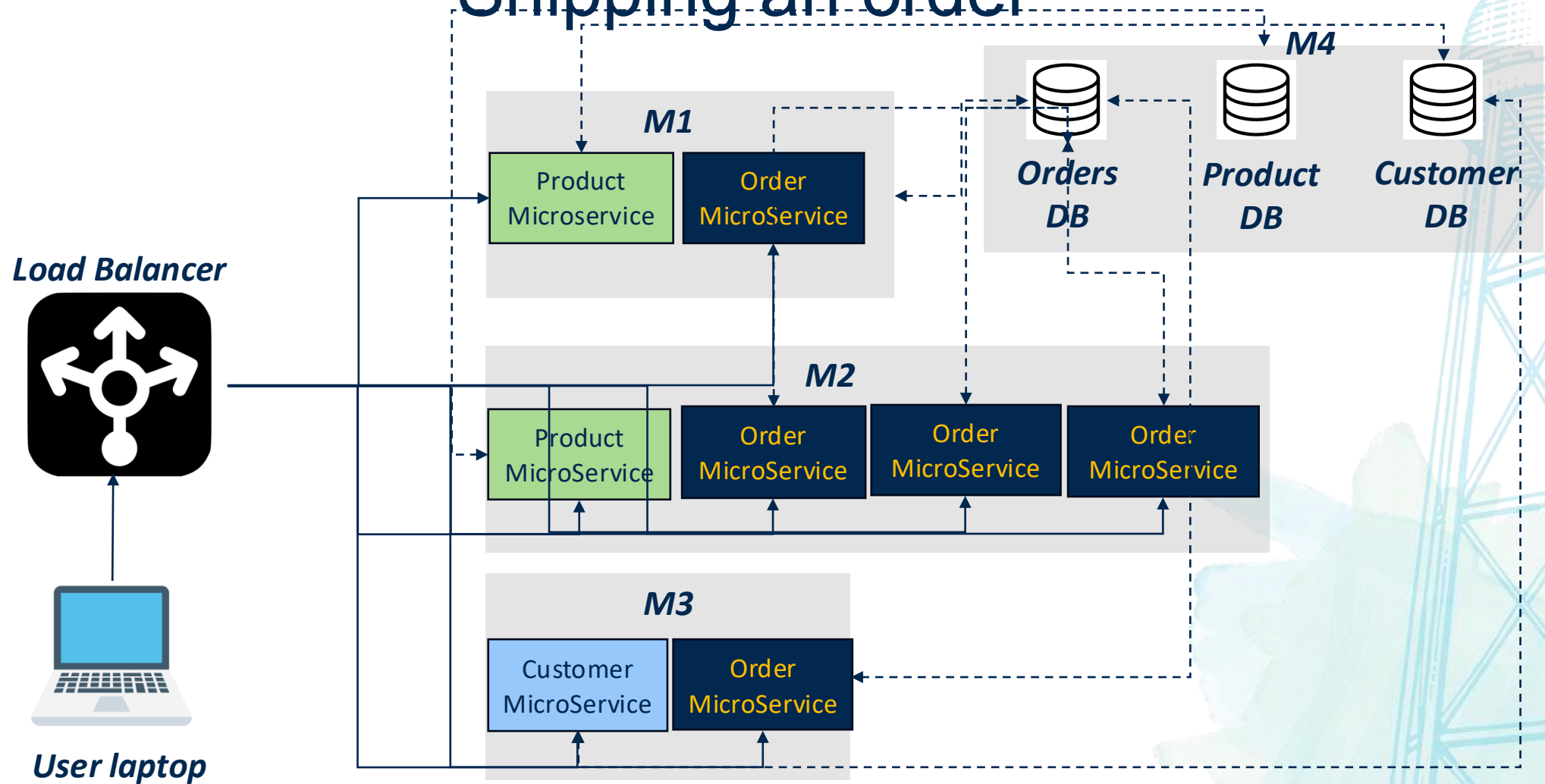
# Each microservice chooses its own database



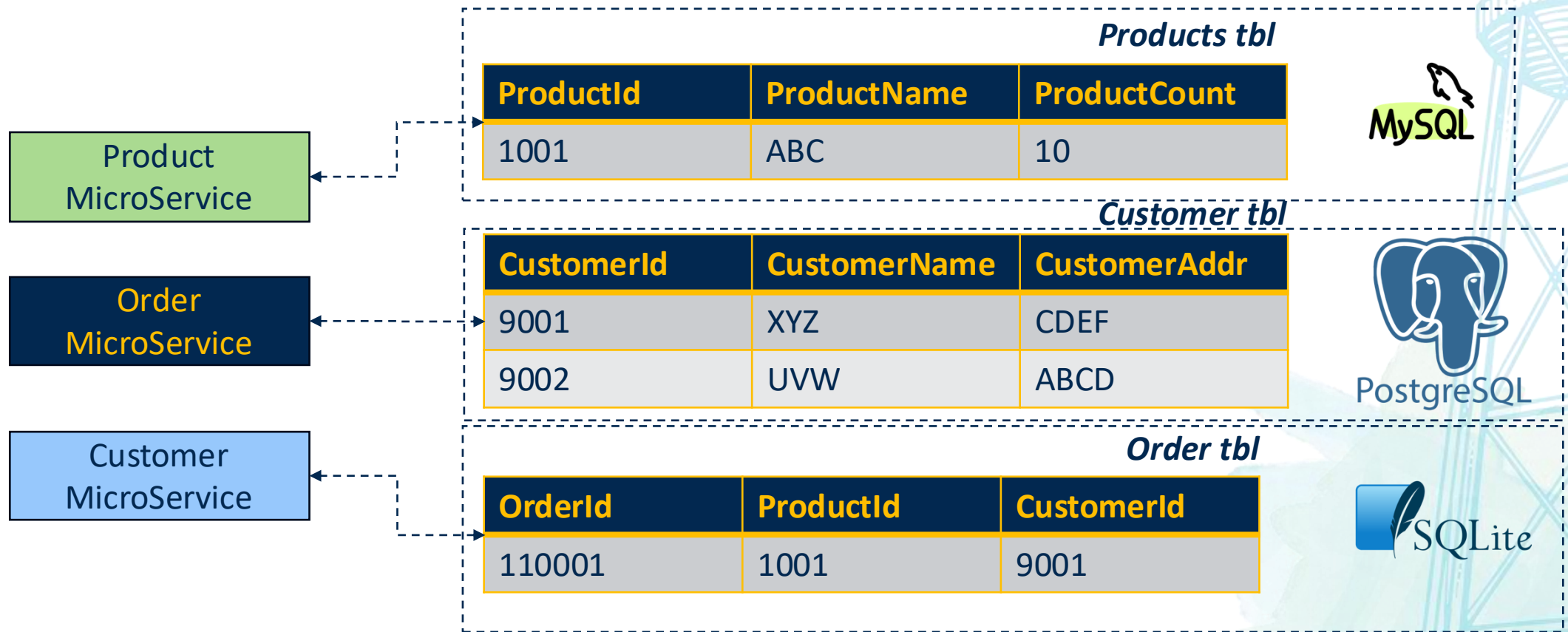
# What did we achieve?

- 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



***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?

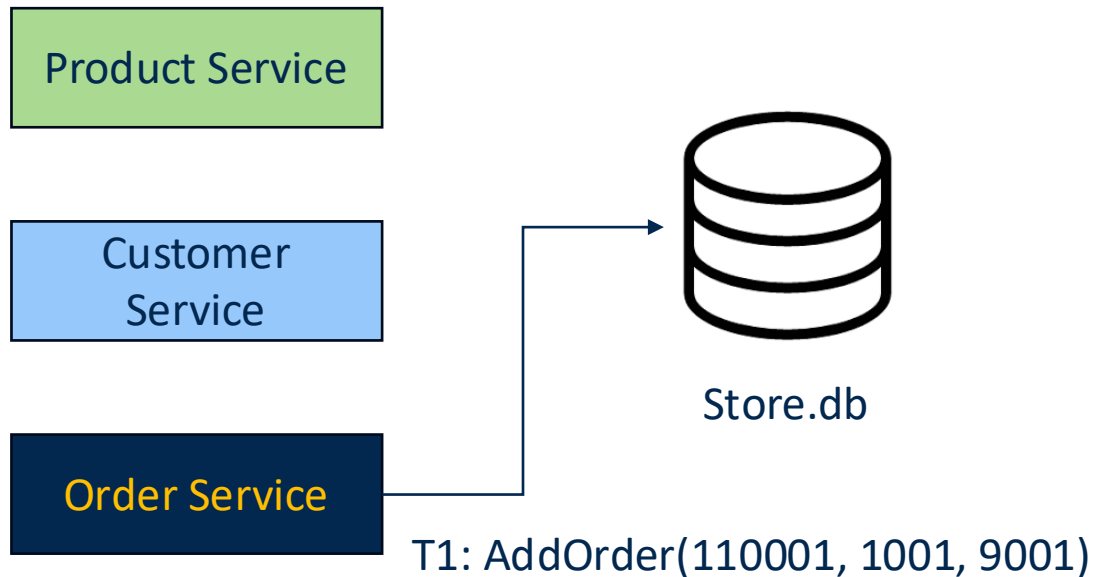
# Denormalization

- 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



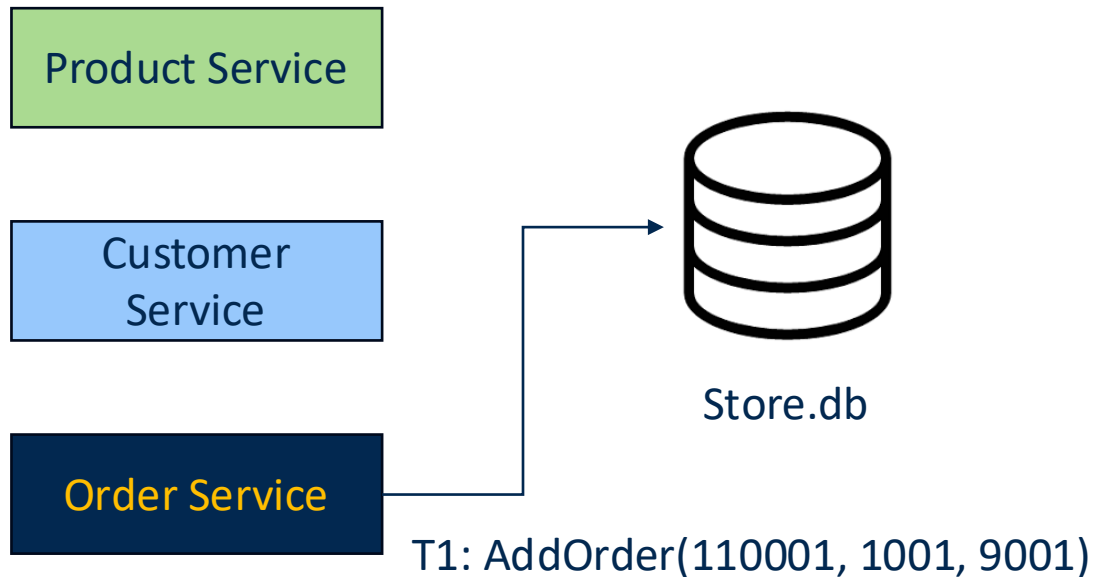
ProductId	ProductName	ProductCount
1001	ABC	10

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



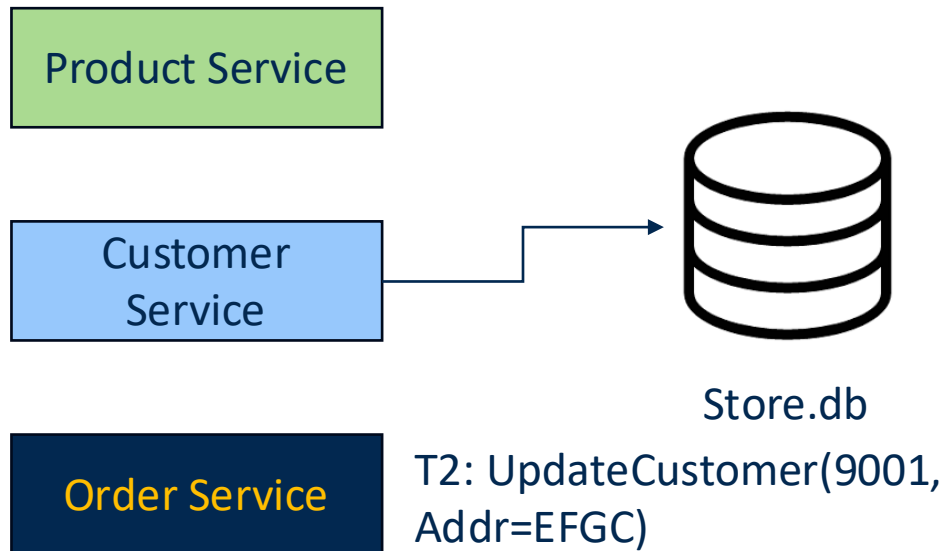
ProductId	ProductName	ProductCount
1001	ABC	10

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



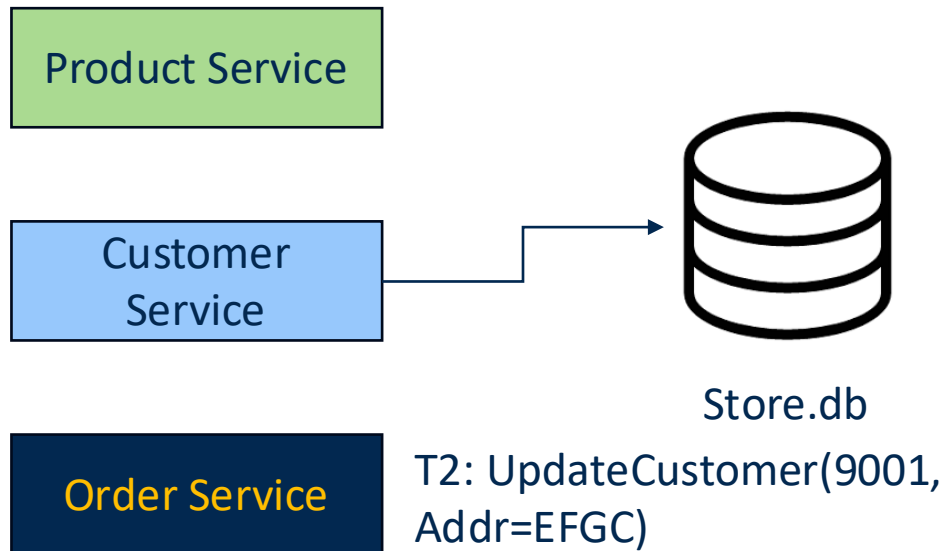
ProductId	ProductName	ProductCount
1001	ABC	10

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



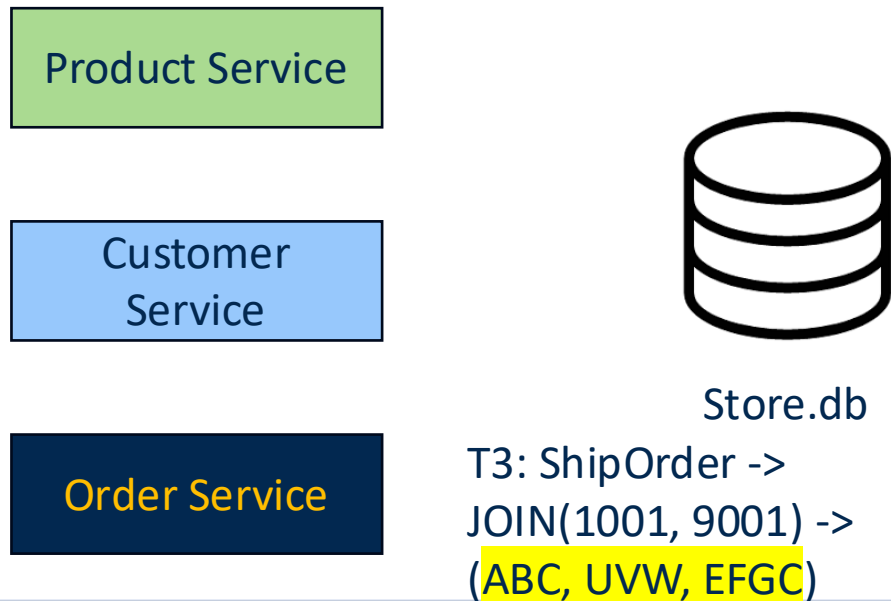
ProductId	ProductName	ProductCount
1001	ABC	10

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



ProductId	ProductName	ProductCount
1001	ABC	10

CustomerId	CustomerName	CustomerAddr
9001	XYZ	CDEF
9002	UVW	EFGC

OrderId	ProductId	CustomerId
110001	1001	9001

# Microservice consistency guarantees

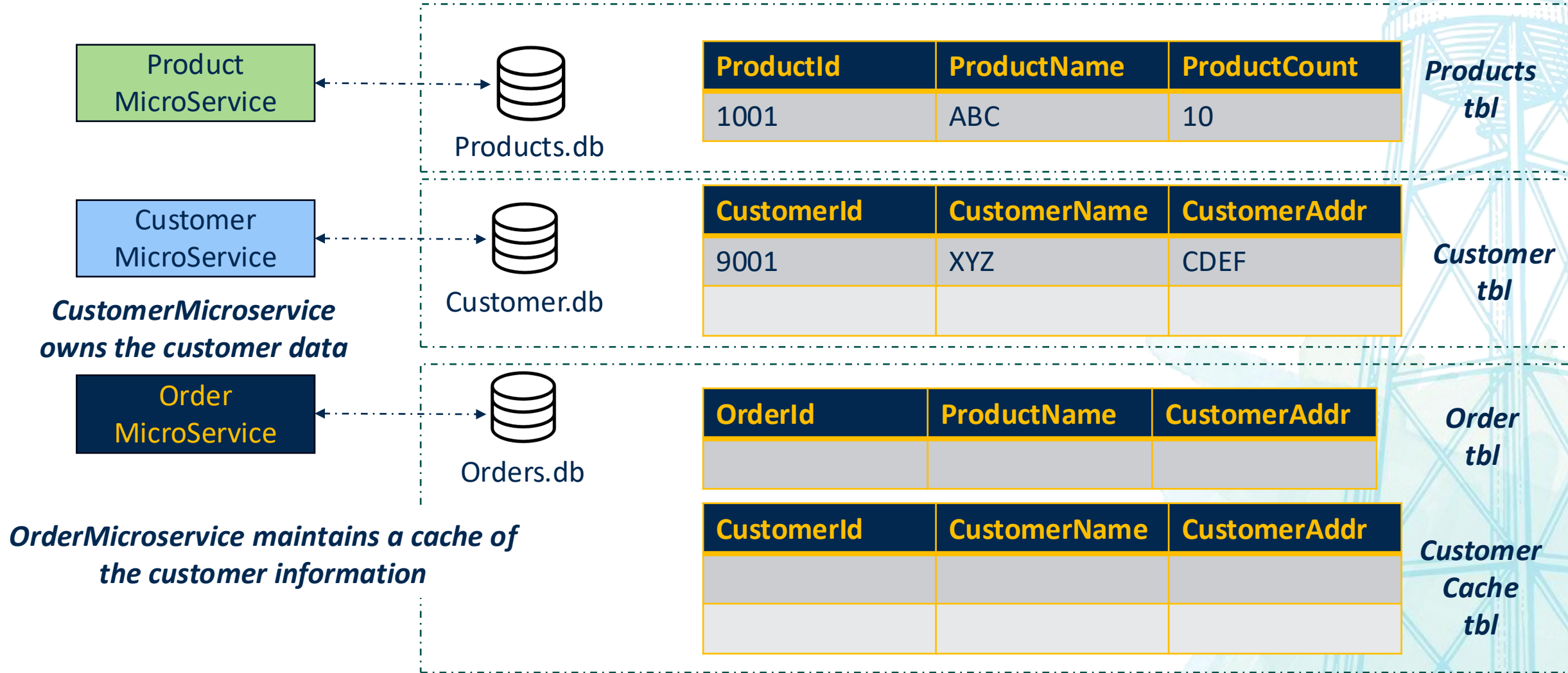
- Core idea: a microservice ***owns*** some data
  - ***Notifies*** dependent microservices of change
  - ***Soft guarantees*** on when the updates synchronize



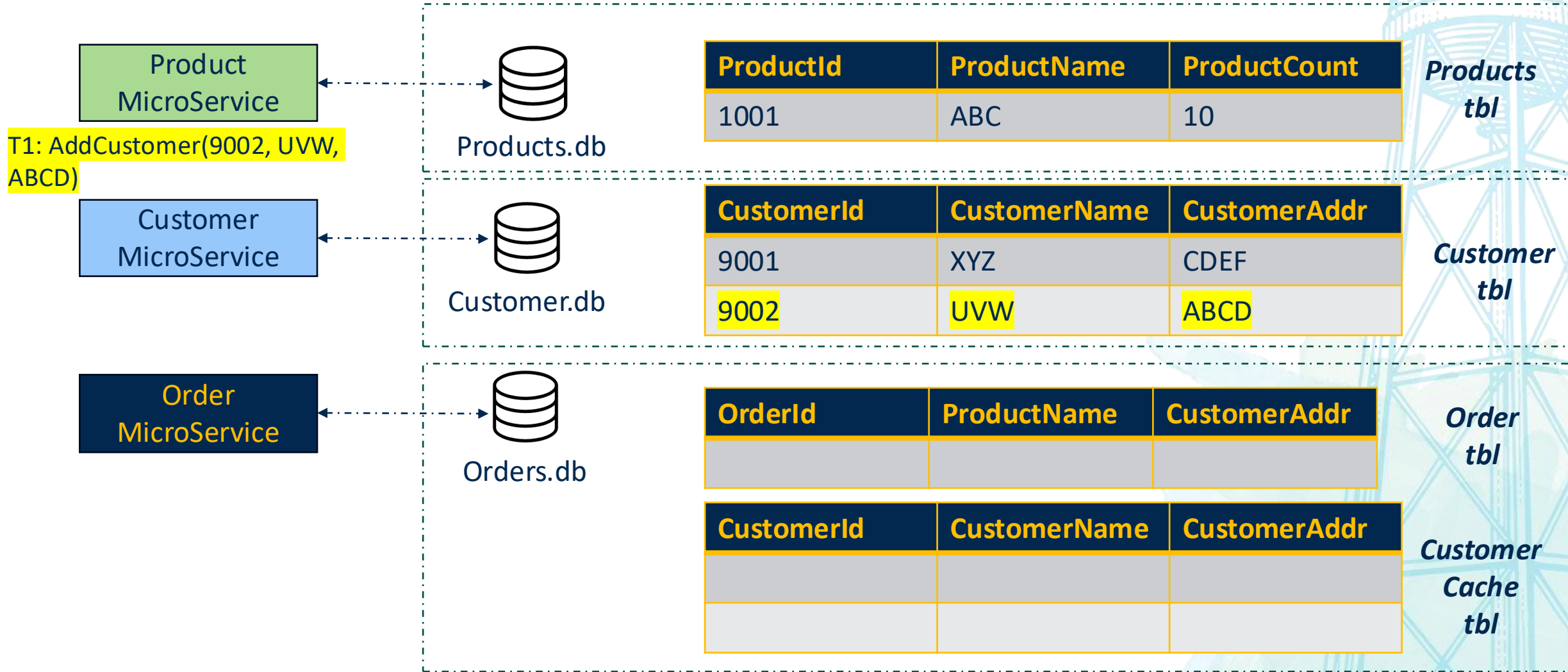
# Microservice consistency guarantees

- 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

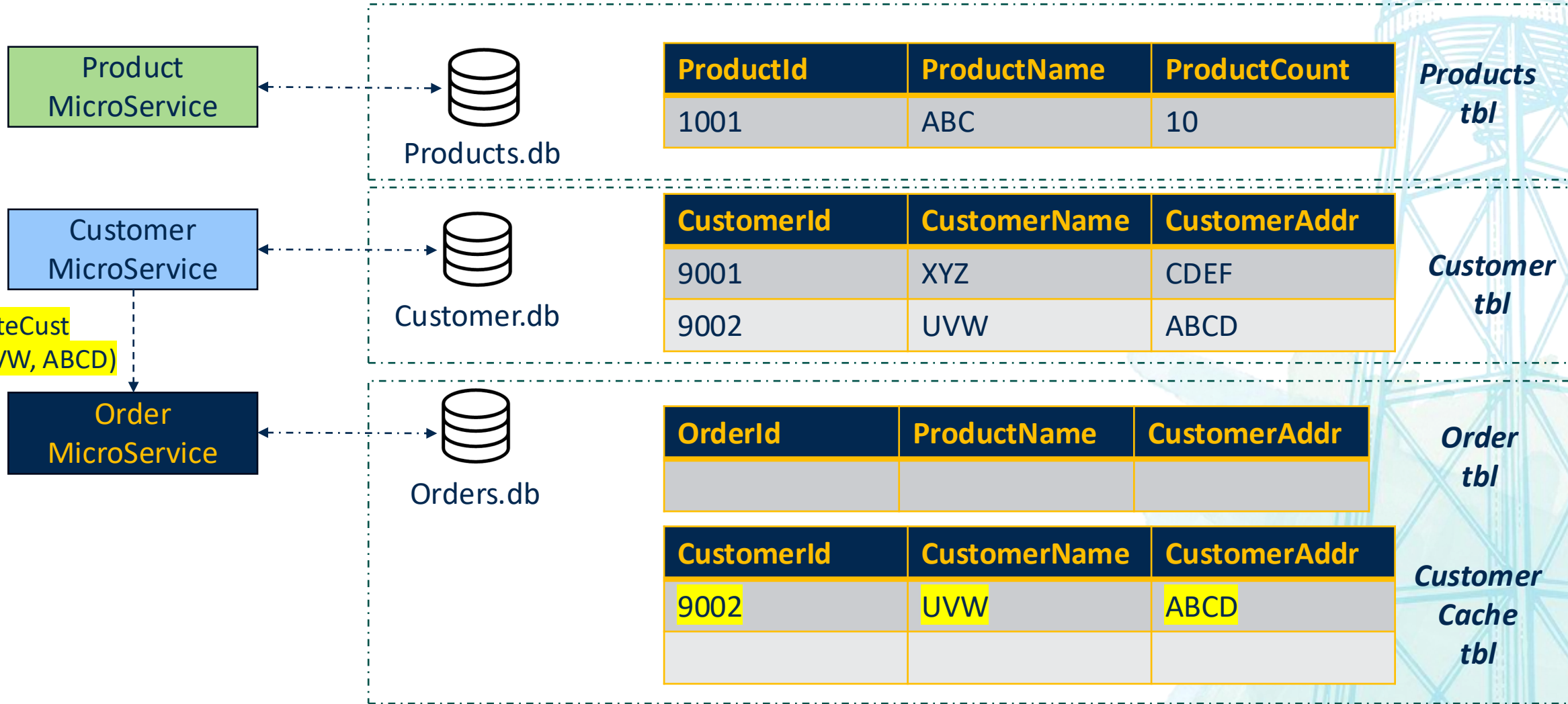
# Working with eventual consistency



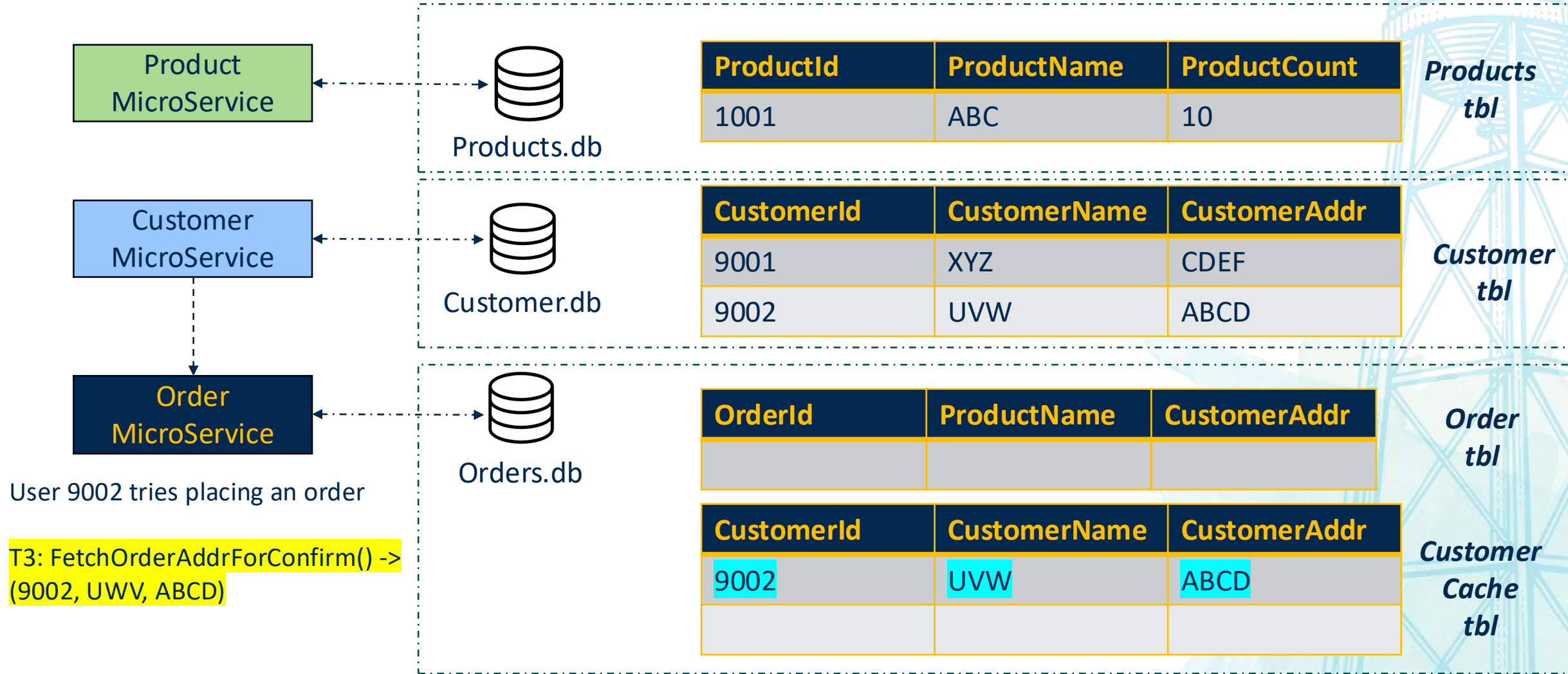
# Working with eventual consistency



# Working with eventual consistency



# Working with eventual consistency



# Working with eventual consistency

**Name: UVW,**  
**Address: ABCD**

amazon prime

Secure checkout ▾

Delivering to Tapti Palit


Add delivery instructions

FREE pickup available nearby ▾

Paying with [REDACTED]

Use a gift card, voucher, or promo code

Arriving Feb 14, 2026 - Feb 20, 2026



Harney & Sons Loose Leaf Black Tea, Darjeeling 8 Ounce

**\$18.00 (\$2.25 / ounce)**

Ships from and sold by Amazon.com

1

+

Add gift options

Subscribe & Save:

☐ Save 5% today; Save up to 15% on future auto-deliveries ▾

Delivery every: 3 months (most common)

Saturday, Feb 14 - Friday, Feb 20

FREE

Change

Place your order

By placing your order, you agree to Amazon's [privacy notice](#) and [conditions of use](#).

Items:

Shipping & handling:

Estimated tax to be collected:

**Order total:**

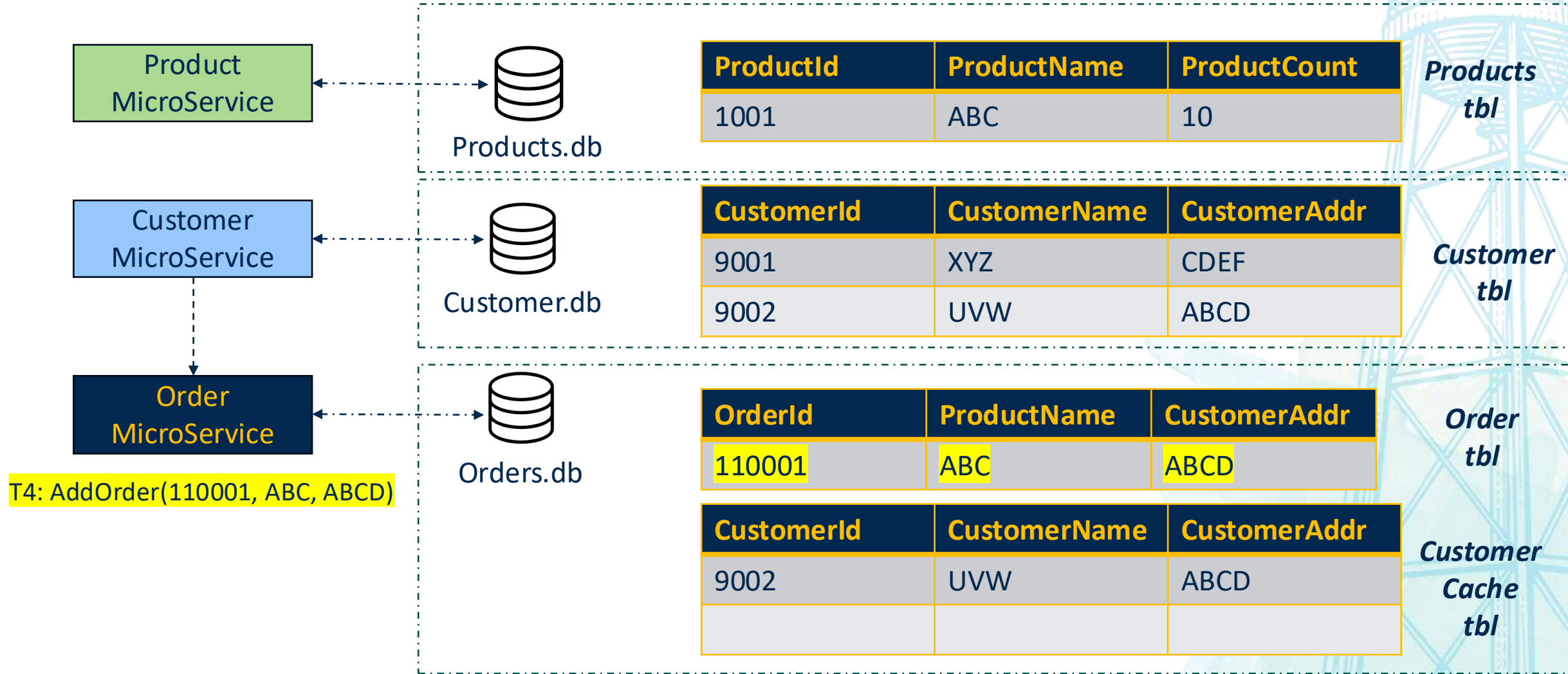
\$18.00

\$0.00

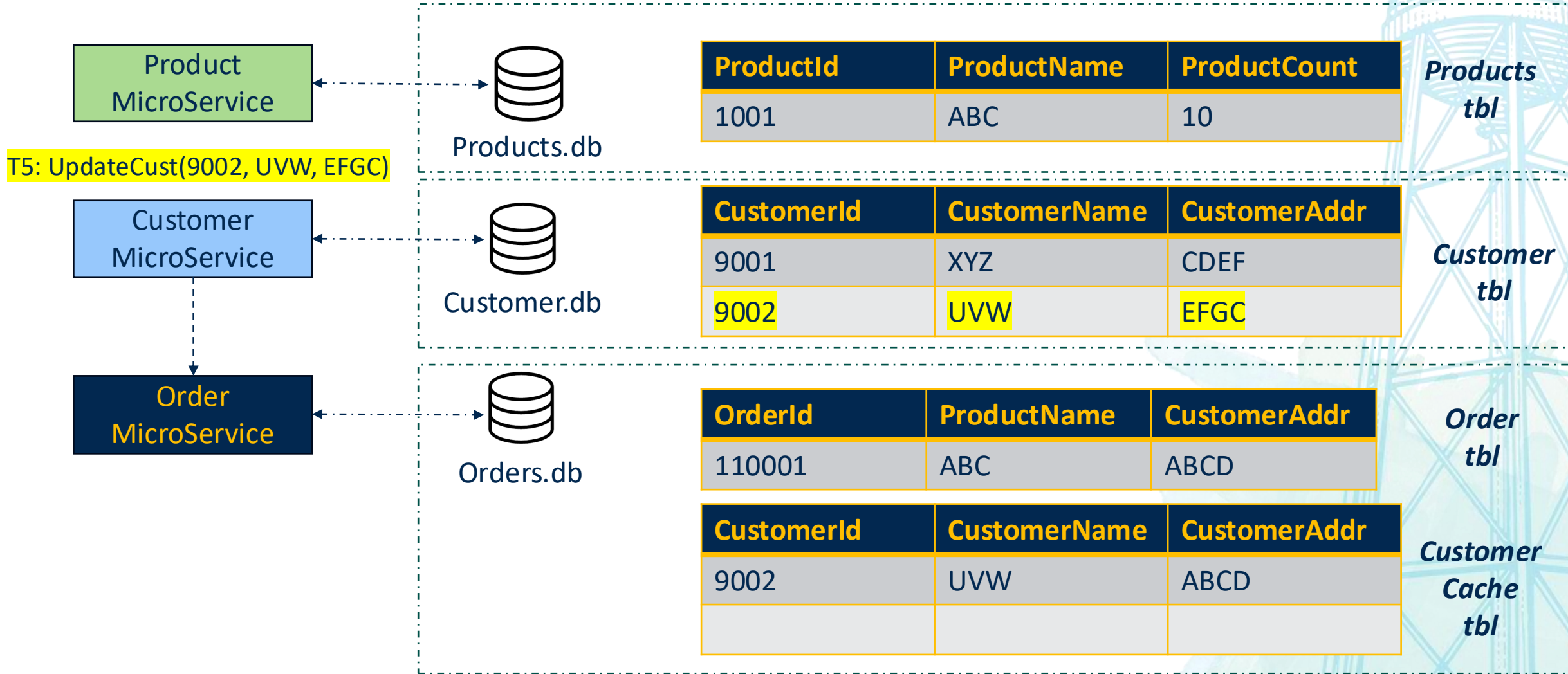
\$0.00

**\$18.00**

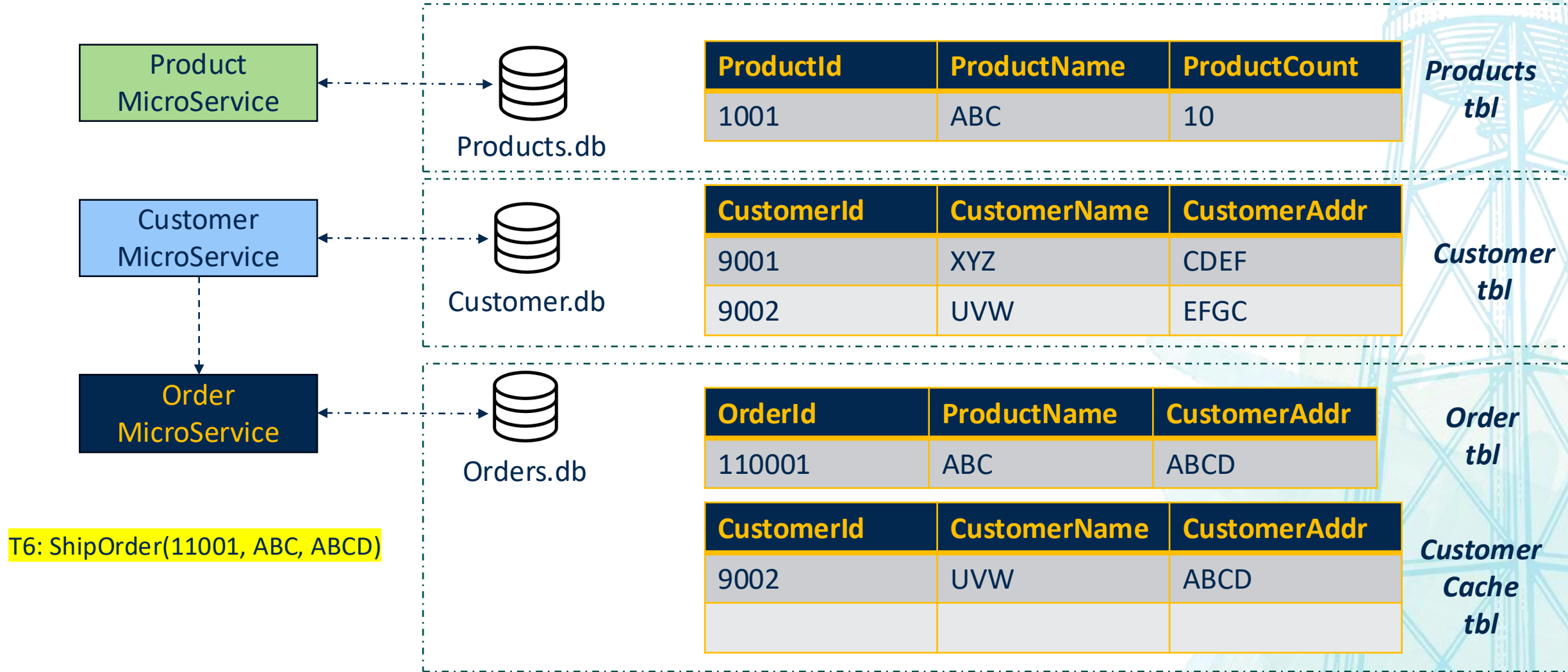
# Working with eventual consistency



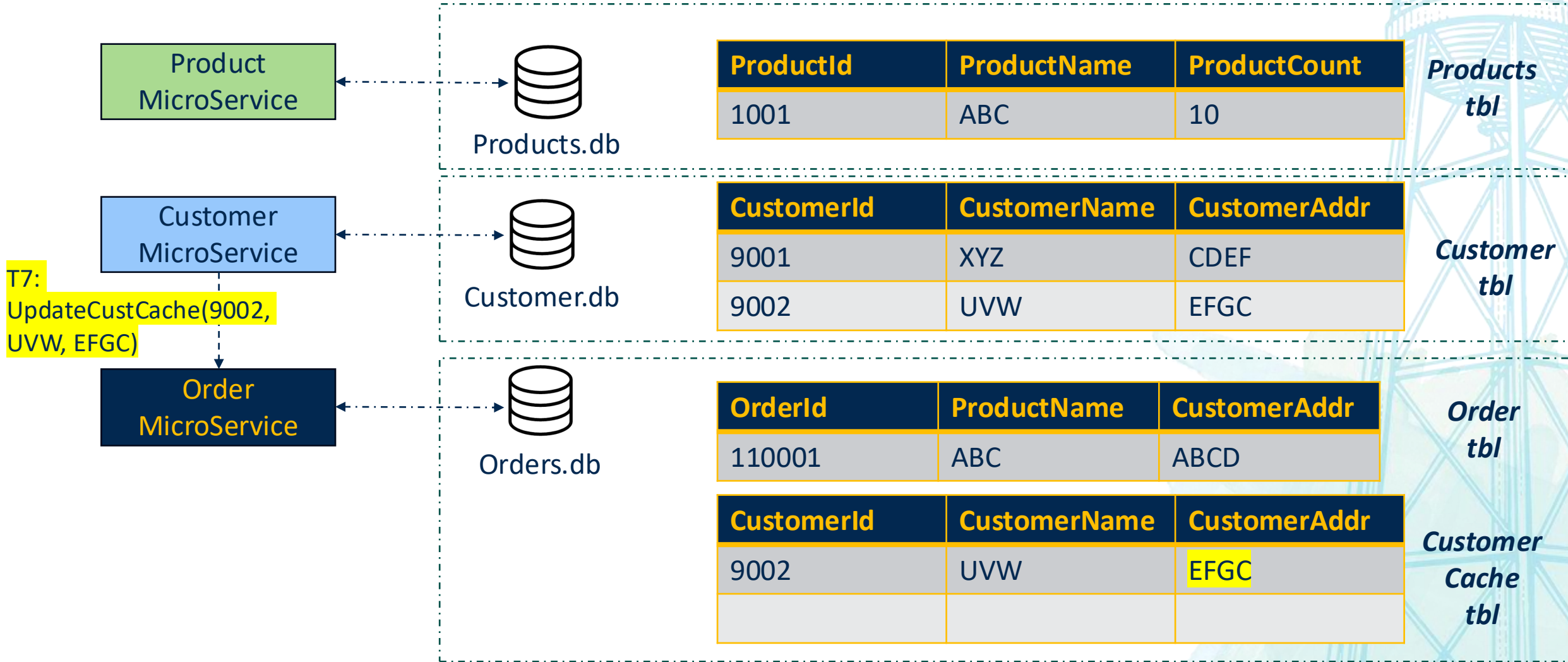
# Working with eventual consistency



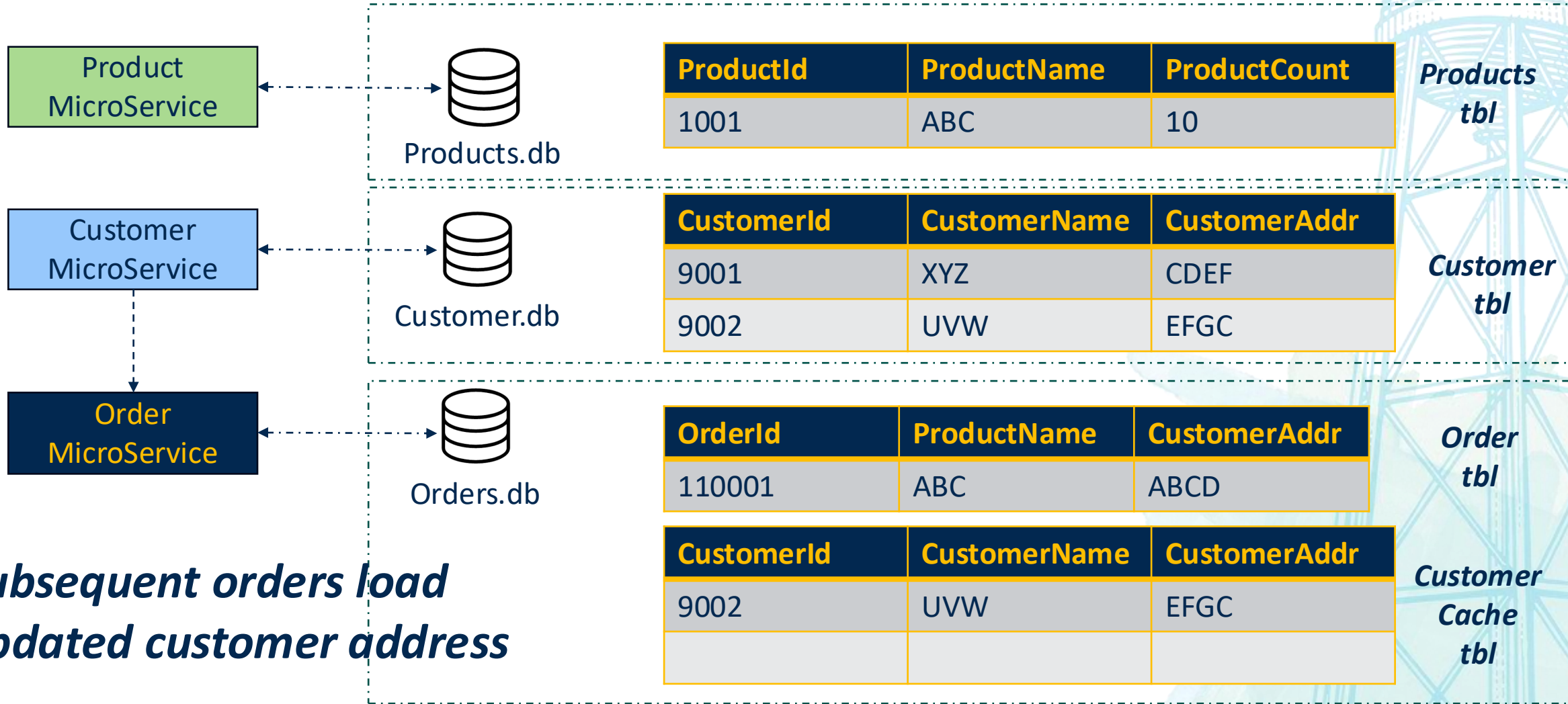
# Working with eventual consistency



# Working with eventual consistency



# Working with eventual consistency

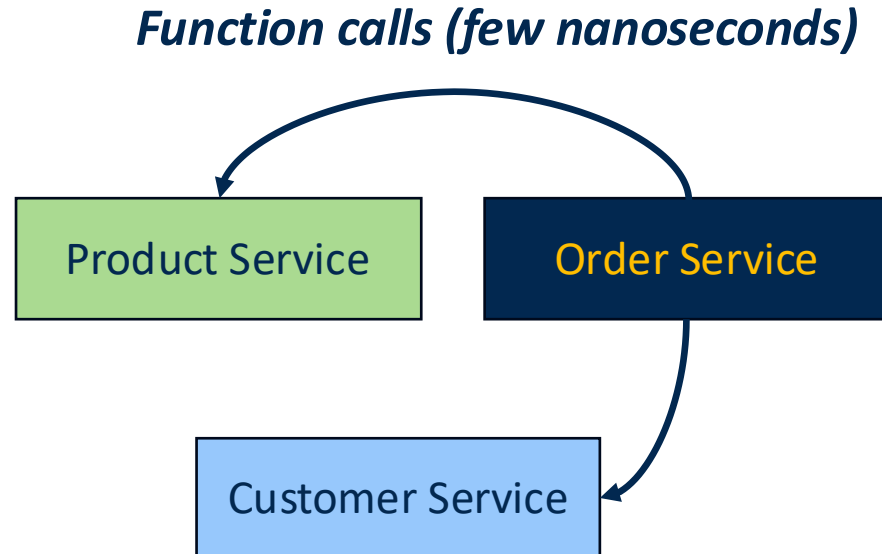


# Microservice design concerns

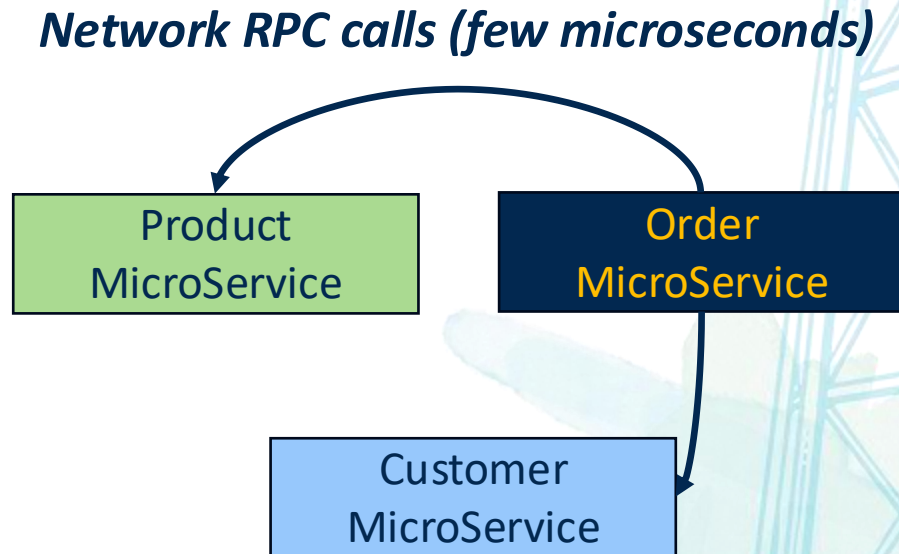
- Solution for eventual consistency is generally use-case dependent
- Reduce tight coupling
  - A microservice should not know the internal DB schema of another microservice
  - A microservice should not depend on another to be running to perform its task
- Limit network communication

# Network communication cost

## Monolithic applications



## Microservices



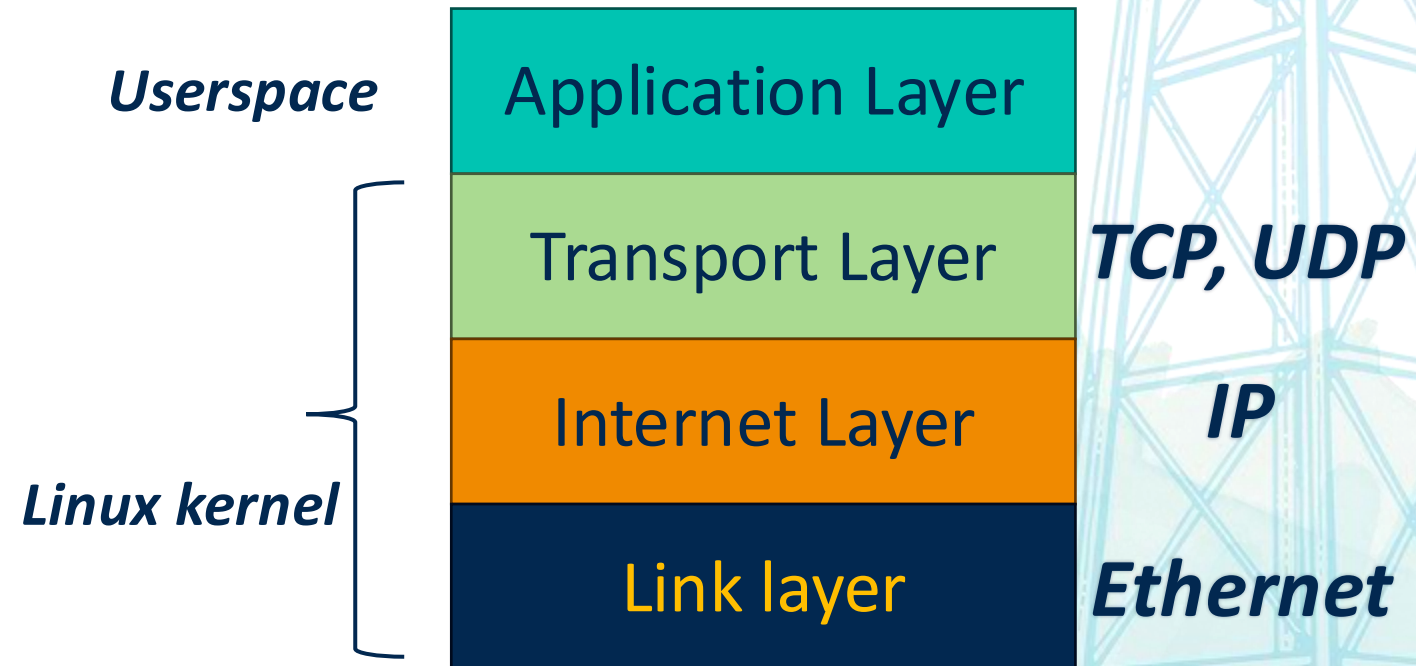
# Network overhead decomposition

- 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



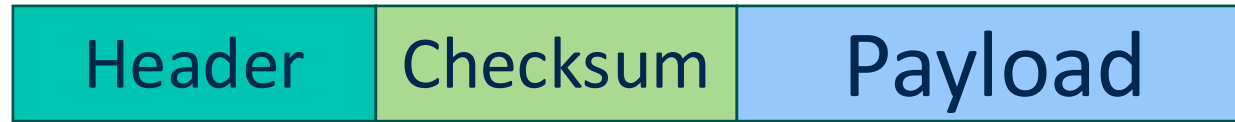
# Kernel overhead for network comm. (Not in syllabus)

- 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***

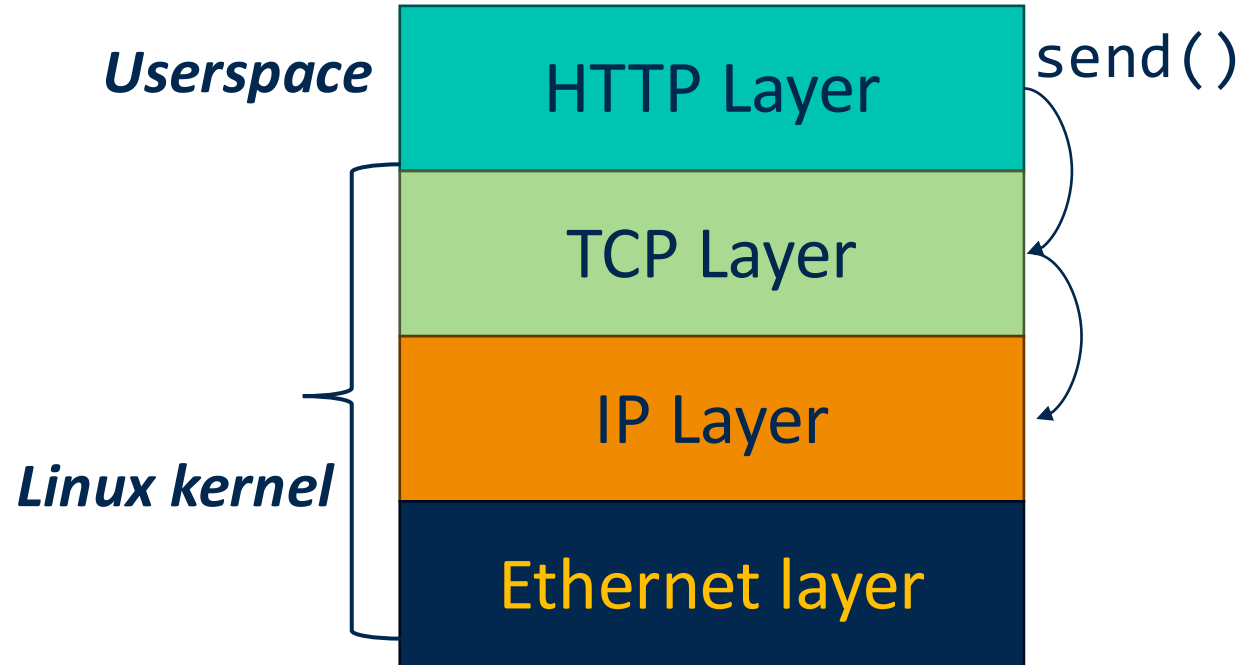


# Each layer has a protocol (Not in syllabus)

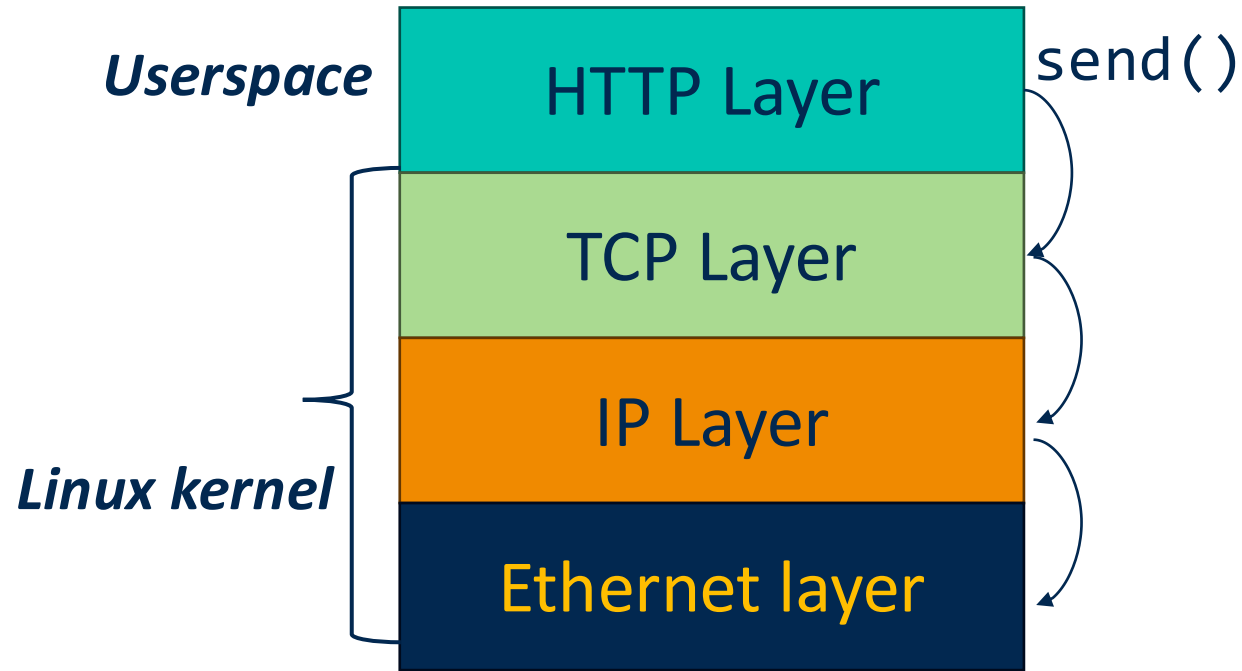
- Every layer/protocol has a fixed message format
  - Header
  - Payload
  - [optional] Checksum
- As the packet traverses through the layers, packets are rewrapped



# Life of a packet (Not in syllabus)

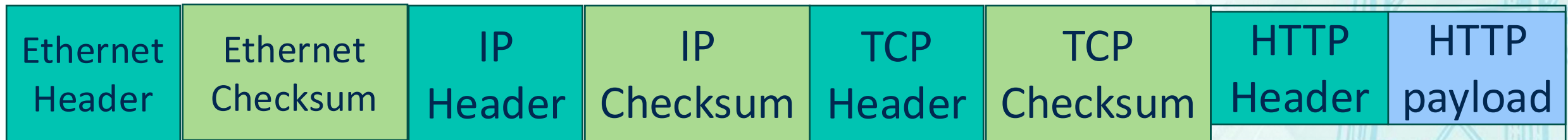


# Life of a packet (Not in syllabus)



*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
- Easier deployment



# Microservice cons

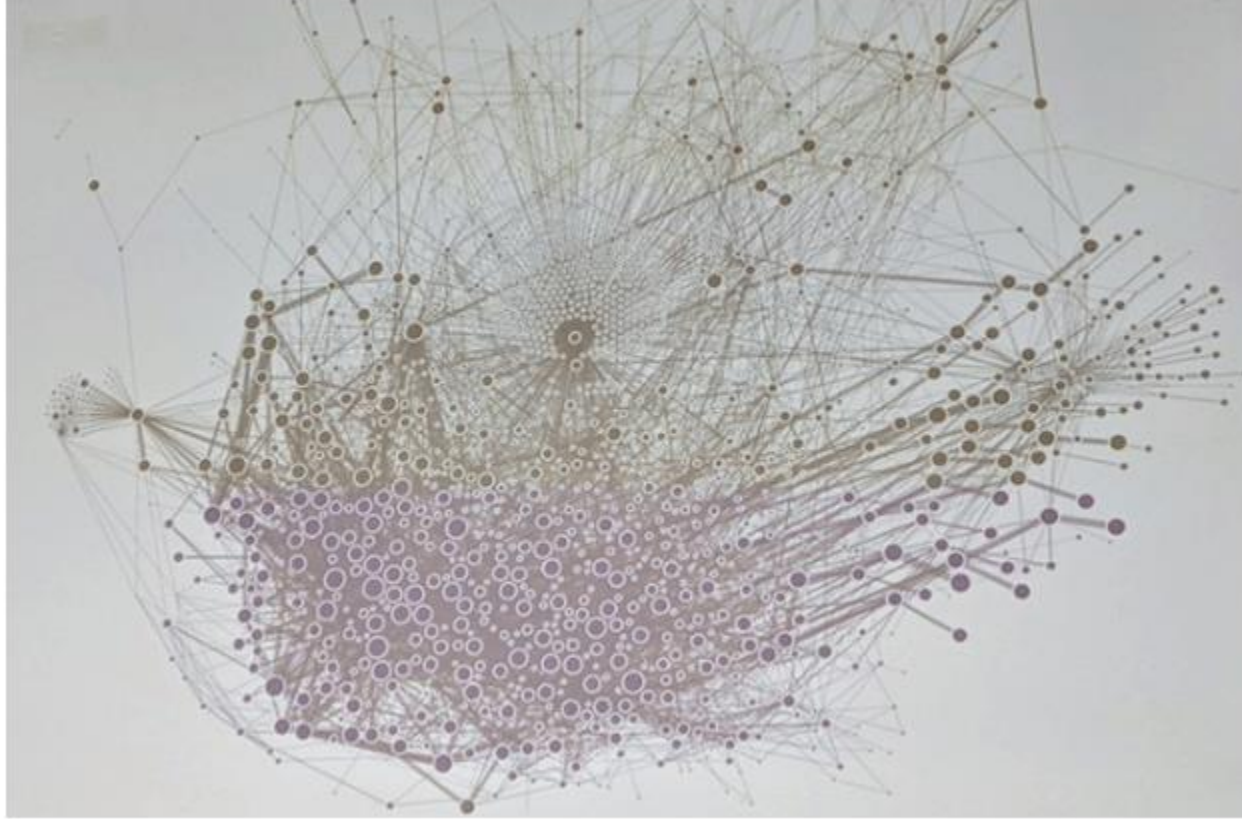
- Causes data denormalization
- Network overhead
- Higher complexity
- Debugging complex interactions is harder



# 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

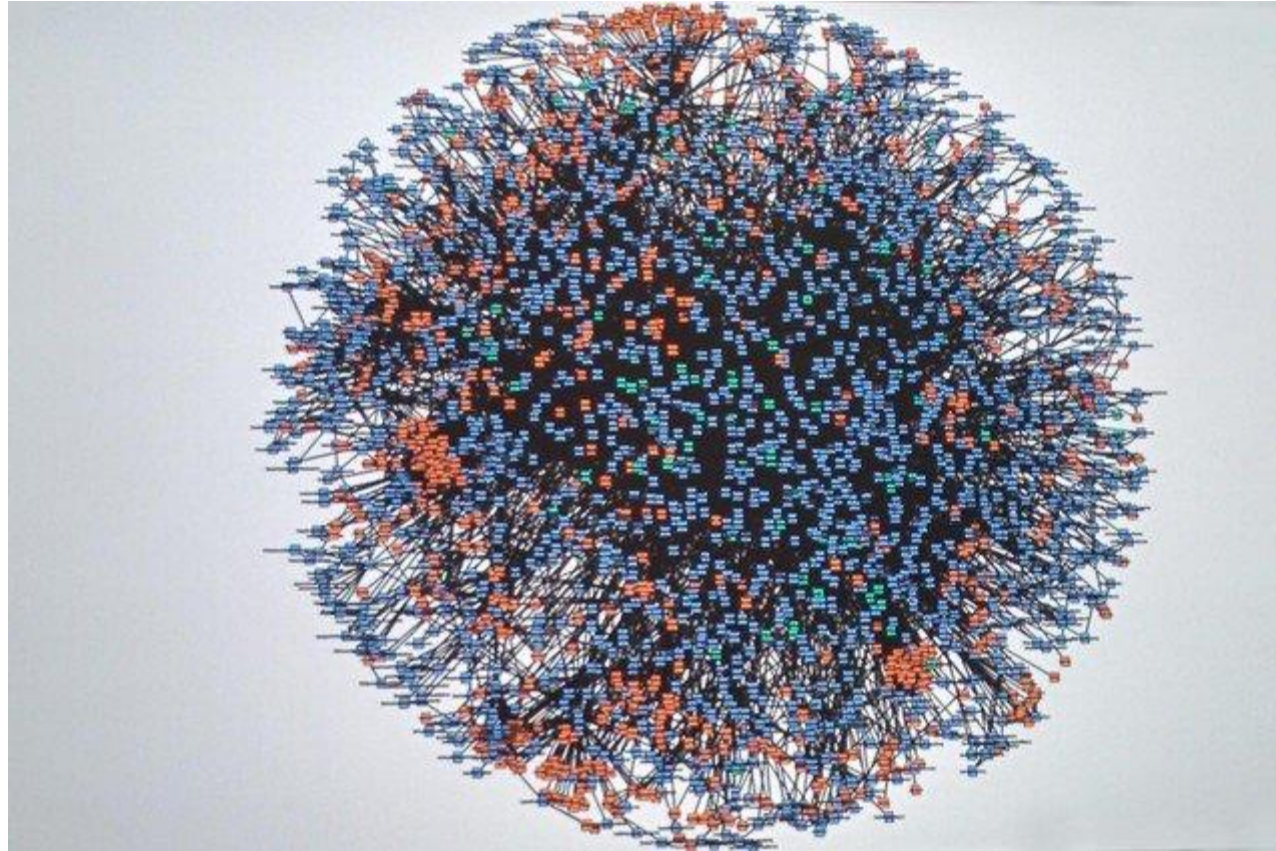
# Microservices at Uber (2019)



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

# Microservices at Amazon (2008)

- Code-named “Deathstar”



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

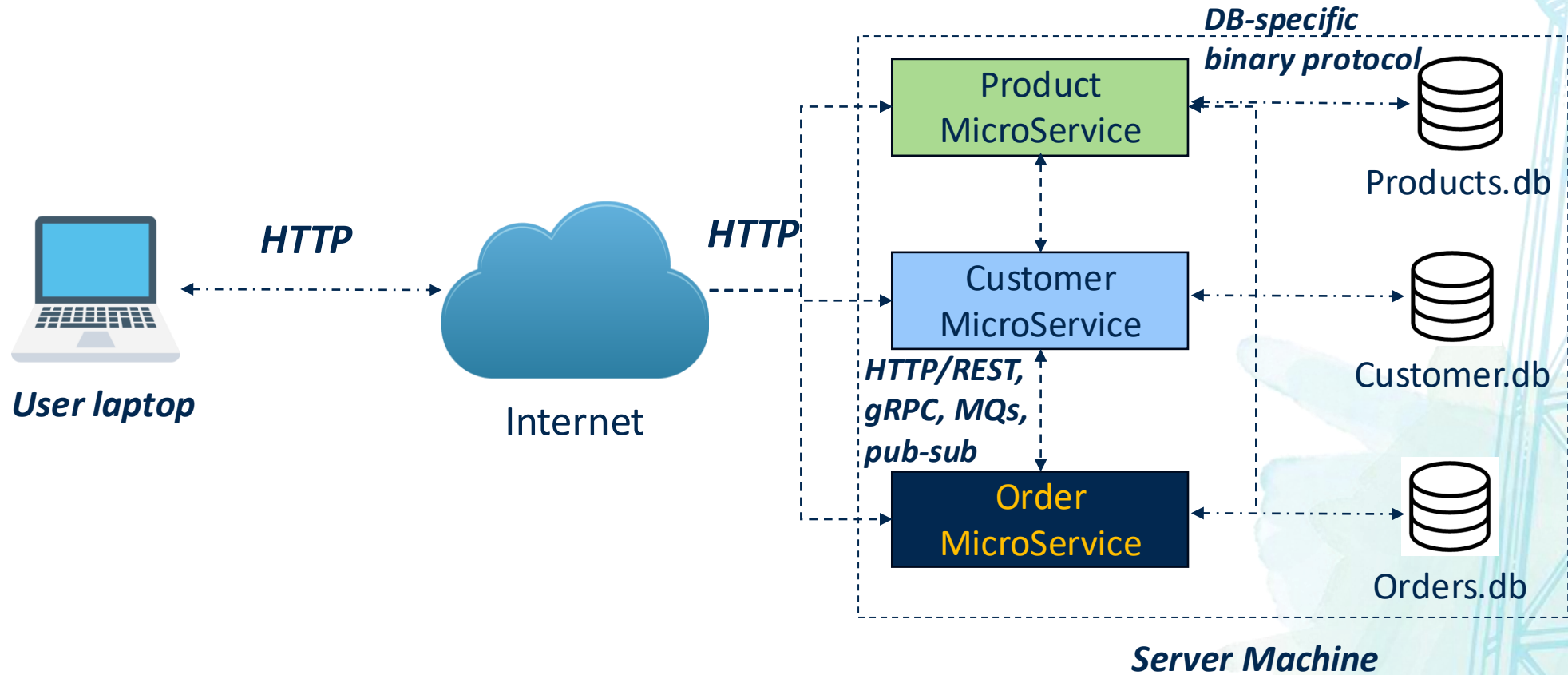
# Data management and communication

- 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

- POST **URL** `/users` HTTP/1.1

Content-Type: application/json{

"name": "John Doe",

"email": john.doe@example.com

}

***Post "body"***

# 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`**

# 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

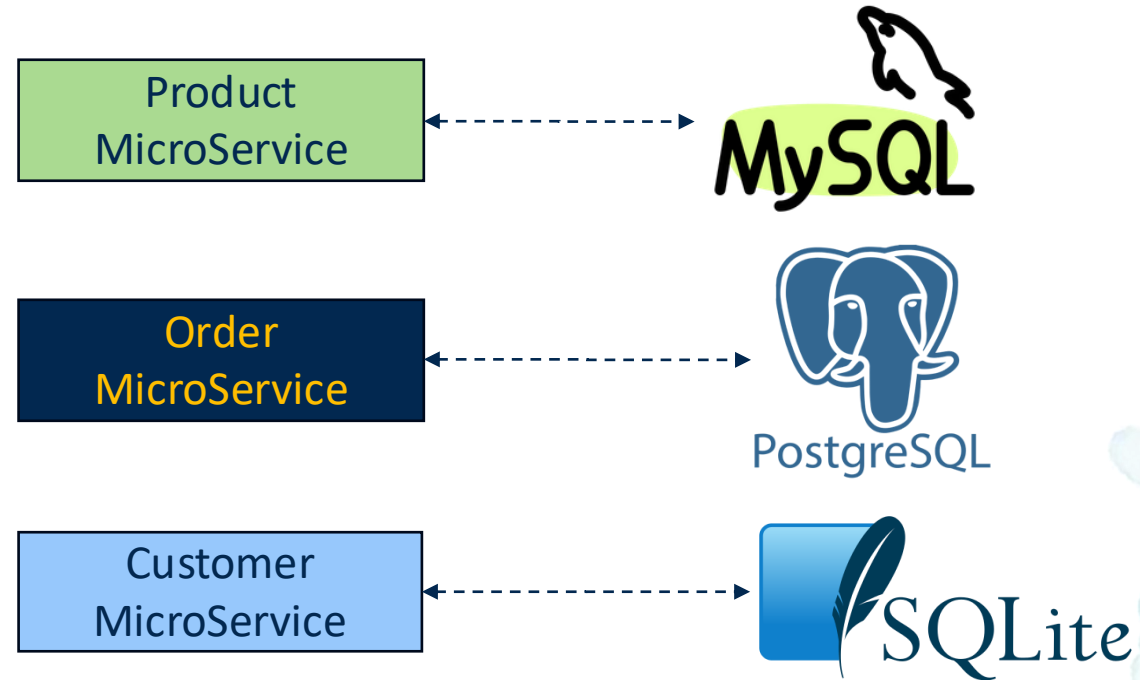
- Once data and computation are split across services, two problems remain
  - How is data stored?
  - How services communicate?



# Databases

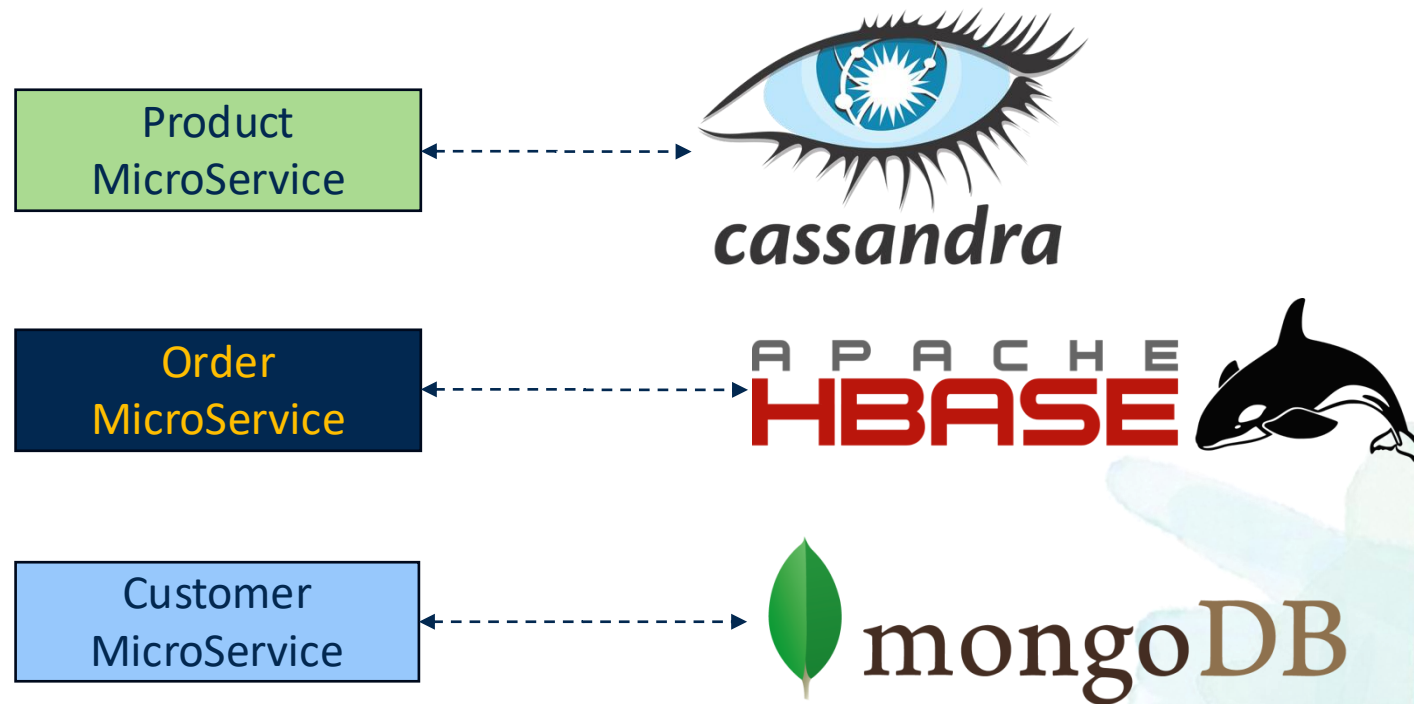


# Microservices can have individual DBs



*MySQL, PostgreSQL, SQLite are relational databases*

# Microservices can have individual DBs



*Non-relational, NoSql databases*

# How to select which database to use?



# 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)

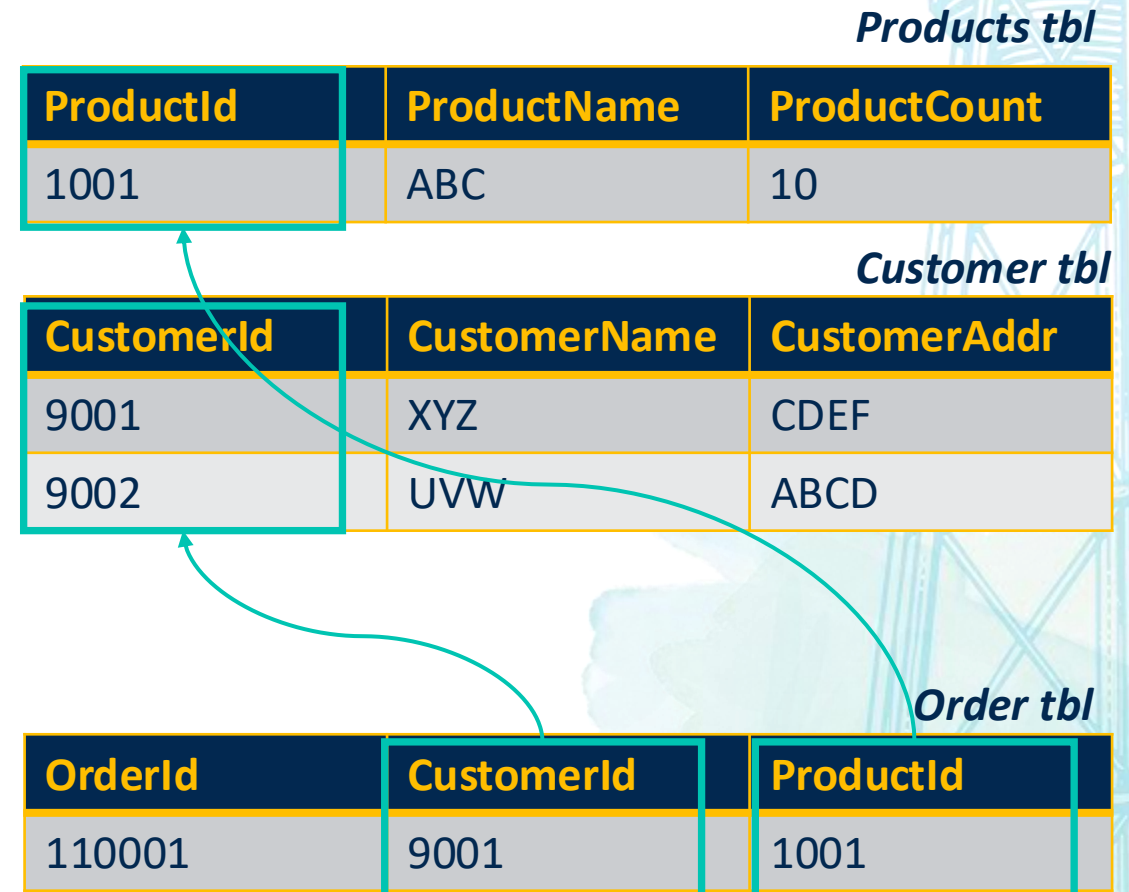
# Data models and query languages

- 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



# 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

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

*JSON object*

*Nested JSON object*

*JSON array*

# 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)

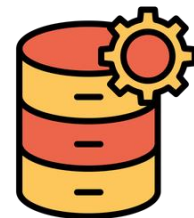
Key	Value	
	Field	Value
10279811	Name	ABC
	Age	22
	GPA	3.8
	Credits	45
10279812	Name	DEF
	Age	21
	GPA	3.9
	Credits	60

*Students Redis DB*

# Key-value stores

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

```
SELECT * FROM  
T1 JOIN T2  
JOIN T3  
ON T1.id = ...
```



*Orders db*



redis

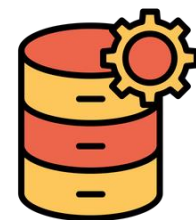


*OrderService.java*

# Key-value stores

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

```
SELECT * FROM  
T1 JOIN T2  
JOIN T3  
ON T1.id = ...
```



*Orders db*



redis

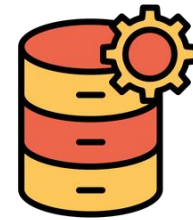
10000 records



*OrderService.java*

# Key-value stores

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



*Orders db*



**redis**

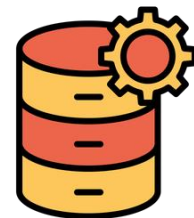
`Set(query-101, /* 10000 records */)`



*OrderService.java*

# Key-value stores

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



*Orders db*



redis

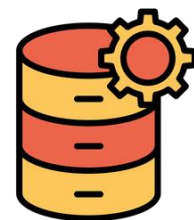
GET(query-101)



*OrderService.java*

# Key-value stores

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



*Orders db*



redis

10000 records



*OrderService.java*

# 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  $O(n)$

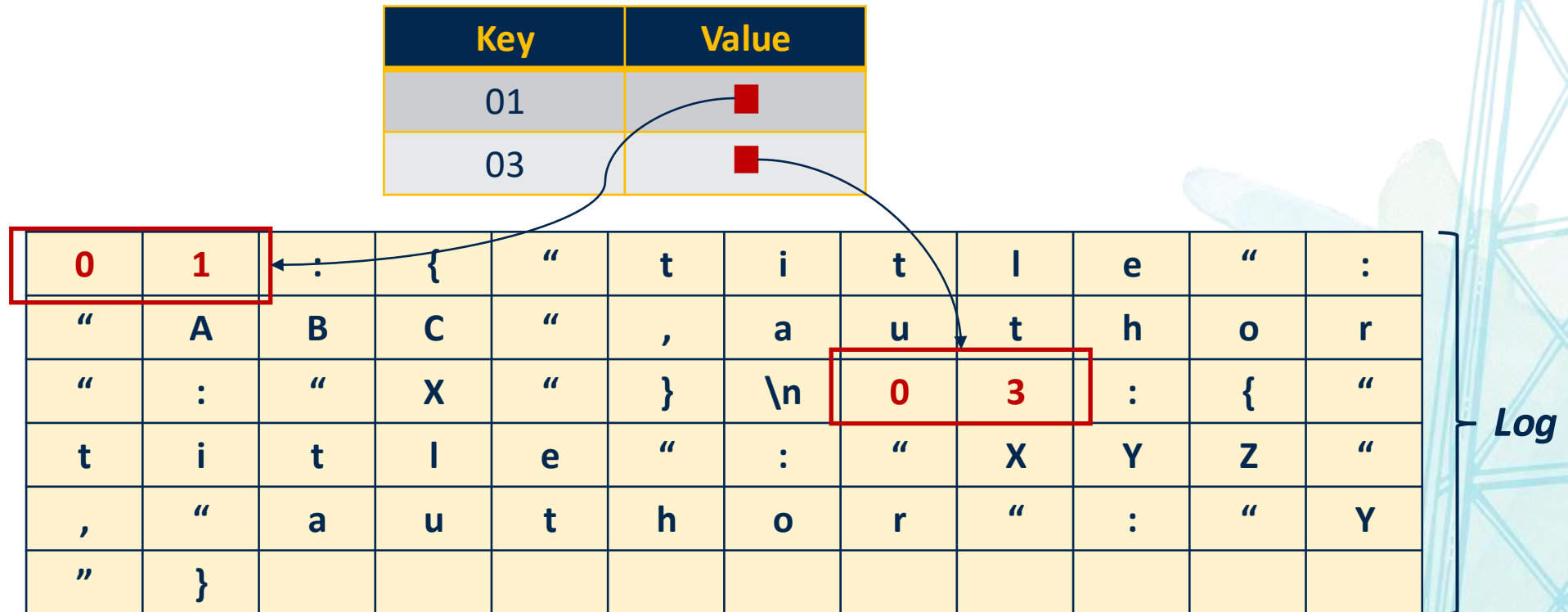
```
{  
  "title": "ABC",  
  "tags": ["S", "T"],  
  "author": "X"  
}
```

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

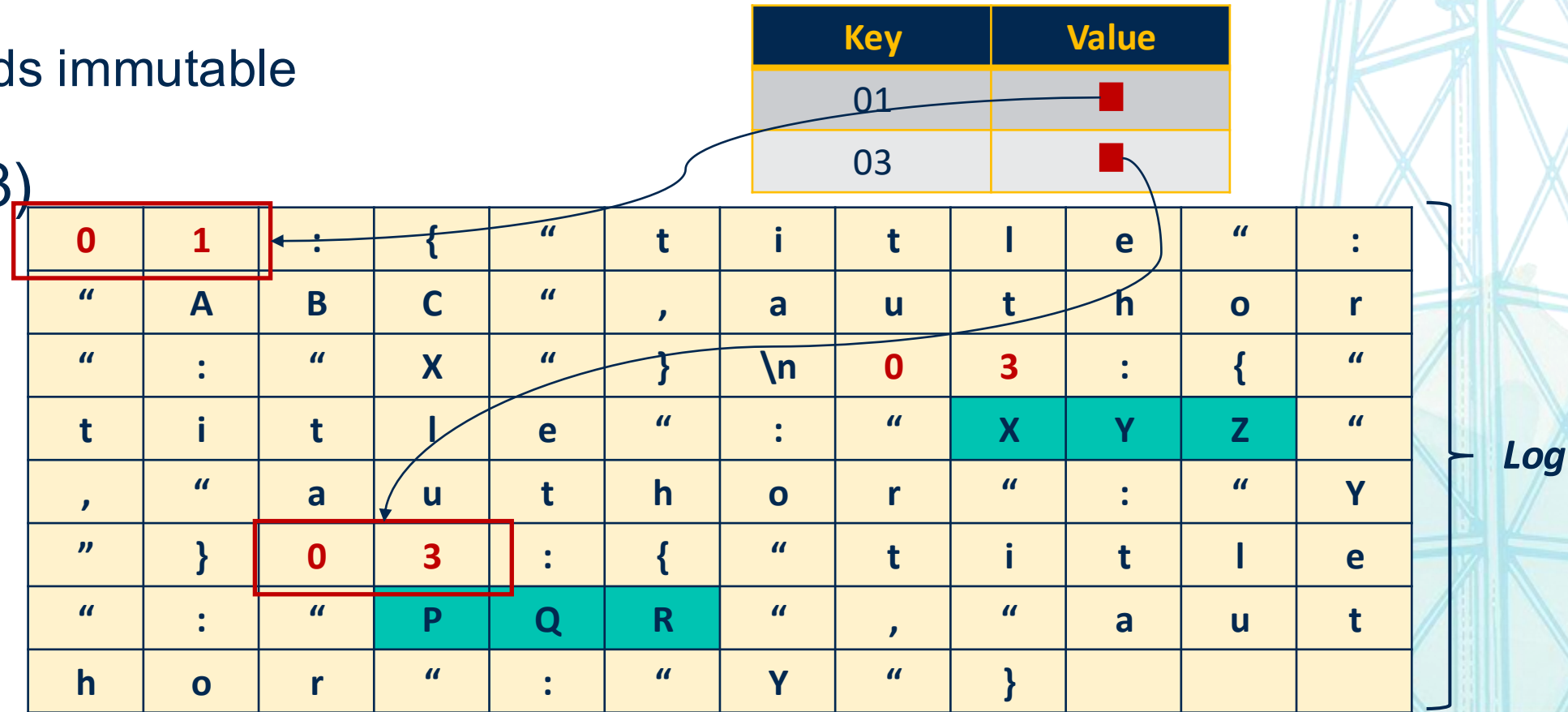
# Hash indices

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





# Hash indices – update operation

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



# Compaction of hash indices

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

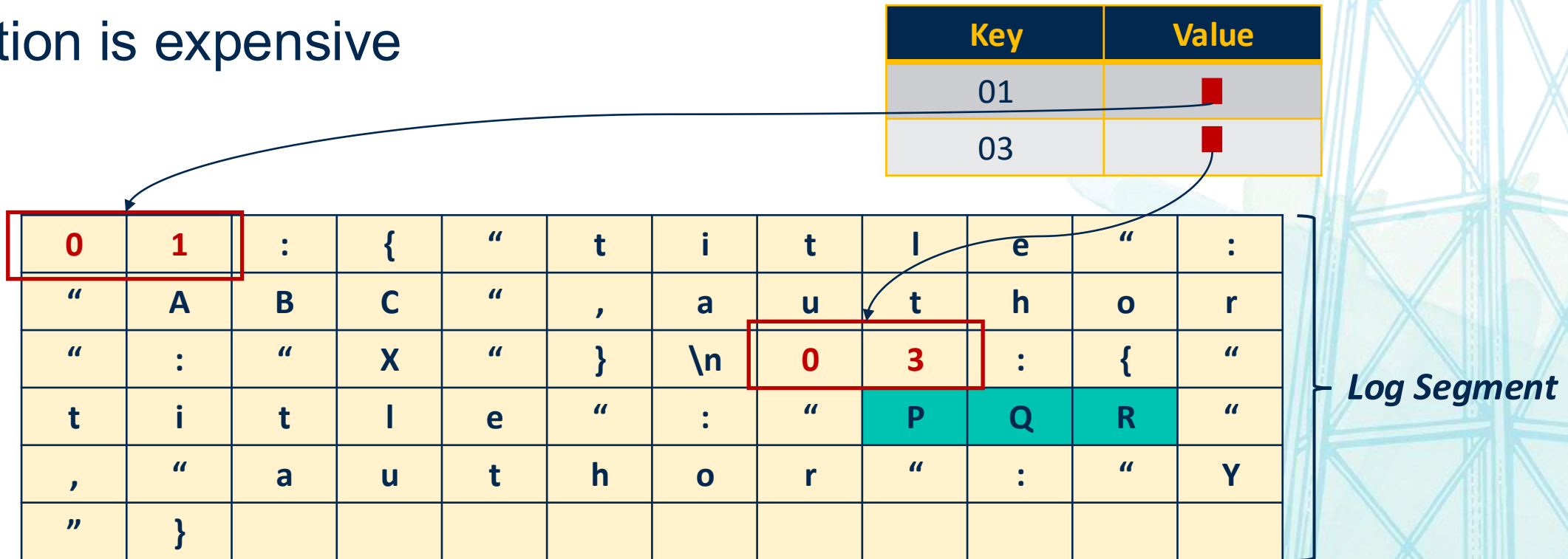
Key	Value
01	
03	

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 needs to "remember" the latest value of a key
- Compaction is expensive



# Hash indices limitations

- High memory usage for large logs
  - A sparse hash index can overcome this limitation, but needs sorting
- 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 is expensive
- Solution: SSTables (memtable) and LSM trees

# Log structured merge tree (LSM tree)

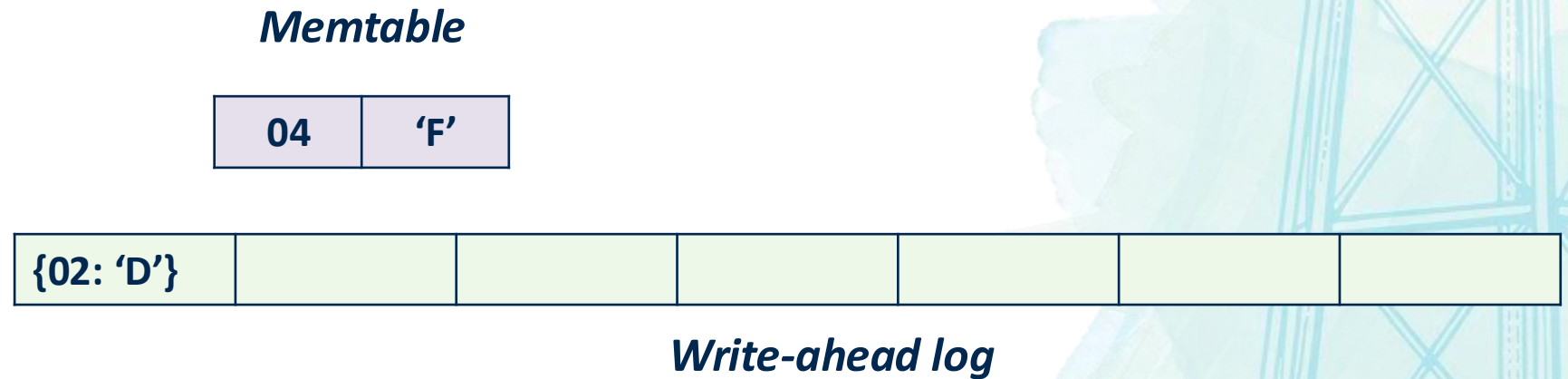
- Designed by Google as part of their distributed database BigTable
  - Chang, Fay, et al. "Bigtable: A distributed storage system for structured data." *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008)
- Adopted by many recent databases (distributed or single-machine)
- LSM tree components
  - On-disk write-ahead log (WAL)
  - In-memory sorted tree (memtable)
  - Immutable on-disk sorted string table (SSTable)

# LSM tree operations



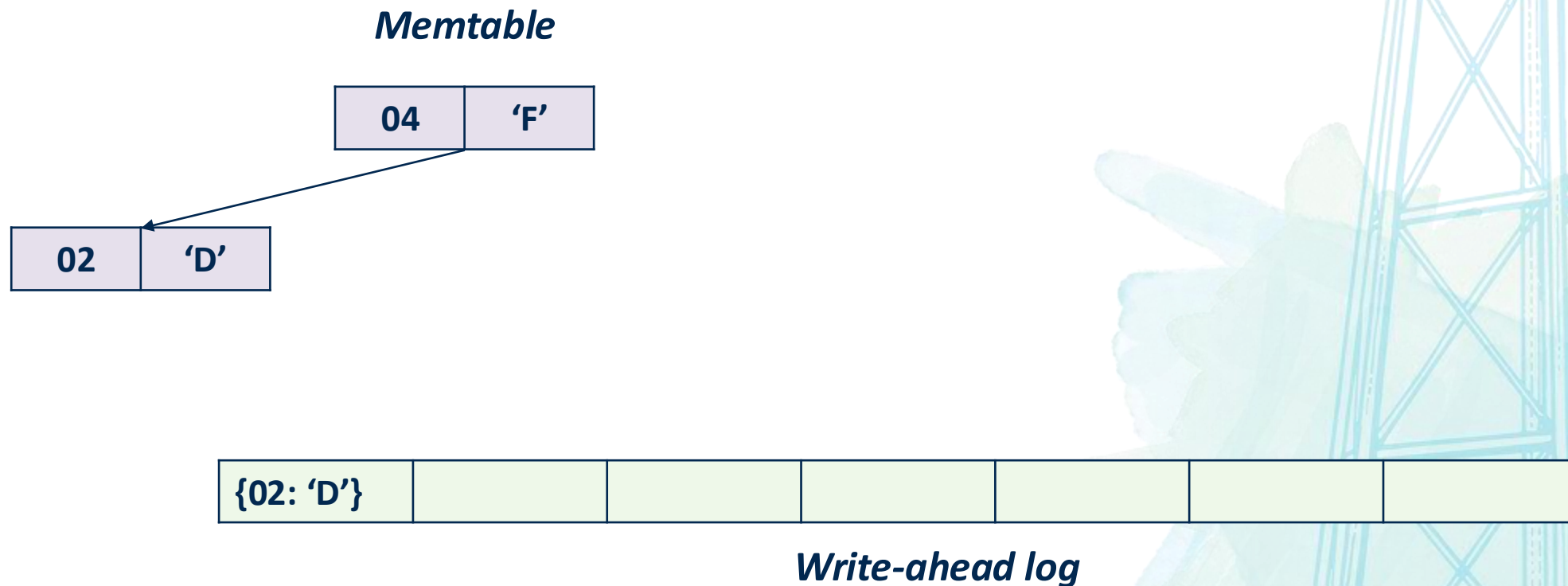
# Write-ahead log

- 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

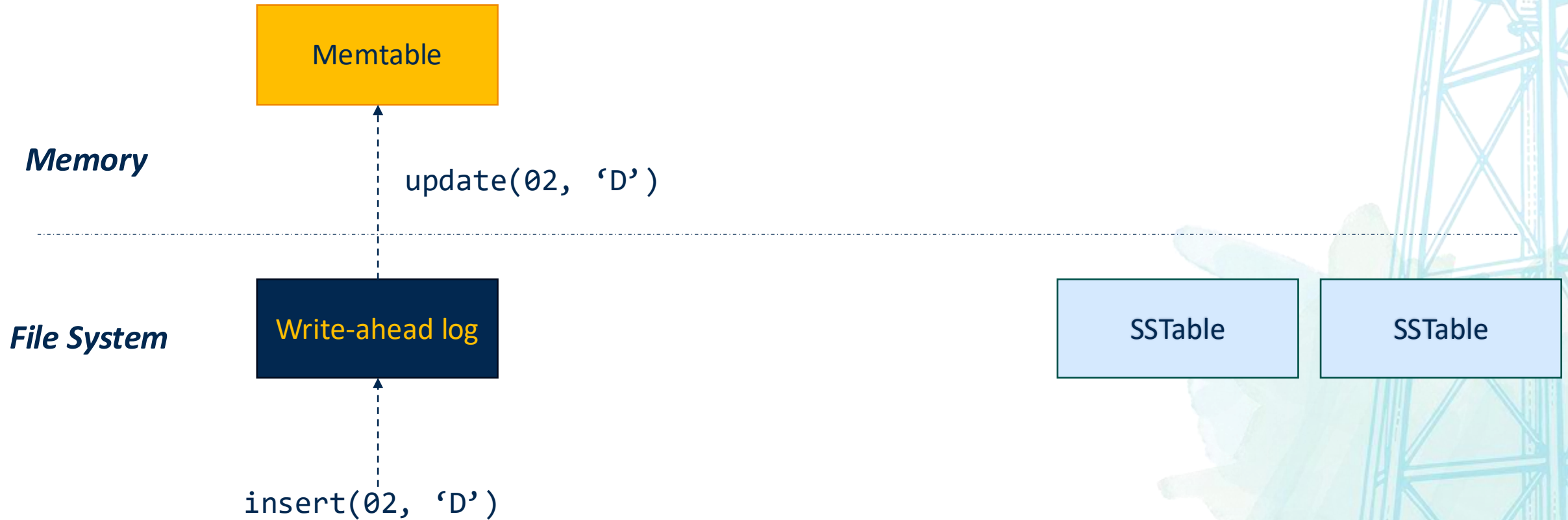


# Memtable

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

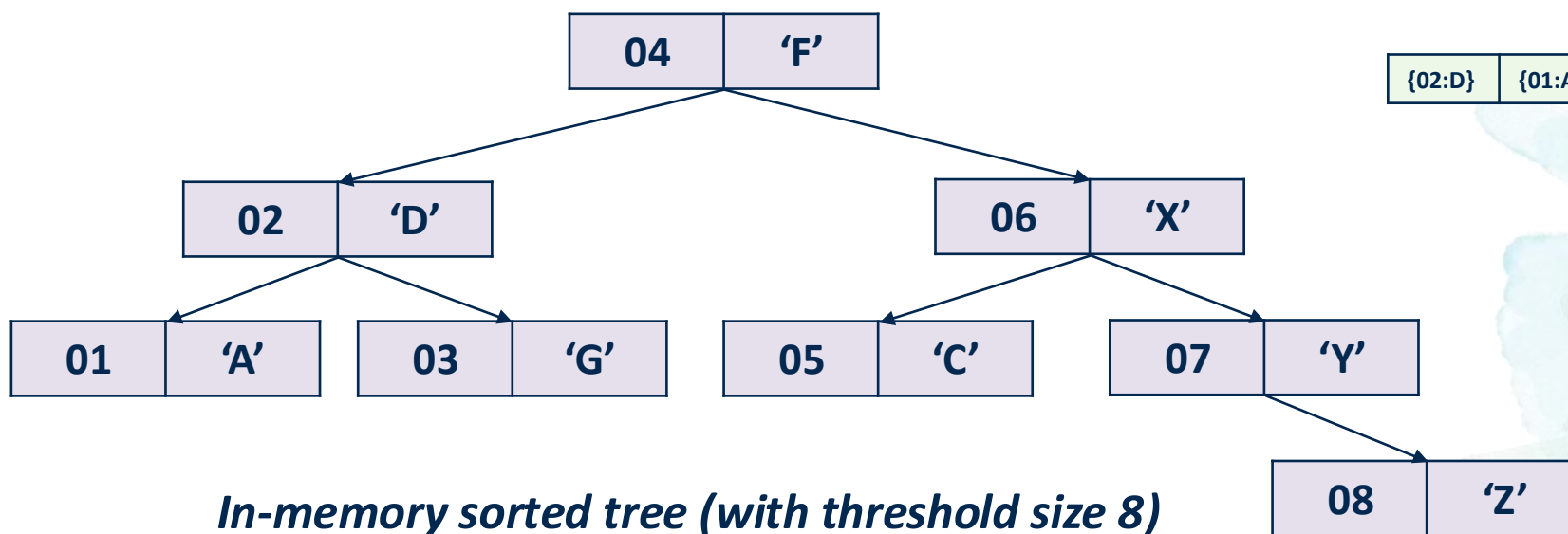


# LSM tree write operation



# Memtable

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



# Memtable

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

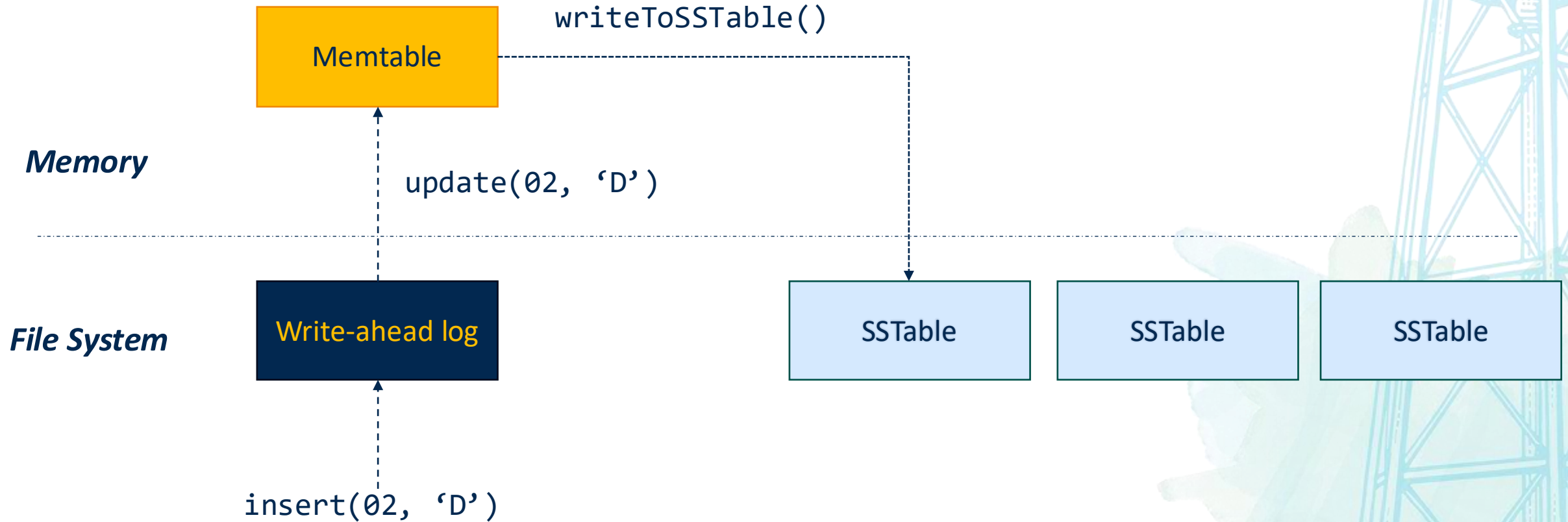
0	1	:	{	'A'	}	\n	0	2	:	{	'D'
}	\n	0	3	:	{	'G'	}	\n	0	4	:
{	'F'	}	\n	0	5	:	{	'C'	}	\n	0
6	:	{	'X'	}	\n	0	7	:	{	'Y'	}
0	8	:	{	'Z'	}						

*Sorted segment file  
(on disk)*

--	--	--	--	--	--	--

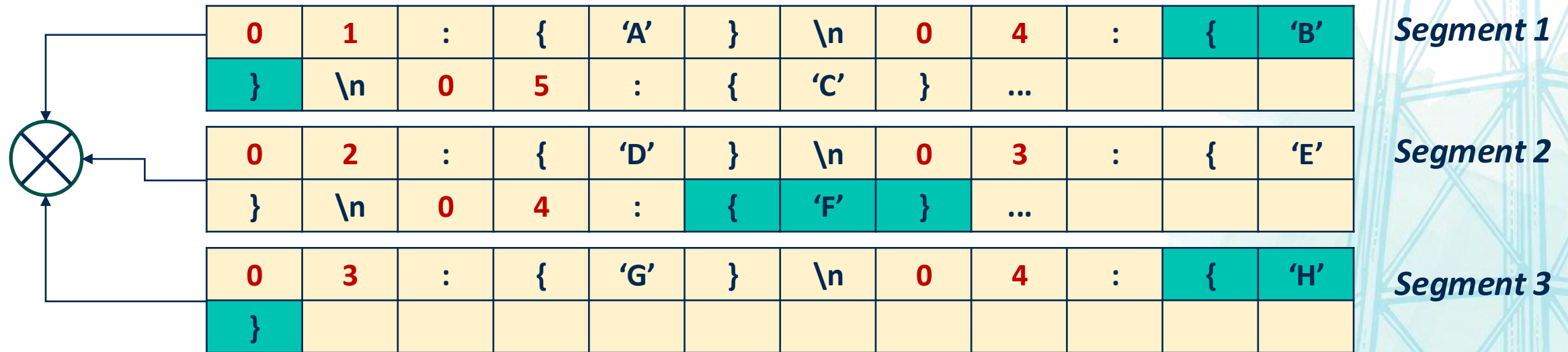
*Write-ahead log*

# 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



# SSTables compaction

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

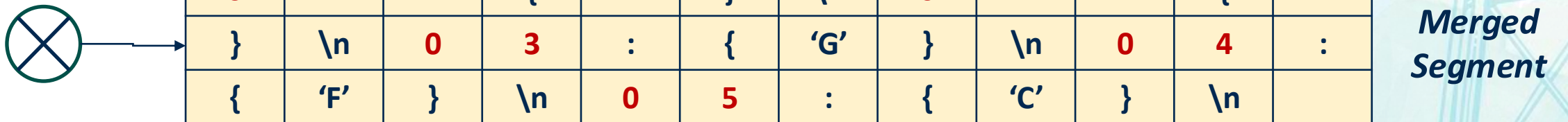


0	1	:	{	'A'	}	\n	0	2	:	{	'D'
}	\n	0	3	:	{	'G'	}	\n	0	4	:
{	'F'	}	\n	0	5	:	{	'C'	}	\n	

*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)



# LSM tree compaction operation

Memtable

*Memory*

*File System*

Write-ahead log

SSTable

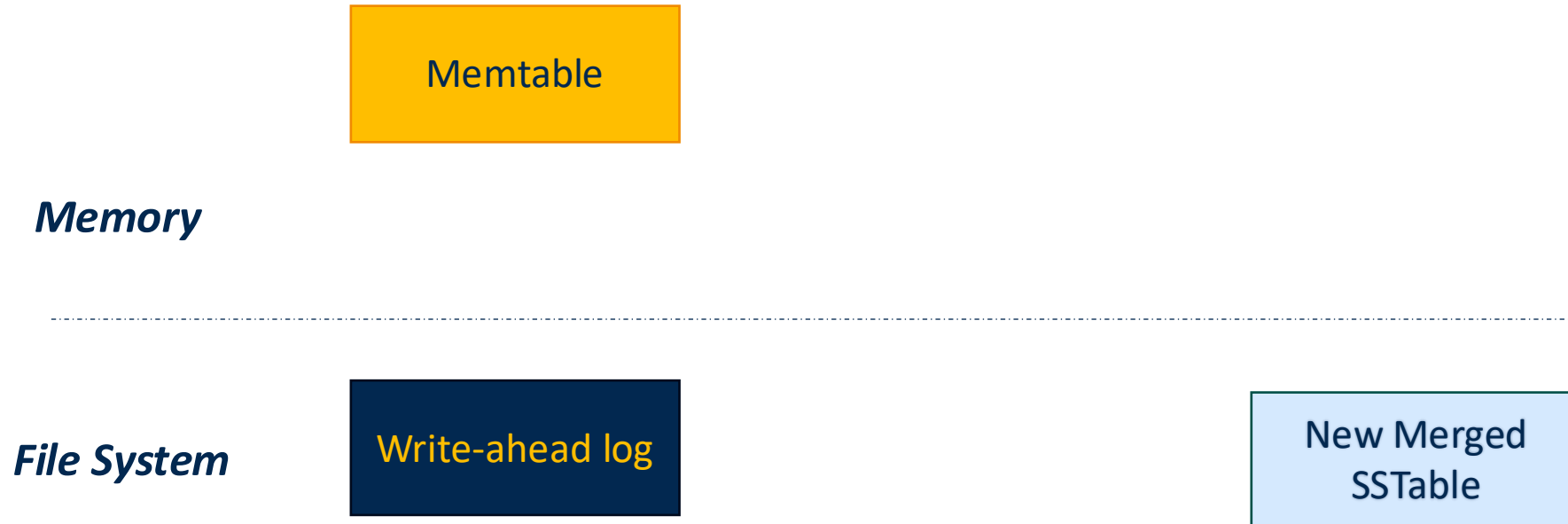
SSTable

SSTable

*Background compaction thread*

compact()

# LSM tree compaction operation



*New merged SSTable can participate in further compactions*

# Typical overheads of LSM write path

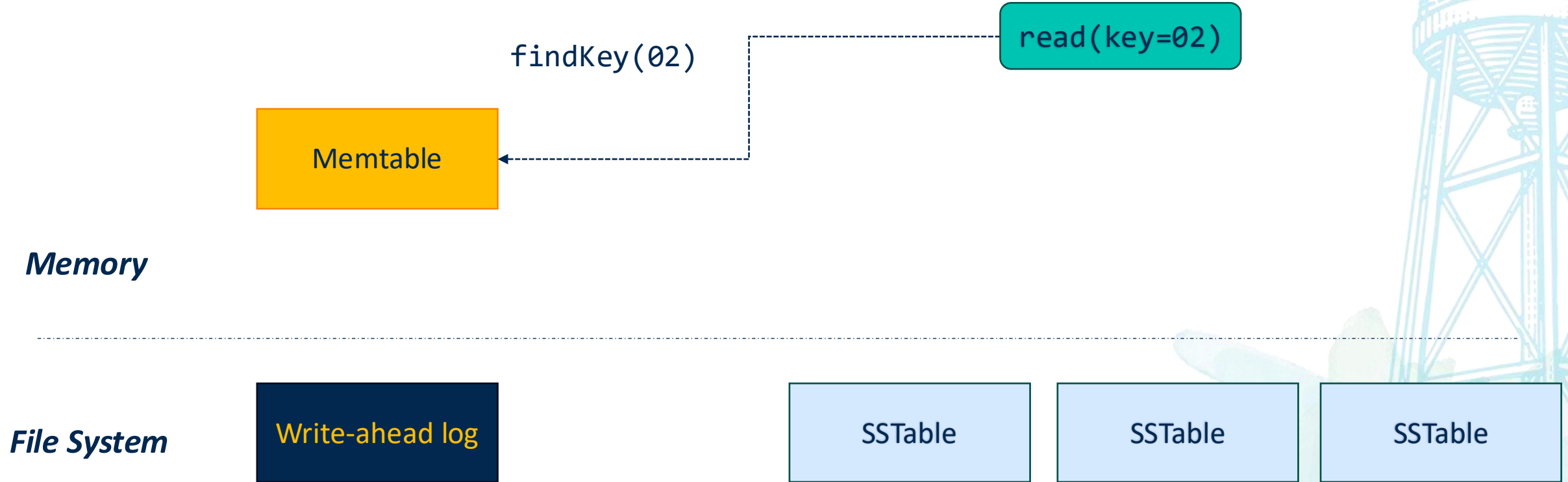
- Append to ***on-disk*** WAL has  $O(1)$  complexity
- Insert into ***in-memory*** memtable (AVL tree) has  $O(\log(n))$  complexity
- Eventually, append to ***on-disk*** SSTable has  $O(1)$  complexity per-key
  - Bulk write amortizes the disk overhead
- ***On-disk*** log compaction and merging runs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

# LSM tree read operation

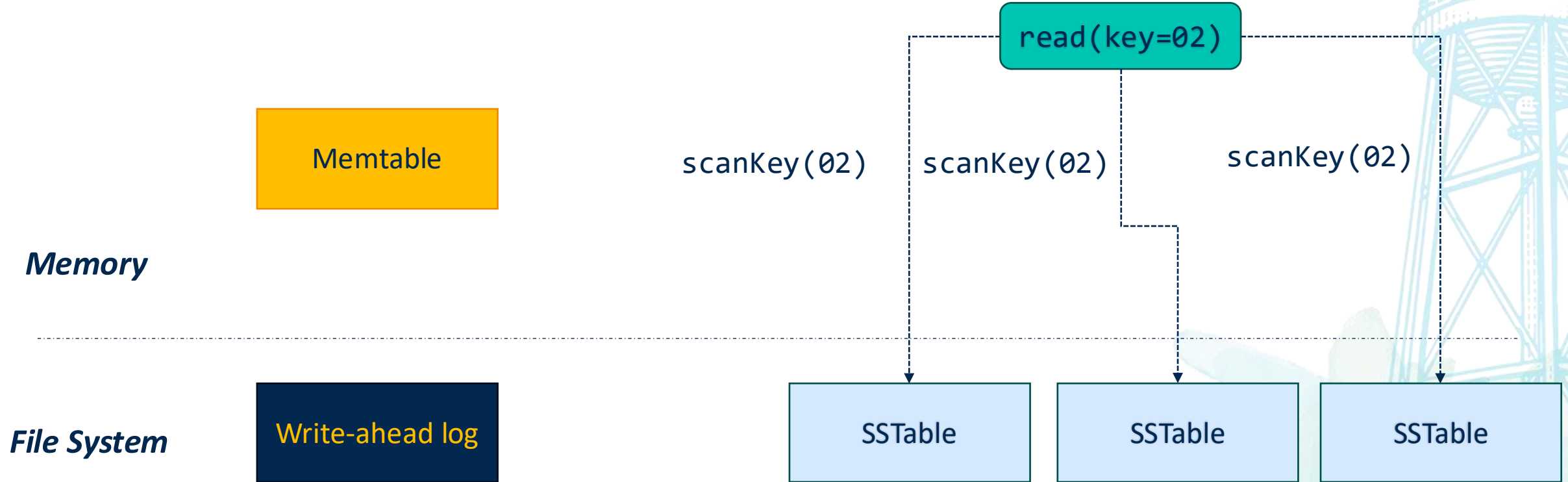
- Check if the key is in the ***in-memory*** memtable
  - Typically,  $O(\log(n))$  complexity
- If not, scan each of the ***on-disk*** SSTables has the key
  - Typically,  $O(n)$  complexity
- Generally, reads are slower than writes



# LSM tree read operation

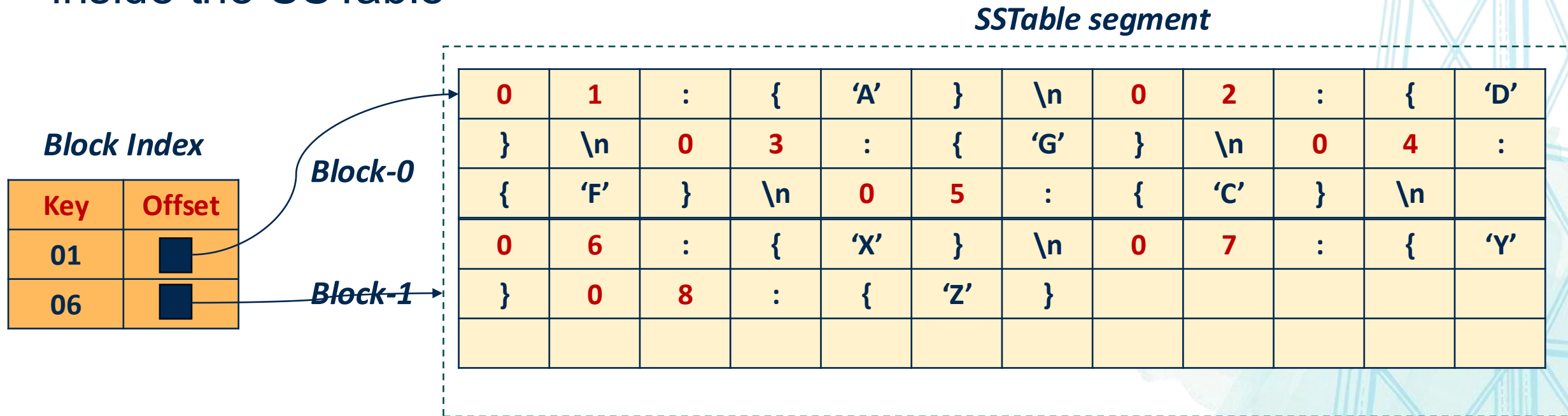


# LSM tree read operation



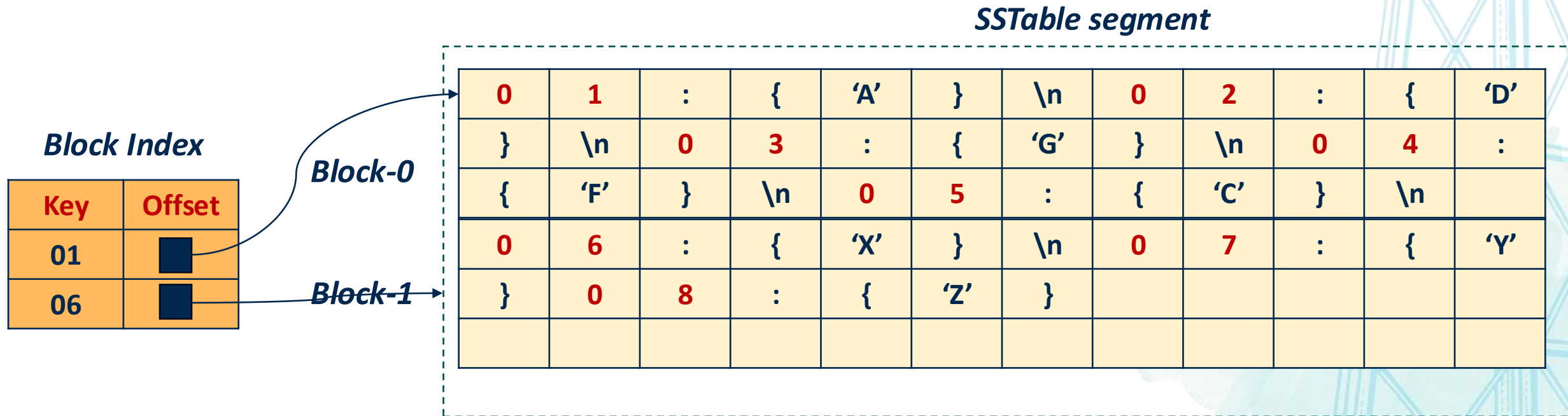
# Block index for read optimization

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



# Block index 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

read(key=02)

Memtable

*Memory*

*File System*

Write-ahead log

SSTable

Block index

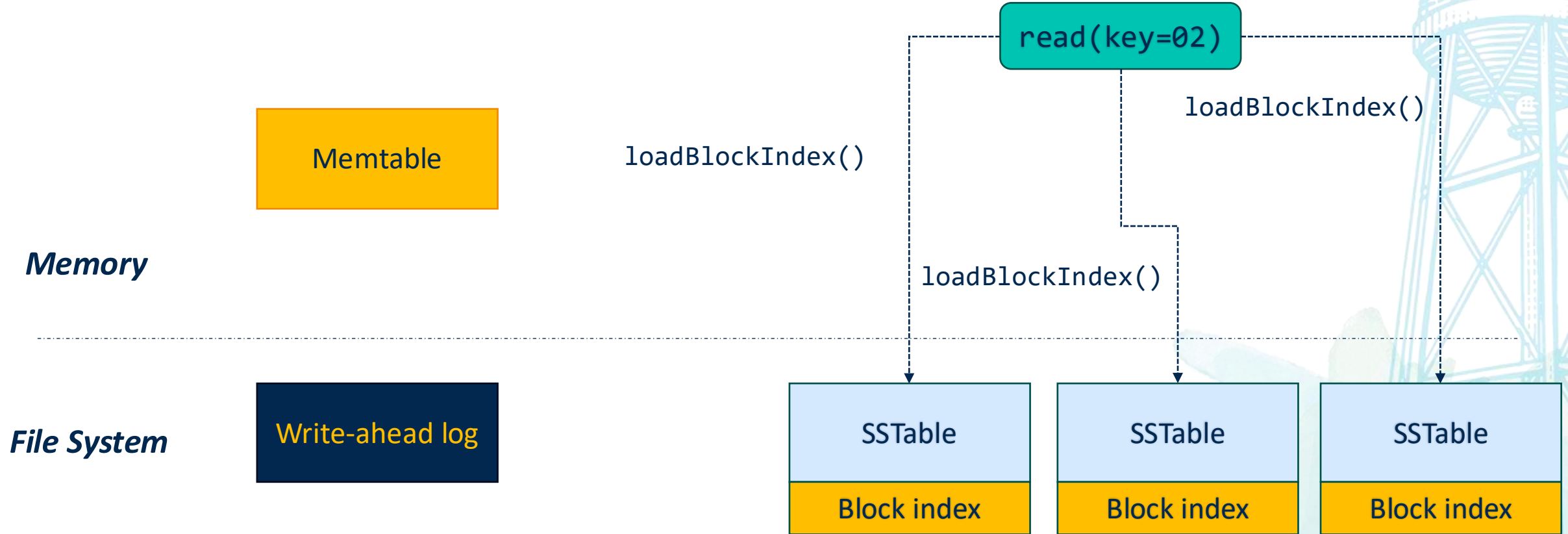
SSTable

Block index

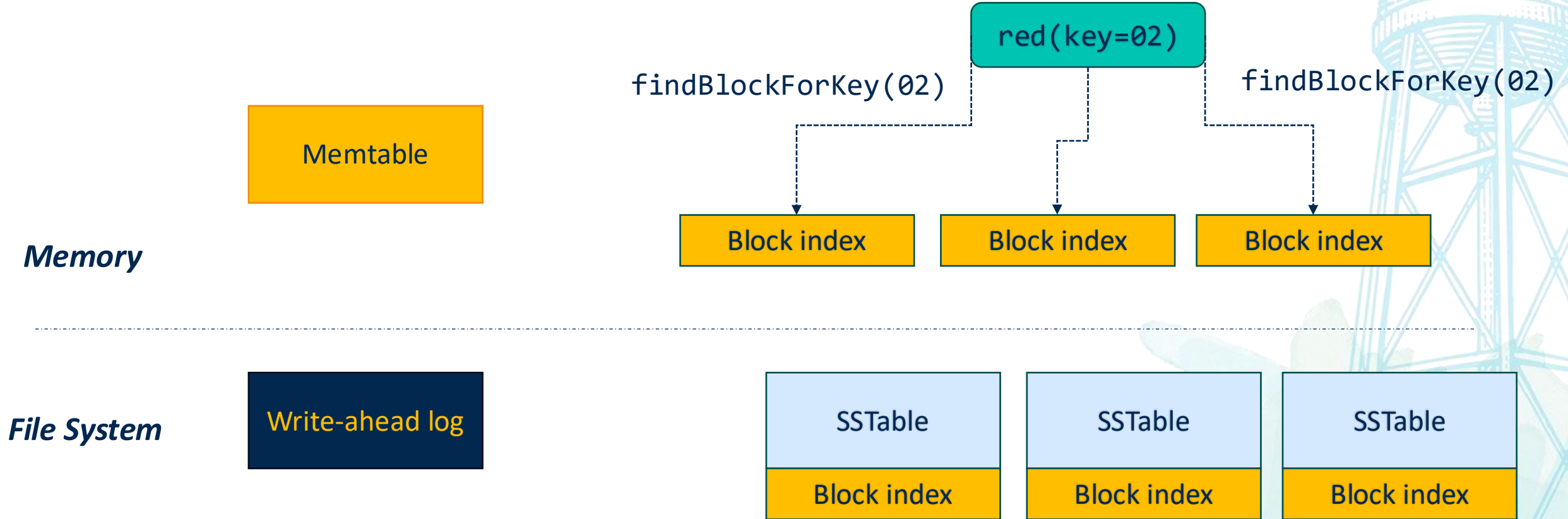
SSTable

Block index

# LSM tree read path with block index

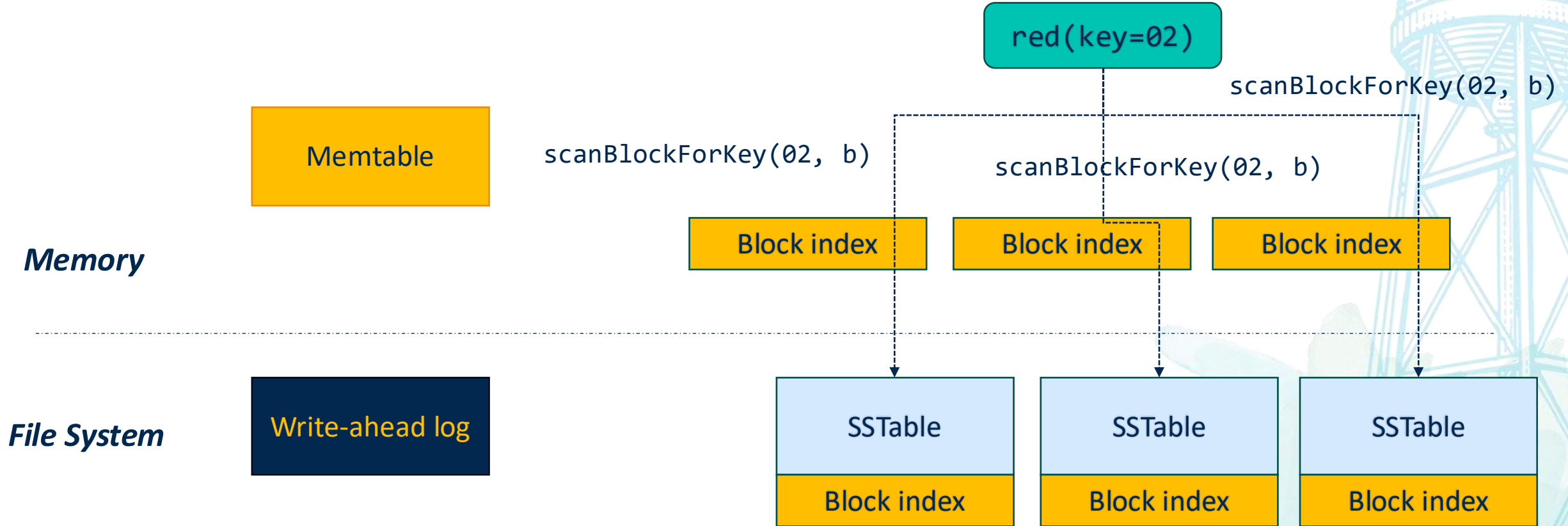


# LSM tree read path with block index



*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)

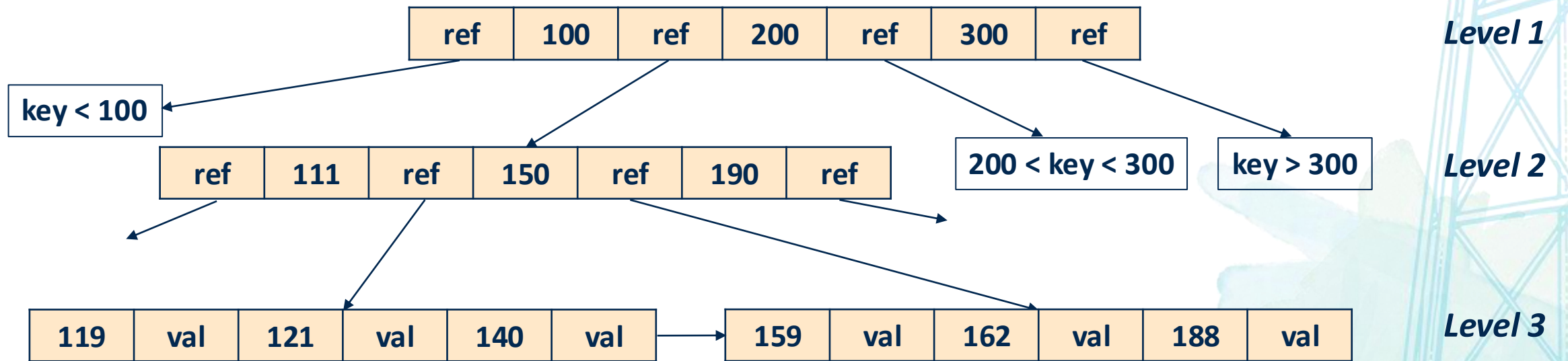
- Used to implement primary key indexes in traditional relational databases
- On-disk data structure that keeps key-value pairs sorted by key
- Supports random accesses
- Writes are performed directly on this on-disk sorted tree, unlike in LSM trees which operated on the in-memory Memtable

# 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

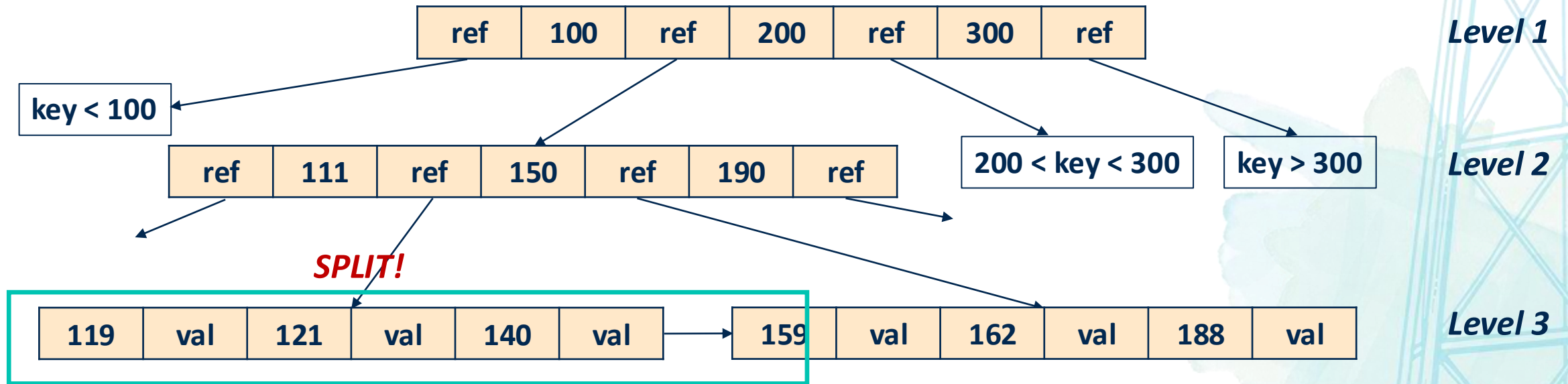
- Traverse the tree following the branches based on the key value
  - Start at the root
  - At each node: search the sorted keys (e.g., binary search) to find the correct child pointer
  - Follow the pointer to the next node and repeat until you reach the leaf
- Complexity is  $O(\log(n))$

# 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 is 3
- insert(152)



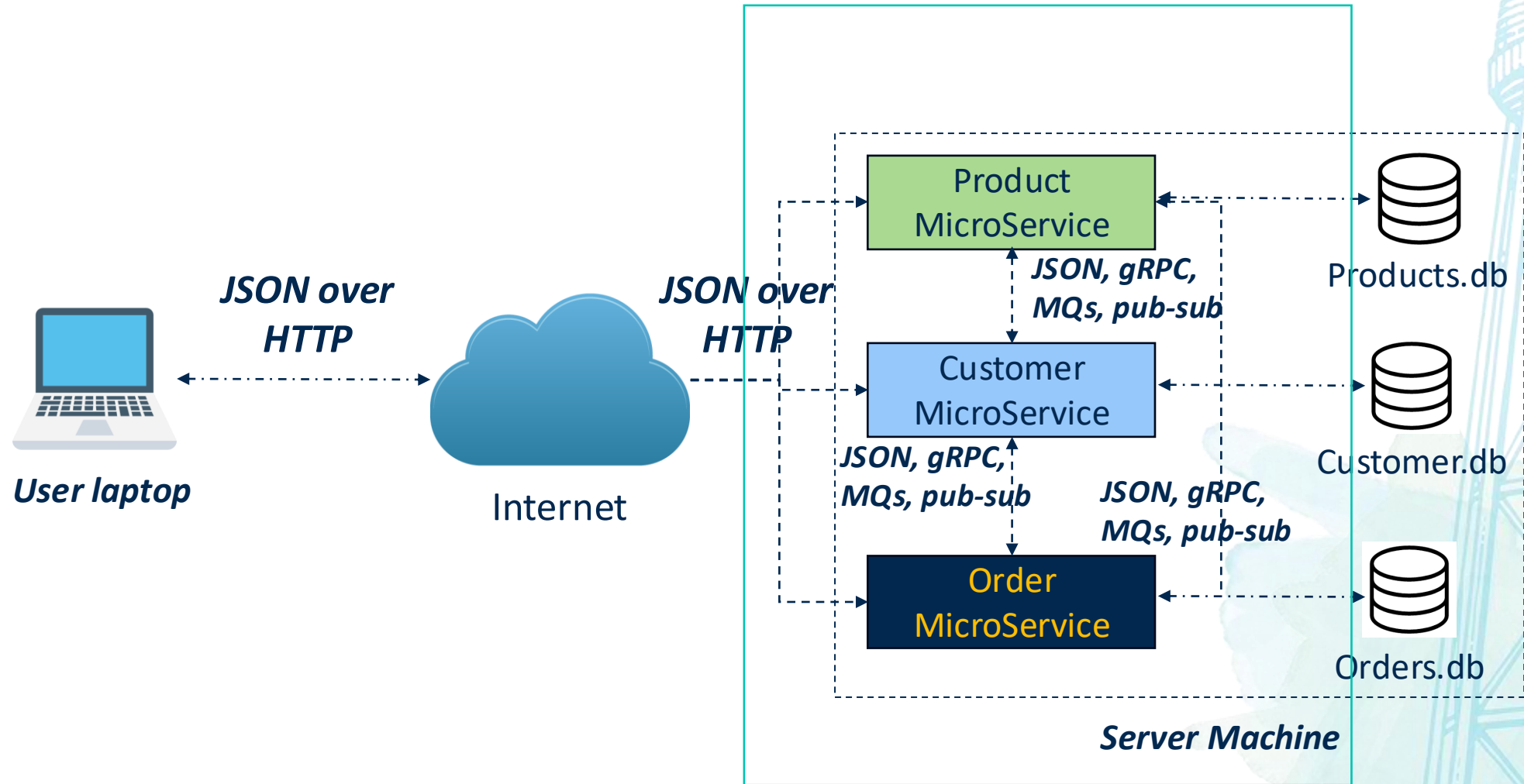
# LSM tree vs B+ tree

- 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

# Communication styles



# Intra-service communication



# Communication choice dimensions

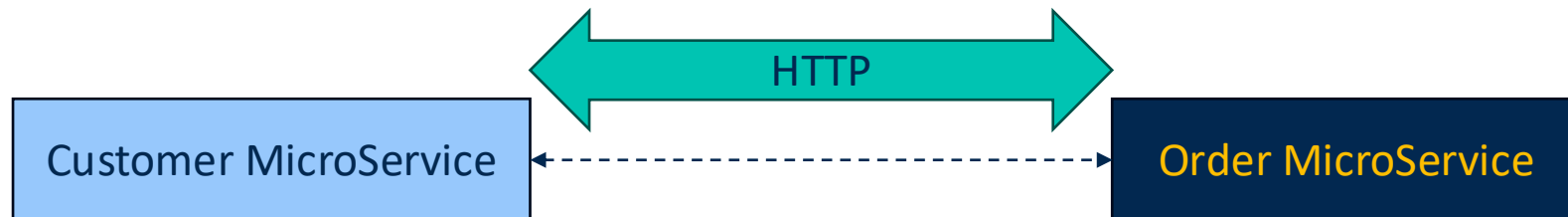
- Synchronous request/response (RPC/HTTP)
- Asynchronous request/response (message queues - RabbitMQ)
- Event-driven publish/subscribe models (Kafka – next module)

# 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

- Representational State Transfer (REST) is an architectural style for web services
- 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());
```

# 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



# Remote procedure call (RPC)

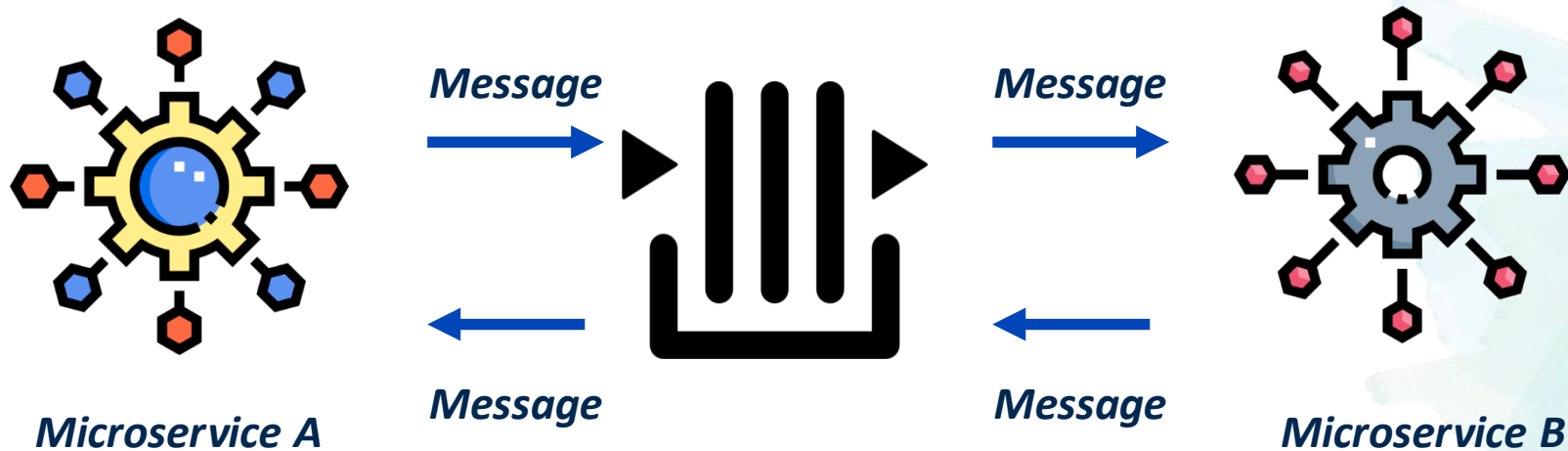
- Library/framework gives the illusion that the other microservice is running locally
- Protocol that allows a program to execute a procedure on a remote server as if it were local

# RPC key features

- Language agnostic: the RPC itself does not depend on the service language
- Abstracts network details
- Typically, over HTTP

# Message queuing (MQ)

- Asynchronous communication model
  - Messages sent to a queue and processed by consumers independently of the producer
- Stronger decoupling



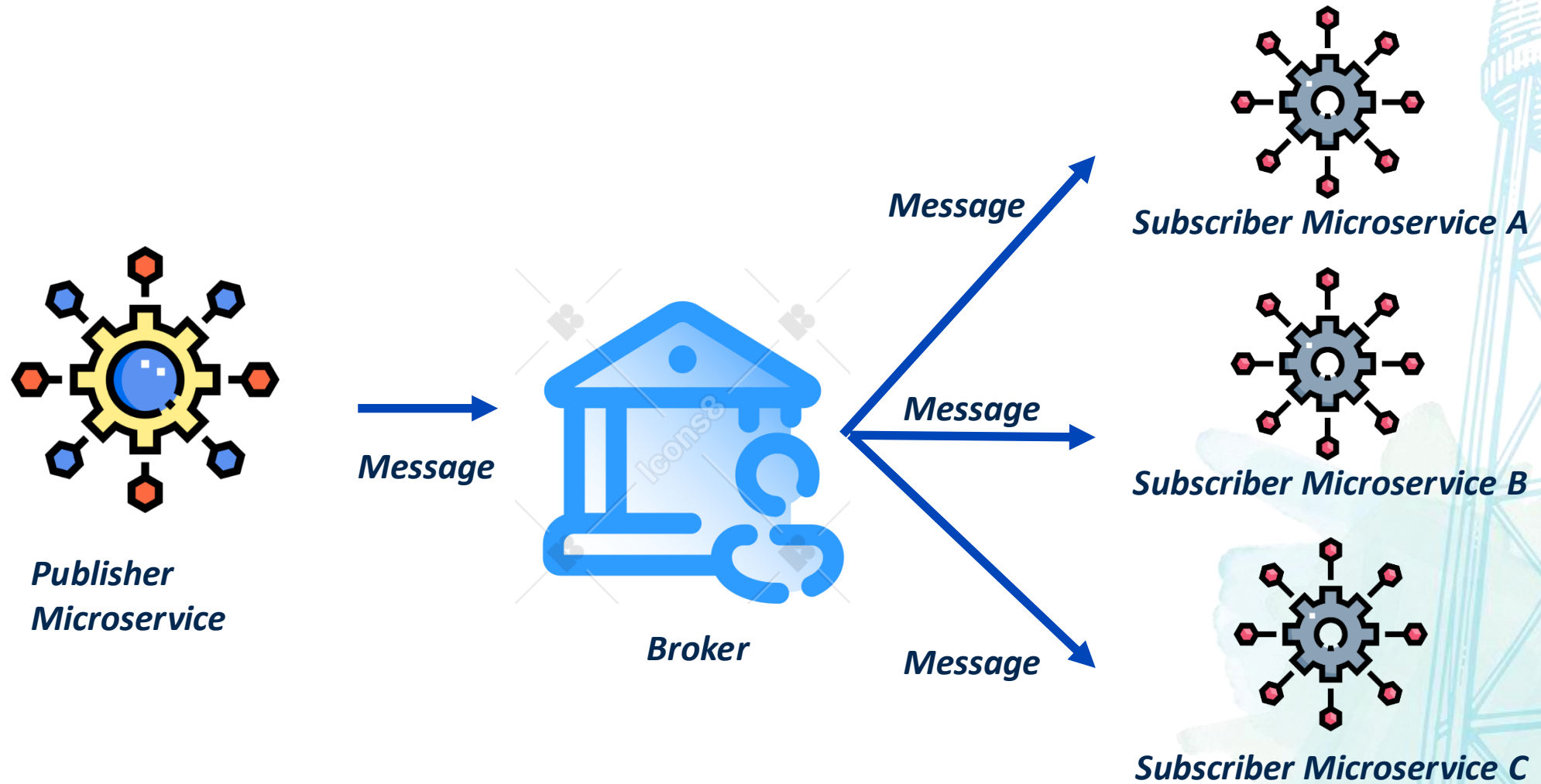
# Open source and paid MQ services



# 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
- Typically, messages are persistent at the broker and must be explicitly deleted
- Same frameworks often can act as both MQ or Pub-Sub depending on configuration

# Pub-sub architecture



# Pub-sub frameworks



Google Cloud  
Pub/Sub

**Redis**



**Pub**

**Sub**