

# **BASE**

## **AN ACID ALTERNATIVE**

**Group 1**

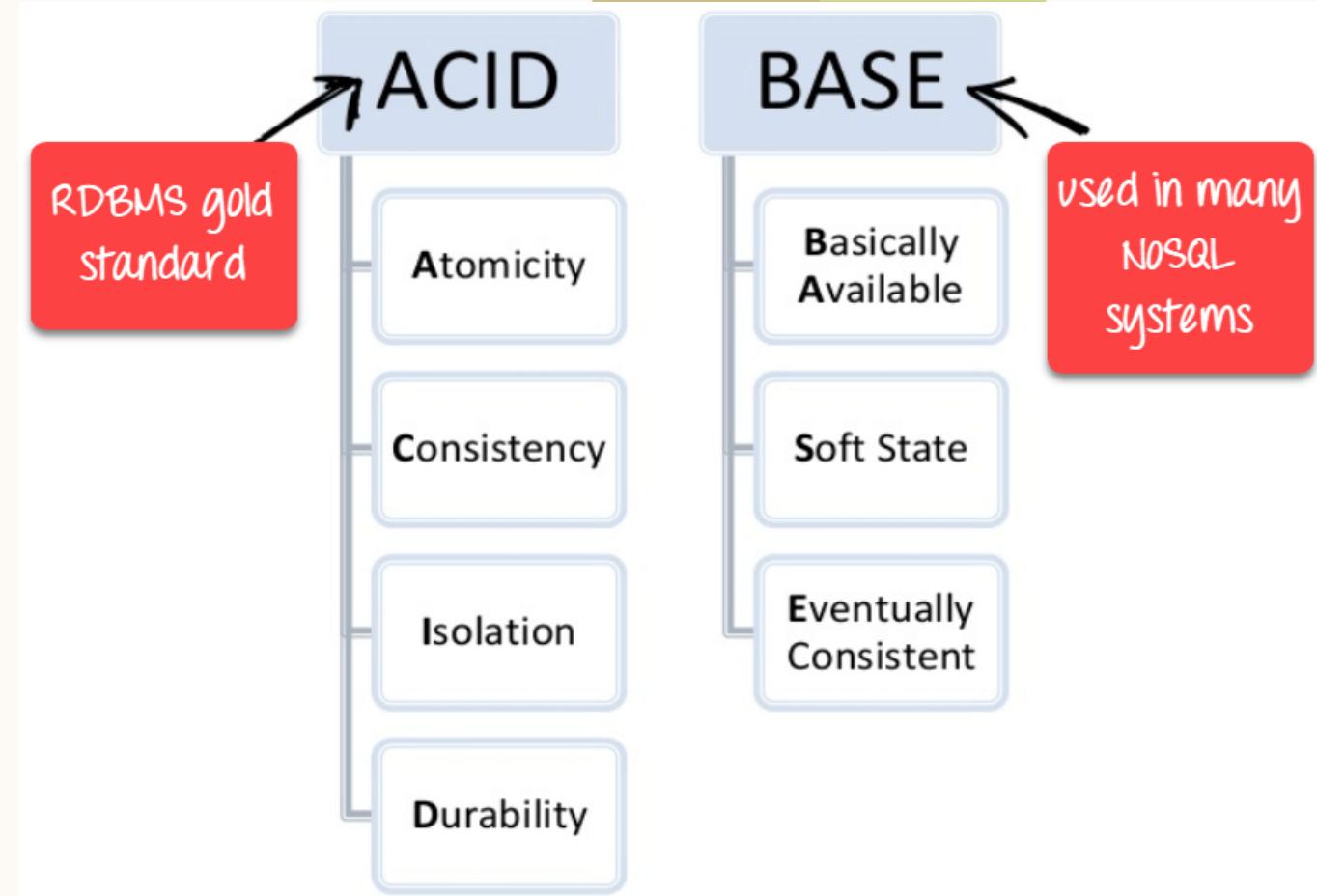
# ACID VS BASE

2

Not this.



But this.



# AGENDA

Introduction

CAP Theorem

ACID vs BASE

BASE Model and its implementation

Toy Example

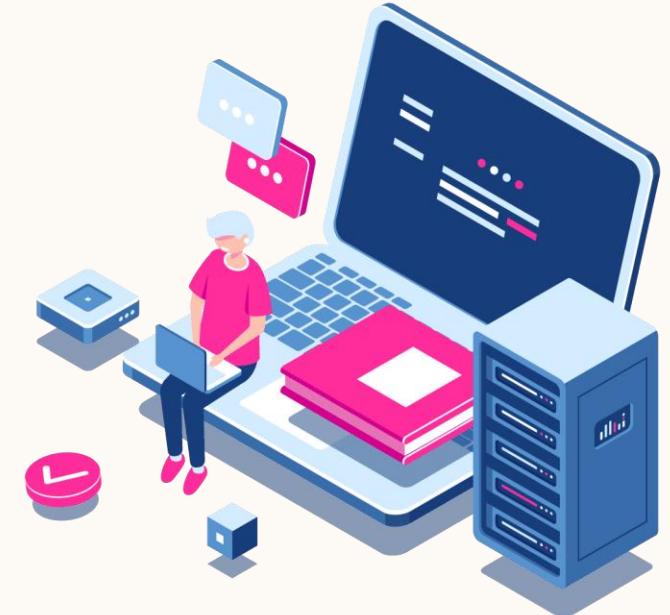
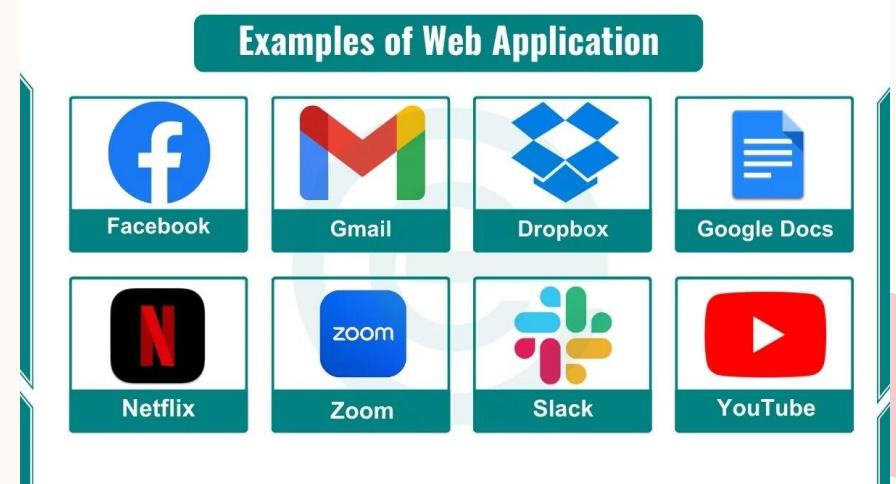
Software State, Eventual Consistency and Event-Driven Architecture

Web applications have grown in popularity over the past decade.

## SAY YOU ARE BUILDING YOUR WEB APPLICATION



What things you may need to worry about?

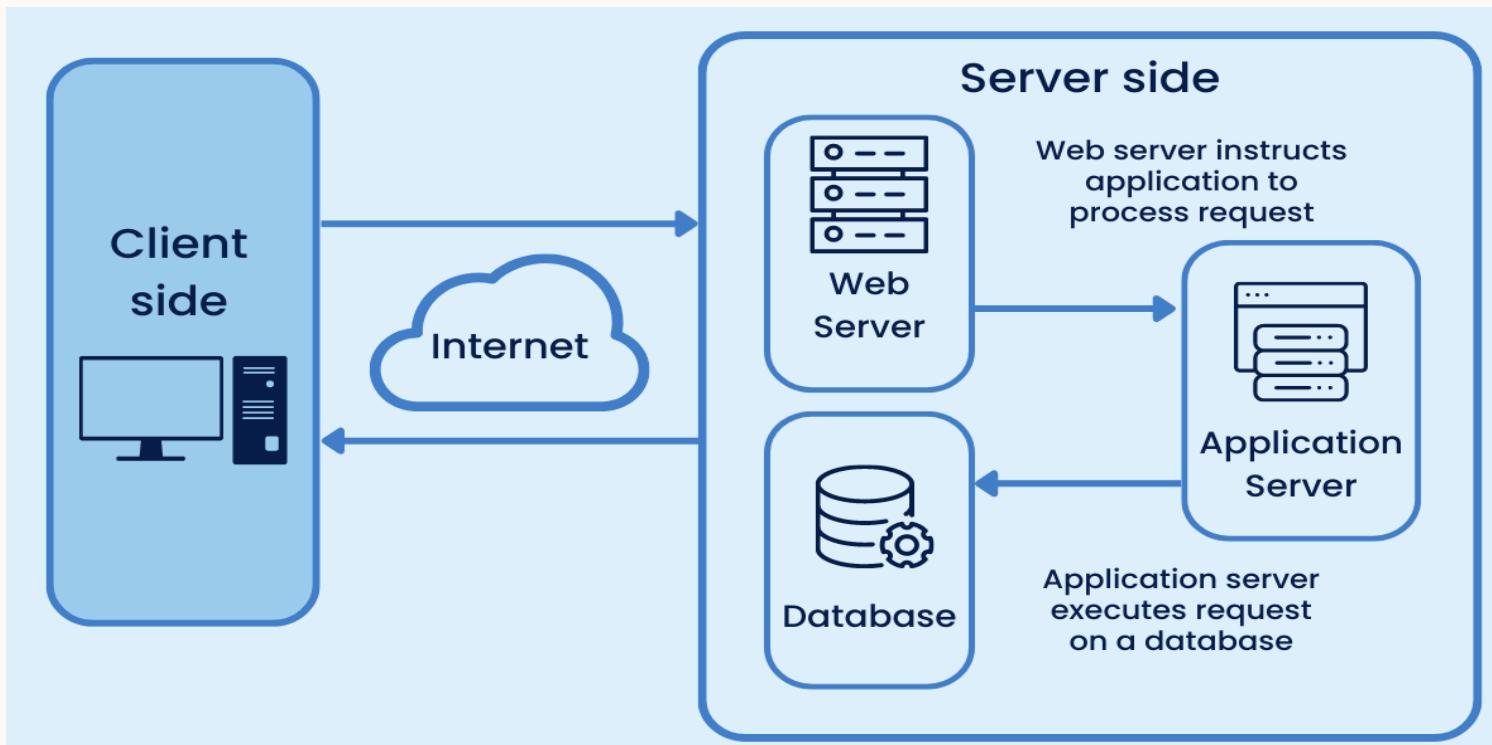


# EVERYTHING AT ONE NODE [CENTRALIZED SYSTEM]

A simple solution

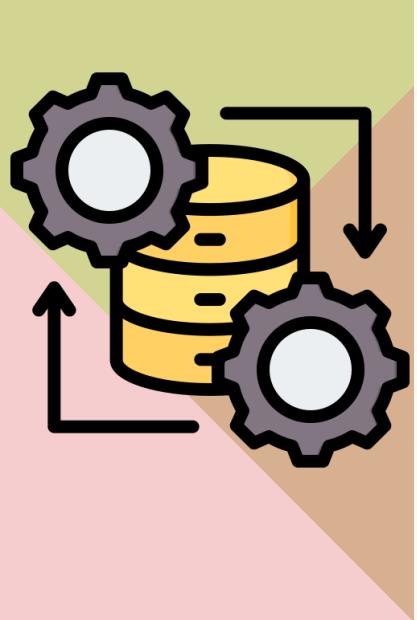
Database service, web server, app server all are at one place (node)

Clients contact one server hosting service (no load balanced) via HTTP over Internet



# YOU WITNESS GROWTH

## GROWING POPULARITY OF YOUR WEB APPLICATIONS



Increasing demand for your application will cause:



higher traffic at your end



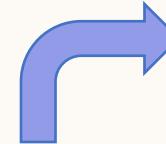
surging data processing requirements for your app

# YOU WITNESS GROWTH

## IMPORTANCE OF SCALABILITY FOR YOUR WEB APPLICATION

You need to scale because of:

- Rapid transactional growth at your side
- Performance bottlenecks issues



# 404

Not Found

The resource requested could not be found on this server!

## DATA STORAGE AS MAIN BOTTLENECK



- read/write performance issues
- traditional databases struggle with high concurrency and large datasets



# SO YOU WANT TO SCALE

## YOUR PAIN BEGINS

Many Challenges in Scaling Data Storage/Web Server

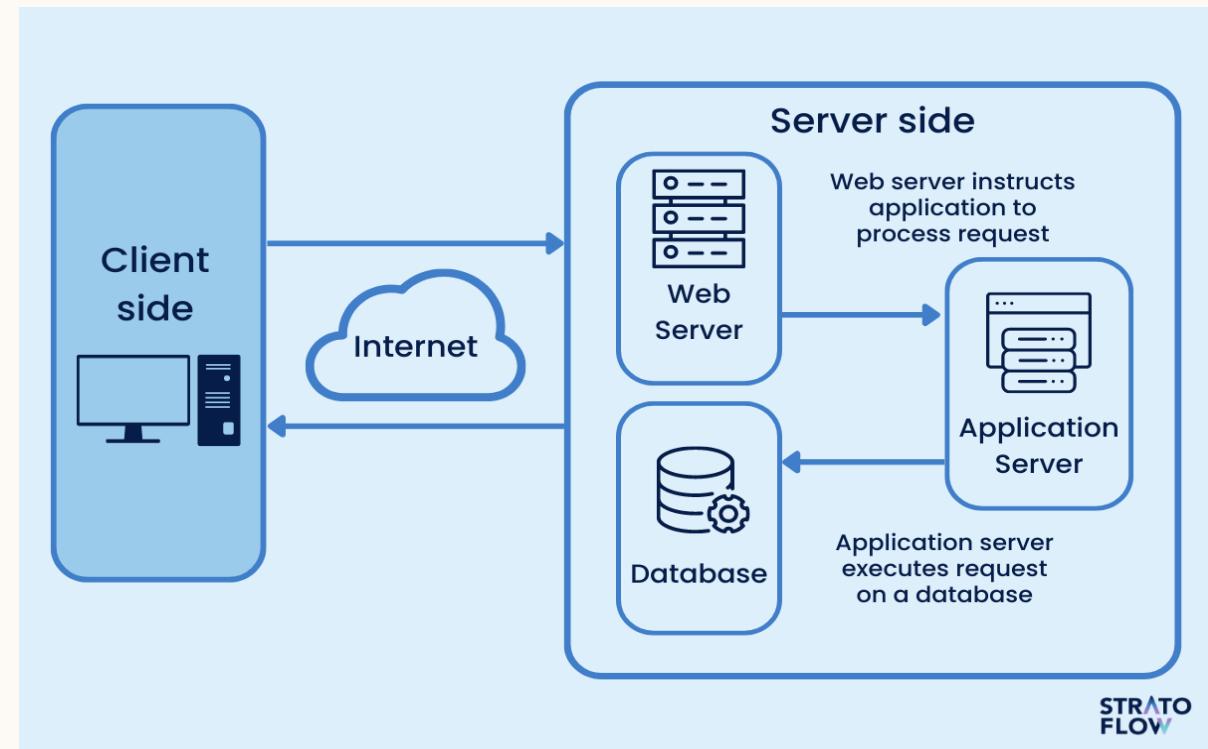
YOU WANT TO IMPROVE:

- Latency:
- Throughput:

YOU GET TO TACKLE:

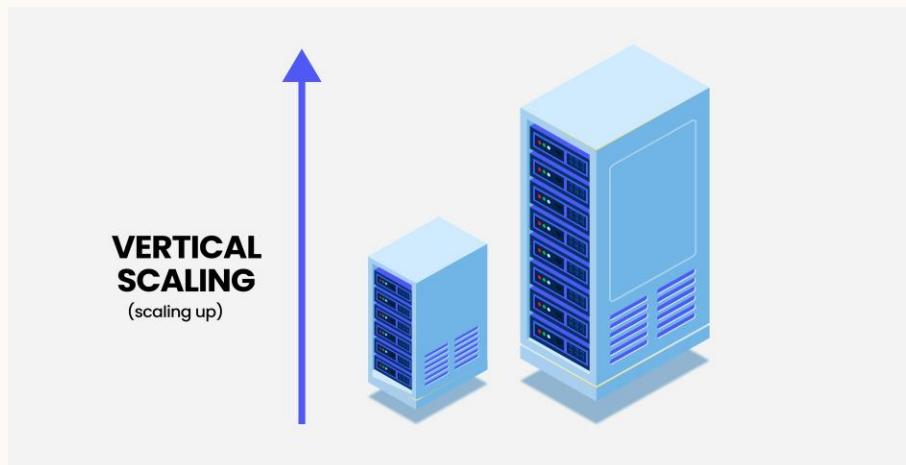
- Consistency:
- Scalability:

Simple Solution  
needs to be scaled



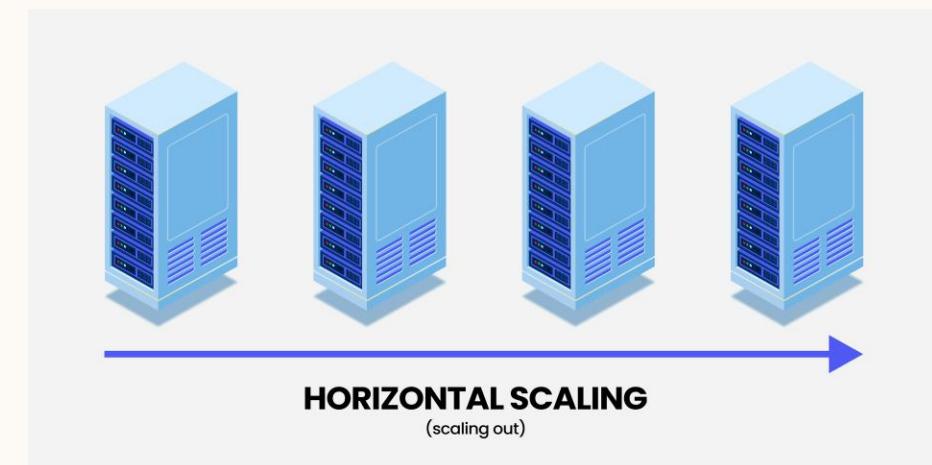
## VERTICAL SCALING (SCALING UP)

- Increases system capacity by upgrading the existing server's hardware (CPU, RAM, disk)
- A single machine is made more powerful to handle increased load.
- Suitable for applications requiring high performance on a single instance.



## HORIZONTAL SCALING (SCALING OUT)

- Increases system capacity by adding more machines (servers) to the infrastructure.
- Traffic is distributed across multiple servers using load balancers.
- Each new server handles a portion of the requests, reducing the load on individual machines.

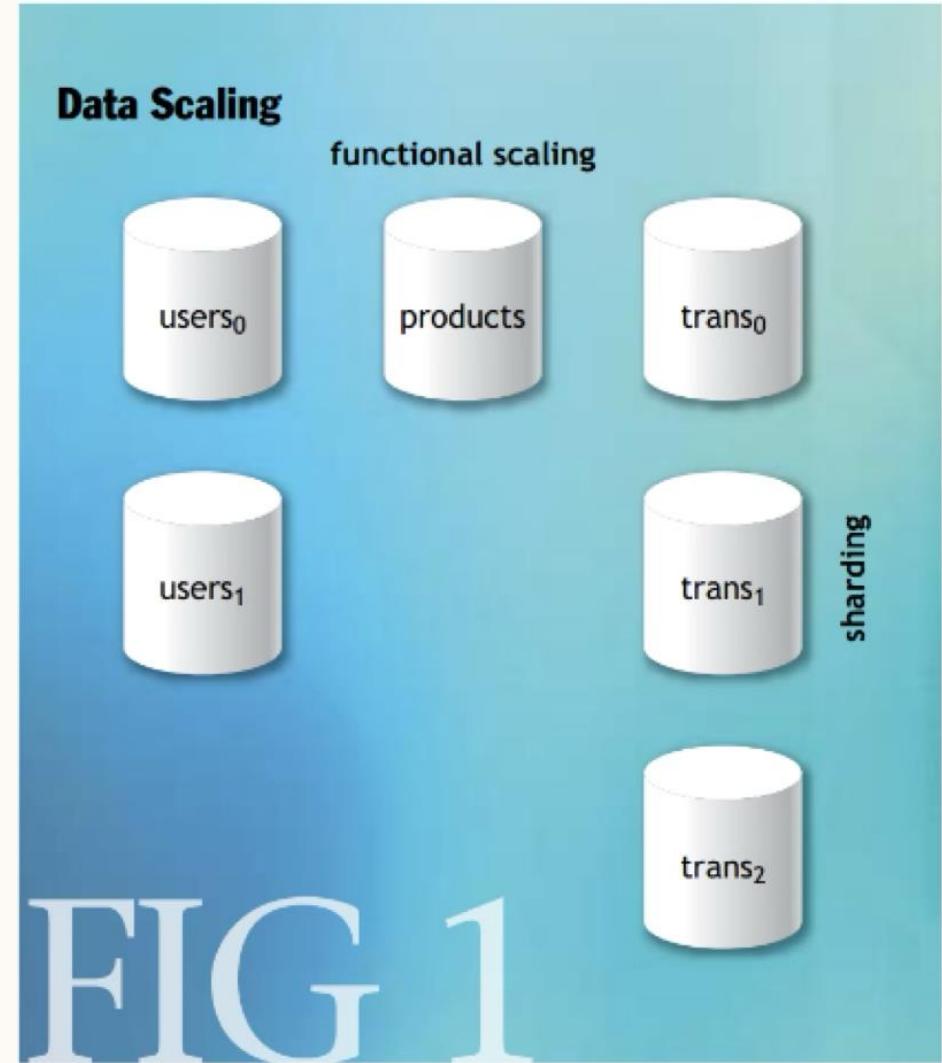


# HORIZONTAL SCALING

Horizontal data scaling can be performed along two vectors:

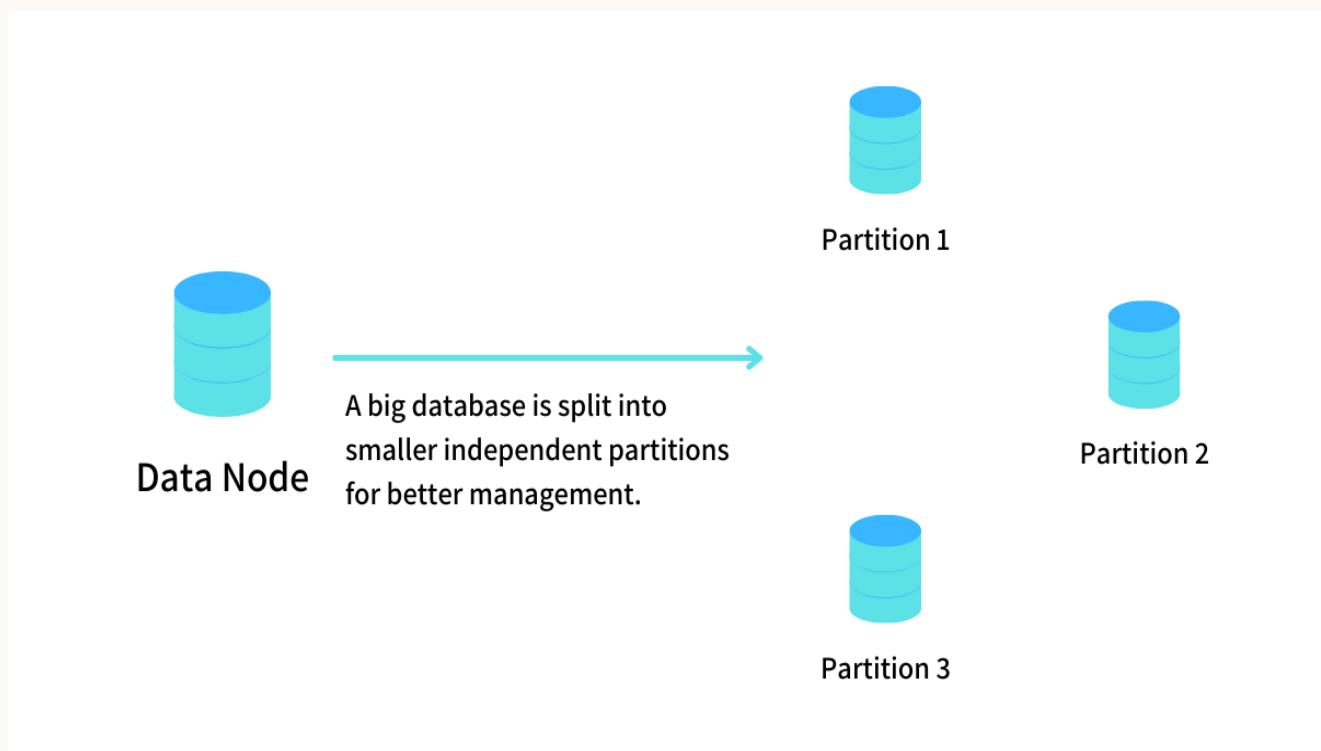
1. **Functional scaling**- Grouping data by function and spreading functional group across databases.
2. **Sharding**- Splitting data within functional areas across multiple databases.

We can also apply both approaches at once.



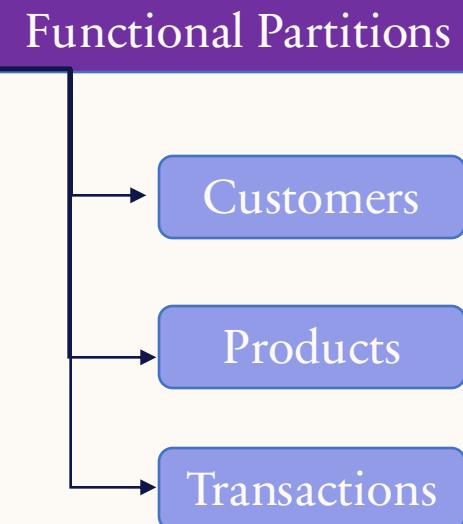
# FUNCTIONAL PARTITIONING

- Functional partitioning is important for achieving high degrees of scalability.  
(like microservices architecture)
- Any good database architecture decomposes the schema into tables grouped by functionality.



# FUNCTIONAL PARTITIONING (1)

## AN EXAMPLE



Product master data			
Product ID	Product name	Product description	Price
1982SP	Lamp	Metal floor-standing lamp	56.95
454YH	Chair	Oak dining chair	70.99

Customer master data		
Customer ID	Customer name	Customer address
67	J. Smith	56 Quy Road, Chester, UK
68	R. Hurst	14 Back Lane, Norwich, UK

Transactions				
Sale ID	Product ID	Customer ID	Actual Sale price	Sale time
002	1982SP	67	56.95	1/3/13 15.34.12
003	454YH	67	65.99	1/3/13 15.34.25
004	454YH	68	70.99	4/3/13 12.05.43

# FUNCTIONAL PARTITIONING (2)

## ENSURING CONSISTENCY:

- Foreign keys to maintain relationships between tables.
- For constraints to be applied, the tables must reside on a single database server, otherwise we have to use 2PC.

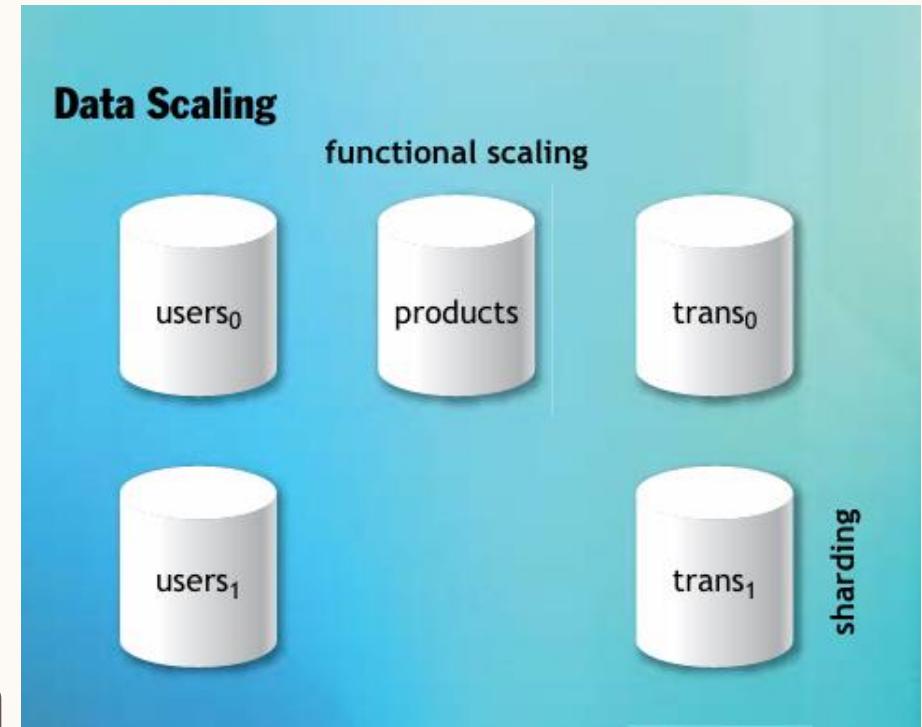
## ENSURING SCALABILITY:

- Moves functional groups of data onto separate database servers.
- Reduces load on a single database.
- Enables independent scaling of different functional areas (e.g., scale user data separately from transactions).

# FUNCTIONAL PARTITIONING (3) CHALLENGES?

- Moving Data Constraints to the Application
- Removes **database-enforced constraints** for flexibility in scaling.
- Requires application logic to enforce data consistency.
- Introduces new challenges, such as handling **distributed transactions** and **eventual consistency** or 2PC.

FUNCTIONAL PARTITIONING IS NOT SHARDING



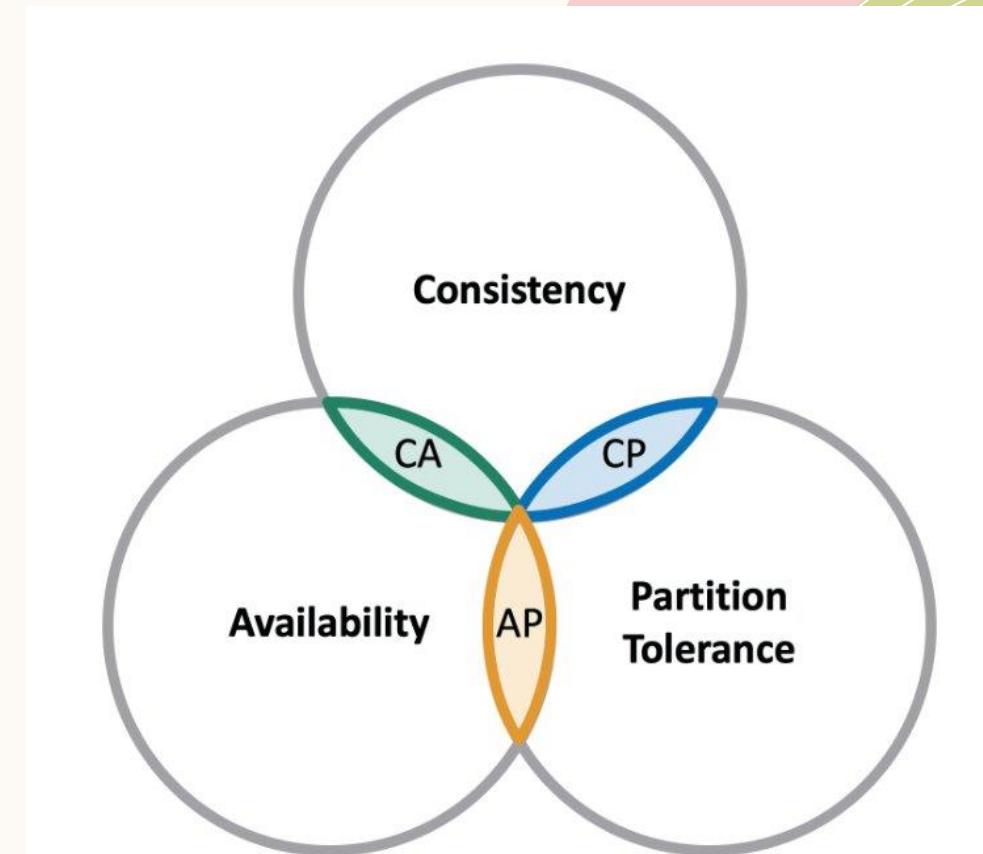
# CAP THEOREM (1)

*proposed by: Eric Brewer (professor at the University of California, Berkeley)*

He made the *conjecture* that Web services cannot ensure all three of the following properties at once

- **Consistency (C)** – Every read receives the most recent write or an error.
- **Availability (A)** – Every request receives a response(not necessarily the most recent).
- **Partition Tolerance (P)** – The system continues to function despite network failures (partitions).

Since network partitions **always** happen in distributed systems, you can only choose between **Consistency (C)** or **Availability (A)** under partition failure.



# CAP THEOREM (2)

## Trade-offs in CAP Theorem

- CP → Low Availability.  
Example: HBase, MongoDB (strong consistency)
- AP → Delayed/no Consistency  
**May return stale (outdated) data.**  
Example: Cassandra, DynamoDB, CouchDB
- CA → No Partition Tolerance  
**Not realistic for large distributed systems**  
Example: Single-node databases like RDBMS (PostgreSQL, MySQL in single-node mode)

Distributed systems **must tolerate partitions (P)**, so they must choose between **C or A**.

**AP systems** prioritize uptime and fast responses.  
**CP systems** prioritize correctness.

# ACID

ACID database transactions greatly simplify the job of the application developer.

As signified by the acronym,  
**ACID** transactions provide the following guarantees:

**Atomicity.** All of the operations in the transaction will complete, or none will.

**Consistency.** The database will be in a consistent state when the transaction begins and ends.

**Isolation.** The transaction will behave as if it is the only operation being performed upon the database.

**Durability.** Upon completion of the transaction, the operation will not be reversed.

# **ACID**

## **COMMITTING IN A SINGLE DB**

**It is easy**

Steps:

- Transaction begins (`BEGIN TRANSACTION`).
- SQL operations execute (e.g., `INSERT`, `UPDATE`, `DELETE`).
- Changes are held in memory (not yet permanent).
- `COMMIT` command is issued.
- The database writes changes to disk and updates the transaction log.
- The transaction is marked as complete.

**Easier Recovery and Consistency:**

- If a failure occurs, a single database can quickly rollback changes because all data modifications are in the same transaction log.

# **WE START WITH A TOY EXAMPLE**

We start with most simple  
ACID (& 2PC)  
implementation.



Present the problems.



Solving using BASE.

# ACID TRANSACTION

## TOY EXAMPLE

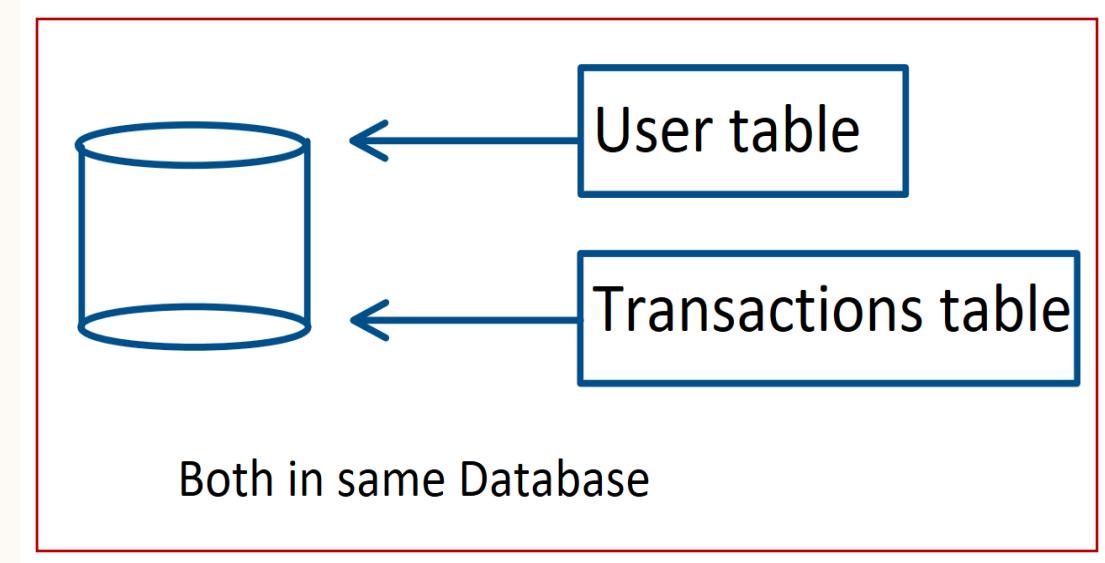
### Example Schema in one node:

Initially all our data tables are in one database which is hosted on a single node.

We can perform operations on both tables in one single transaction.

#### Advantages:

- Simplicity
- DBMS handles ACID for us



#### Sample Schema

user	transaction
id	xid
name	seller_id
amt_sold	buyer_id
amt_bought	amount

# ACID TRANSACTION

## TOY EXAMPLE

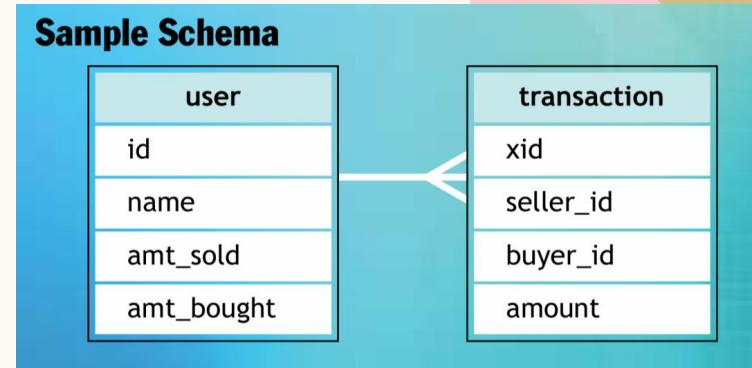
### Example in one node:

We know this will work because on a single database on a single node, transactions will ensure consistency, and durability.

No worry about any such issues!

#### Improvements:

- The two functional groups can be separated
- No need to have one long transaction
- Can do separately for better concurrency and requests handling



```

Begin transaction
Insert into transaction(xid, seller_id, buyer_id, amount);
Update user set amt_sold=amt_sold+$amount where id=$seller_id;
Update user set amt_bought=amount_bought+$amount where id=$buyer_id;
End transaction
  
```

# FIG 3

# ACID TRANSACTION

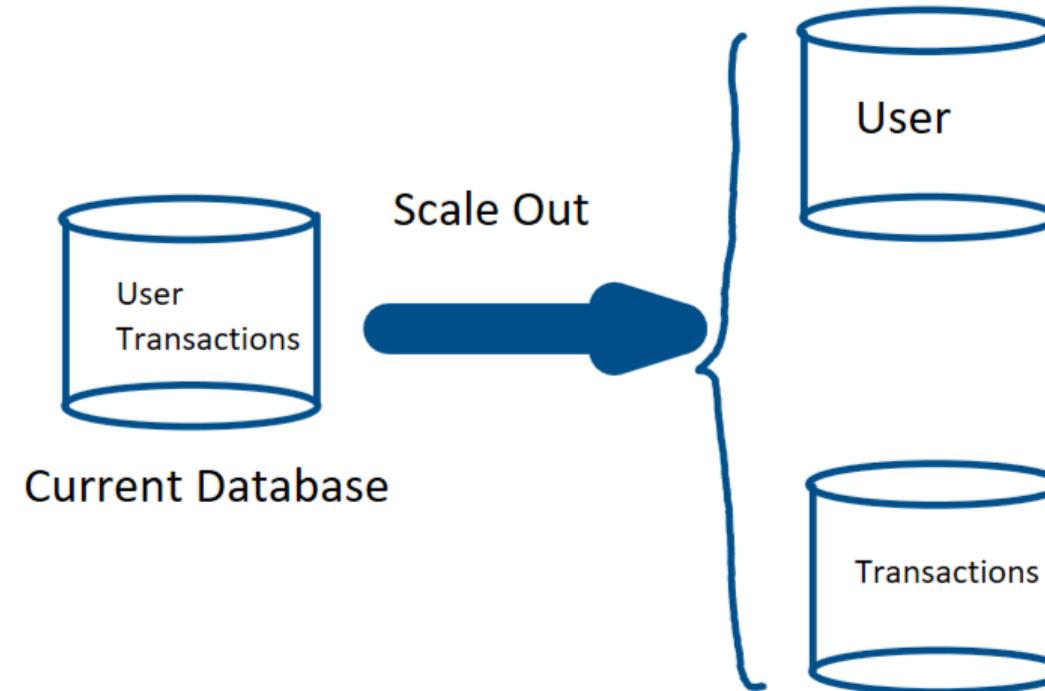
## SAY WE WANT TO SCALE

Following the microservices structure we want to scale out:

We want to separate the services for users and transactions. So, we also have to separate their databases.

Separate service can be on separate nodes, so their databases are also on different nodes.

Adds throughput and lowers latency.

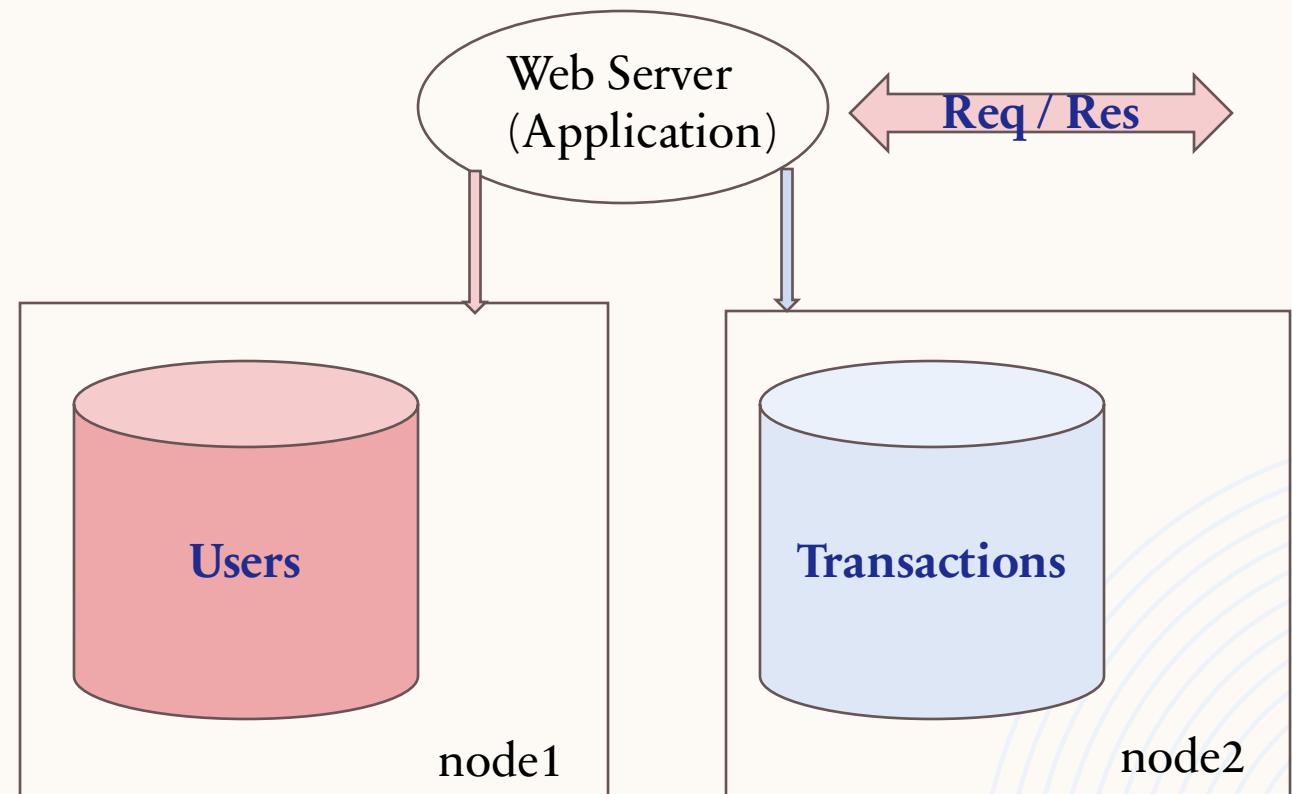
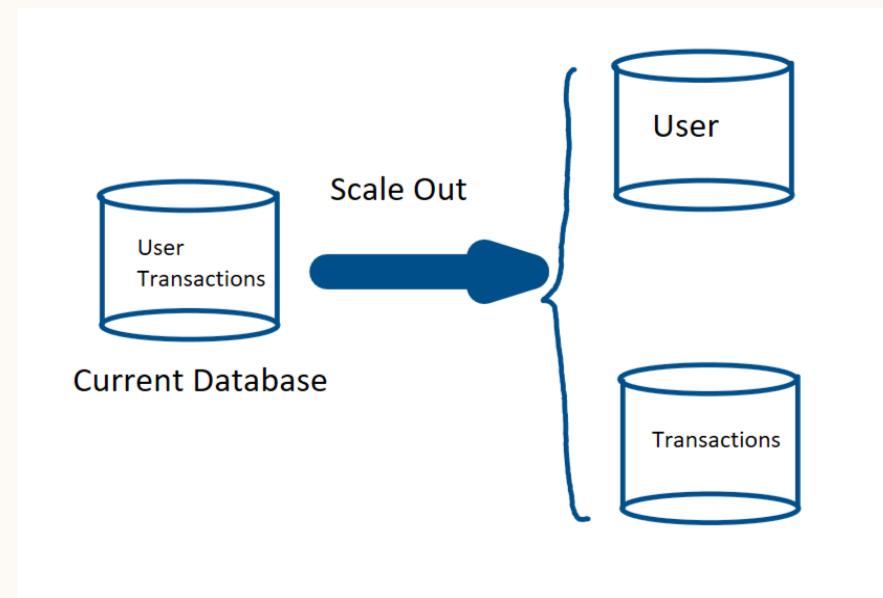


We scale out to two different databases on two different nodes.

# ACID TRANSACTION

- Our schema contains two separate tables 'user' and 'transaction'.
- We separate these into two functional parts.
- And physically keep into two different databases on two different nodes.

Sample Schema	
user	transaction
id	xid
name	seller_id
amt_sold	buyer_id
amt_bought	amount



# ACID TRANSACTION

## After scaling:

We do two separate transactions to communicate updates to two different databases.

```
Begin transaction
```

```
Insert into transaction(id, seller_id, buyer_id, amount);
```

```
End transaction
```

Transaction 1  
(on node 1)

```
Begin transaction
```

```
Update user set amt_sold=amt_sold+$amount where id=$seller_id;
```

```
Update user set amt_bought=amount_bought+$amount  
where id=$buyer_id;
```

Transaction 2  
(on node 2)

```
End transaction
```

# ACID TRANSACTIONS

25

## What can go wrong?

Txn#1 Fails, Txn#2 Goes

No transaction record persisted, but users have seen their amounts getting updated ➔ Inconsistency

Txn#2 Fails, Txn#1 Goes

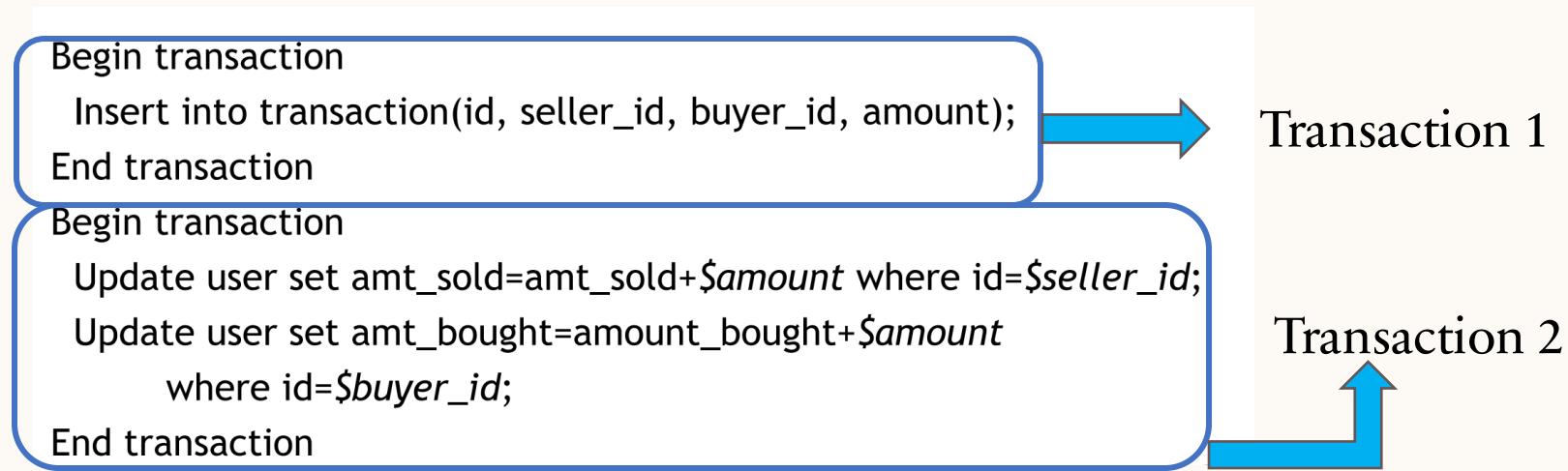
A transaction record persisted, but users have not seen their amounts getting updated ➔ Inconsistency

Txn#1 Fails, Txn#2 Fails

All ok (NONE happened)

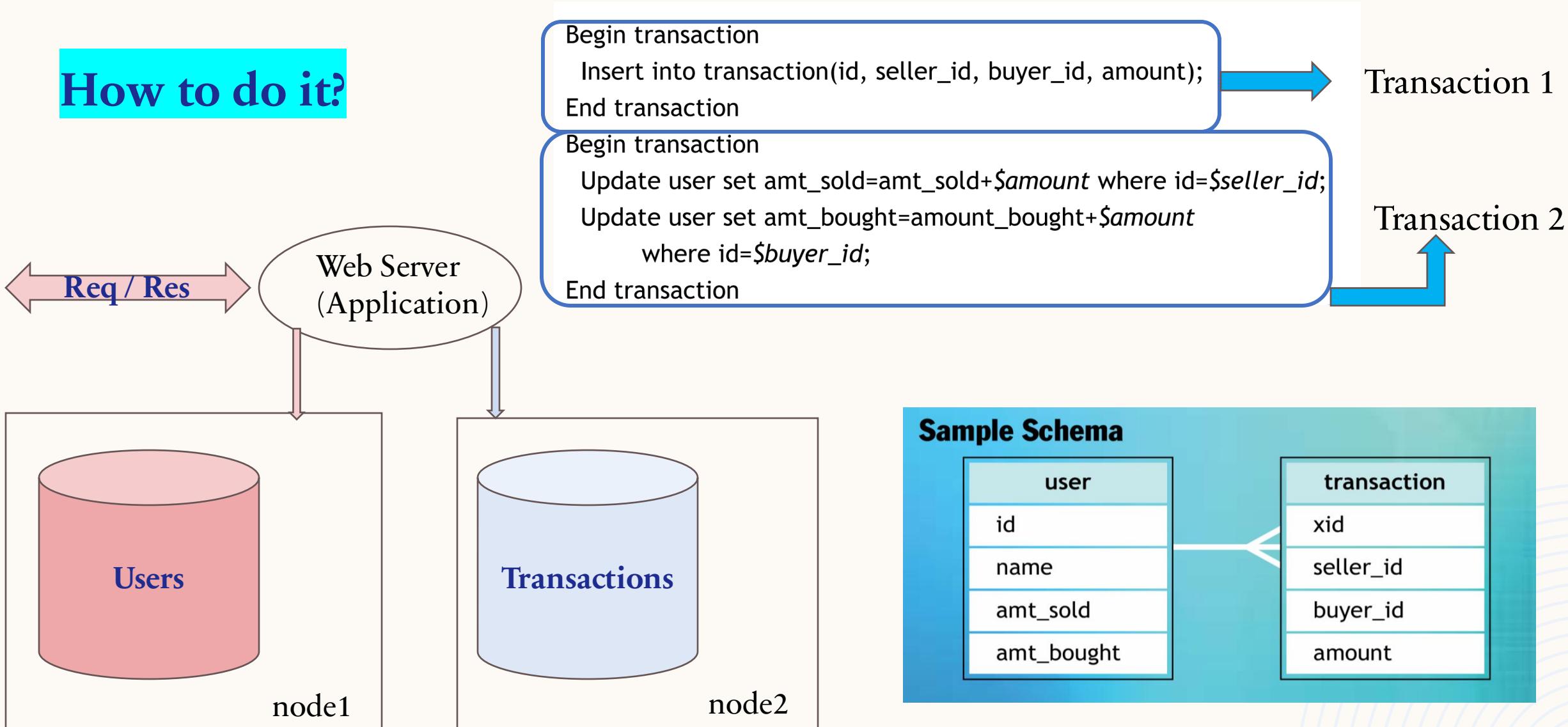
Txn#1 Goes, Txn#2 Goes

All ok (ALL happened)



# ACID TRANSACTIONS

26



# WE NEED 2PC PROTOCOL

## FOR DATABASES IN DISTRIBUTED ENVIRONMENT

- A distributed database requires extra steps:
- (e.g., 2PC, consensus protocols)
- to ensure atomicity across multiple nodes,
- making it more complex and less available reducing overall concurrency.

What is this 2PC?  
In a nutshell

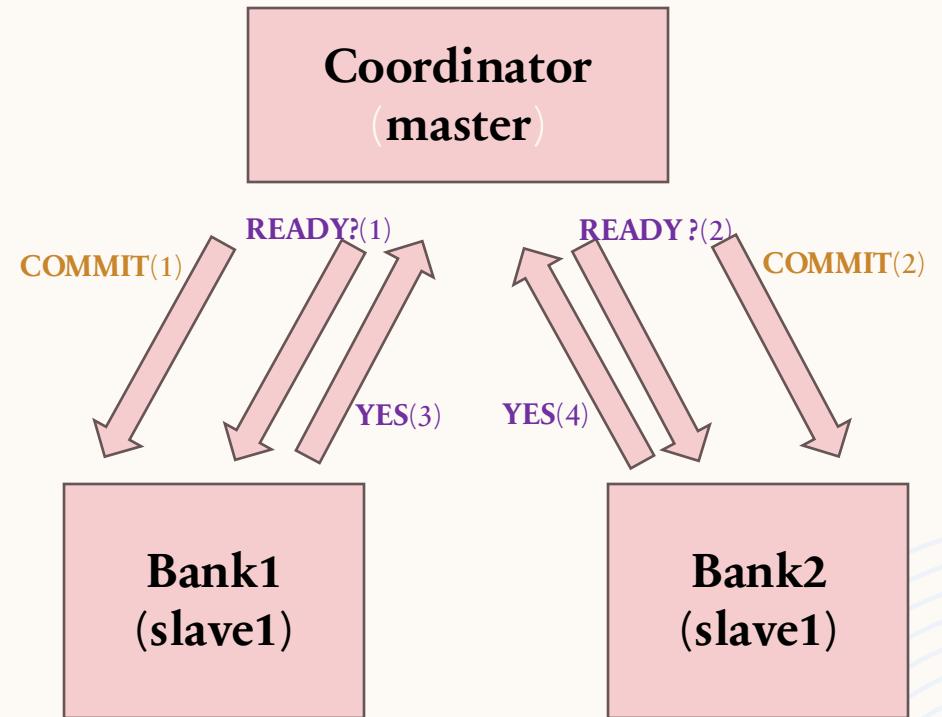
The protocol is broken into two phases:

- **Phase1:-** The transaction coordinator asks each database involved to pre-commit the operation and indicate whether commit is possible.
- **Phase2:-** If all databases agree that the commit can proceed, then transaction coordinator asks each database to commit the data.

# WORKING: 2PC PROTOCOL

## A SIMPLE RUNDOWN

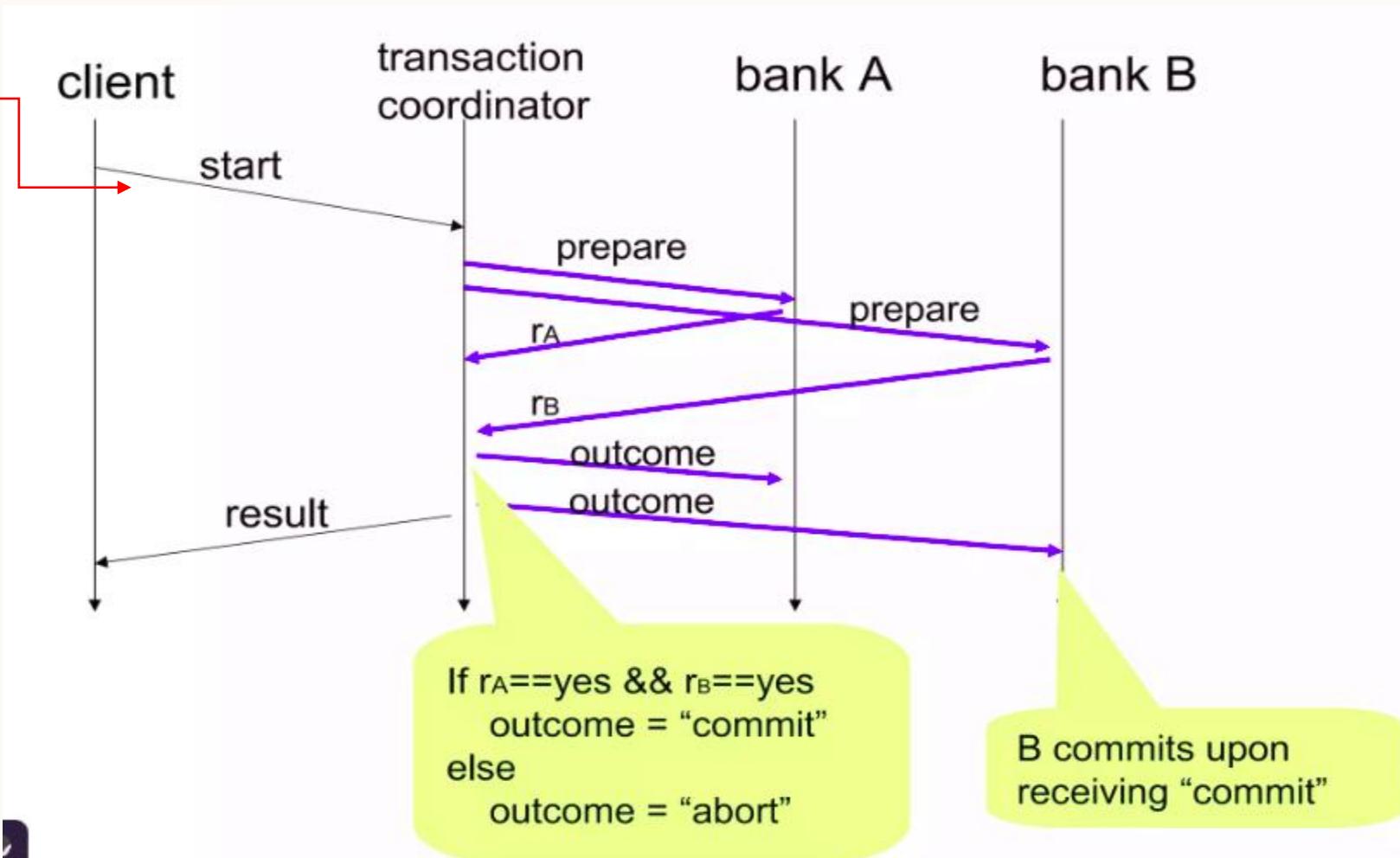
- **PREPARE PHASE:** The master sends message 1 and 2 to both slave nodes asking if they are ready to commit. The nodes reply with yes or no (message 3 and 4). If both the nodes agree, commit can proceed else rollback is done.
- **COMMIT PHASE:** On receiving messages from both slave nodes, master asks them to commit by sending messages 5 and 6.



## WE NEED 2PC PROTOCOL: AN EXAMPLE RUNDOWN(1)

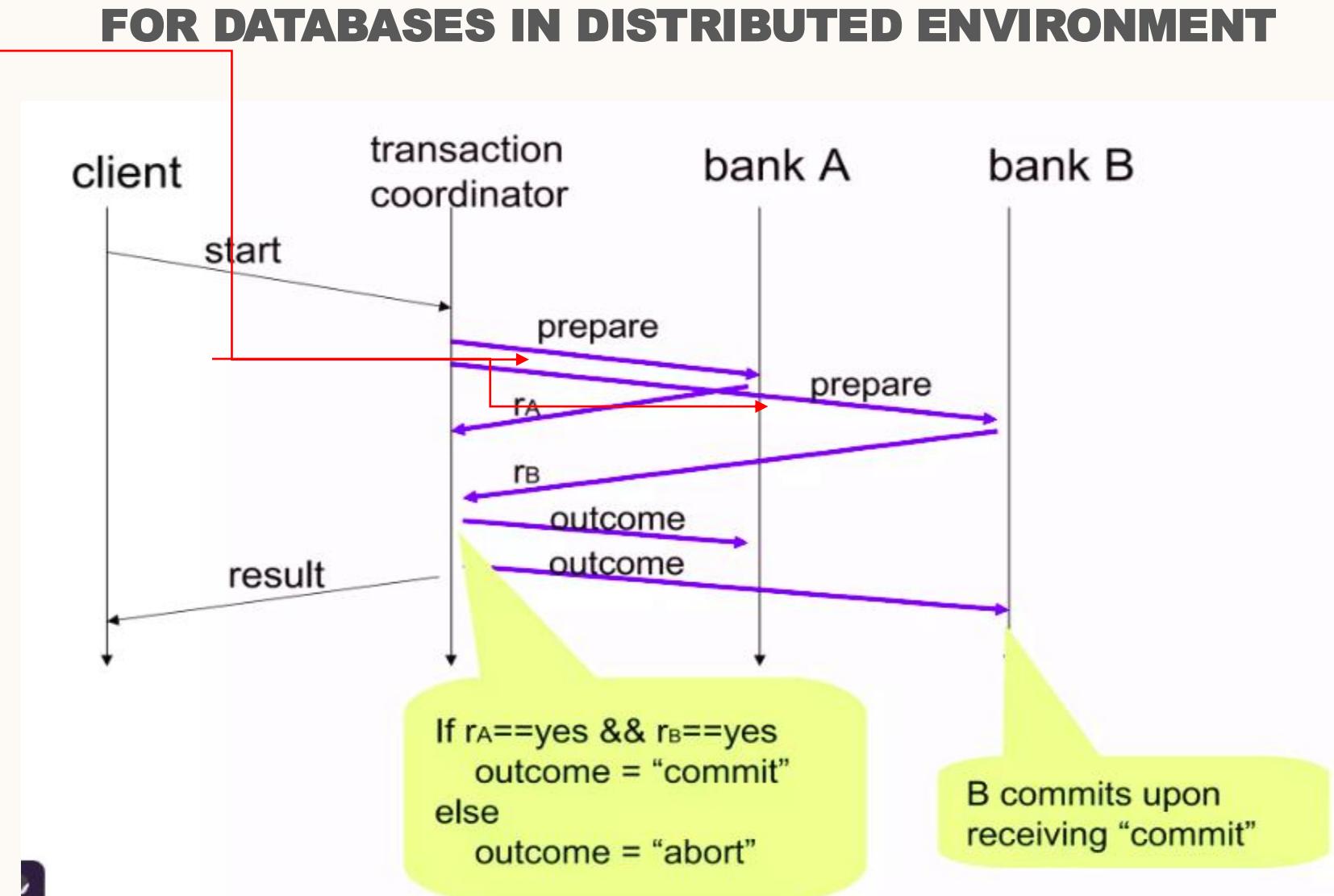
### FOR DATABASES IN DISTRIBUTED ENVIRONMENT

*Client wants to  
save changes*



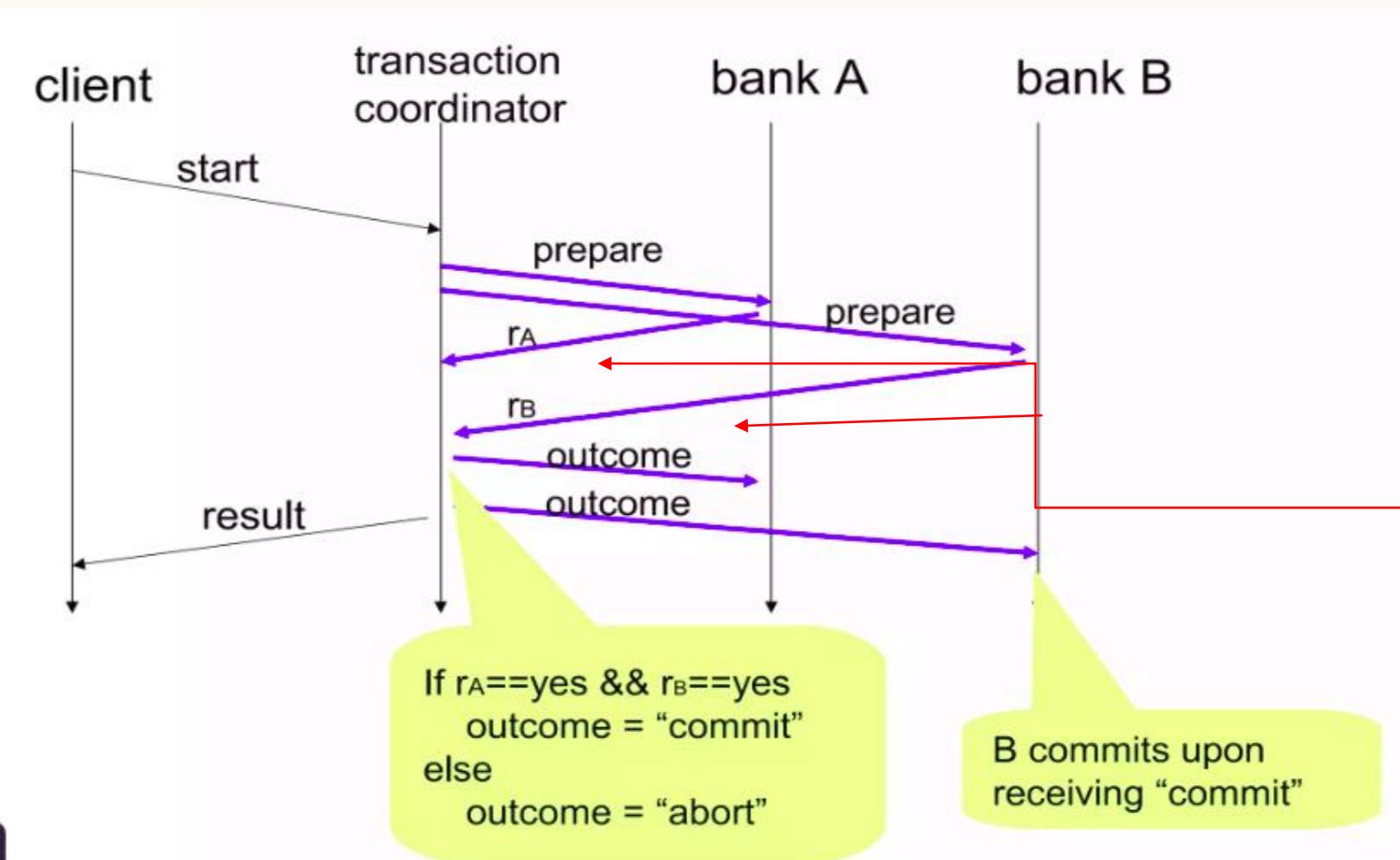
## WE NEED 2PC PROTOCOL: AN EXAMPLE RUNDOWN(1)

*Coordinator  
sends prepare  
message to A and  
B*



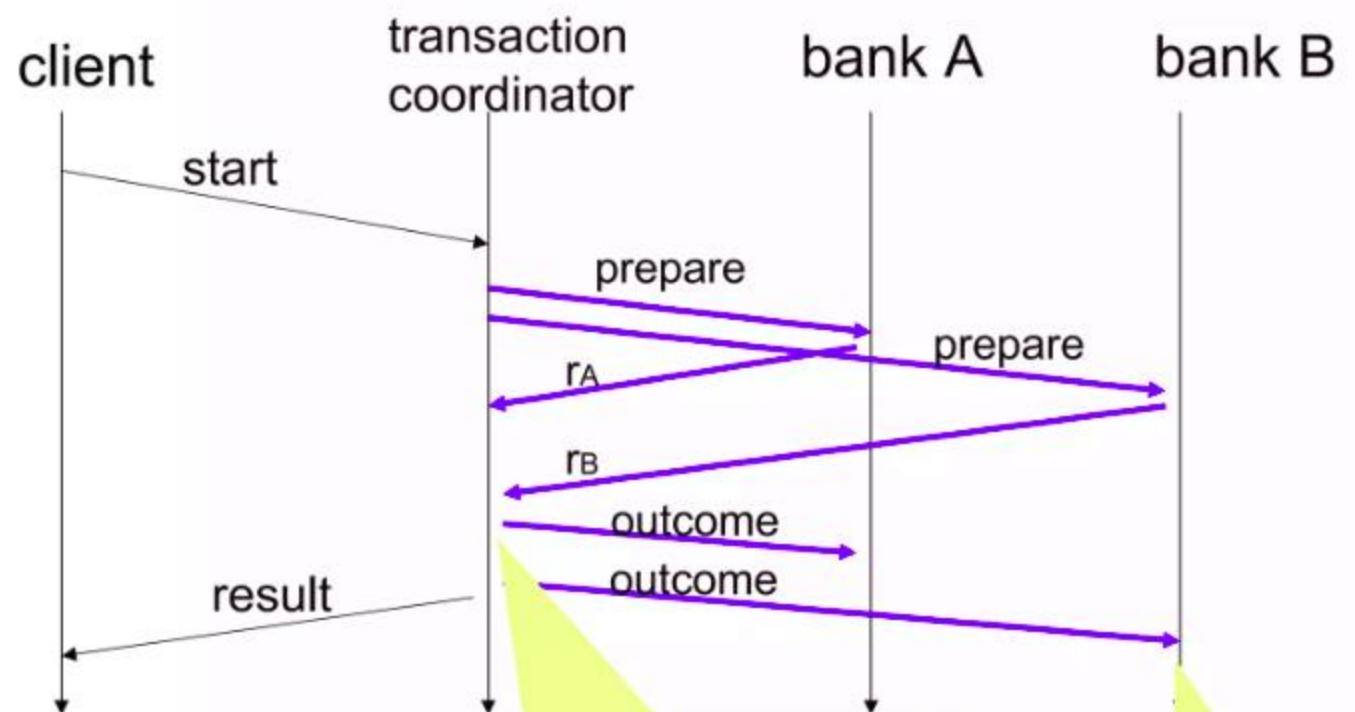
## WE NEED 2PC PROTOCOL: AN EXAMPLE RUNDOWN(1)

### FOR DATABASES IN DISTRIBUTED ENVIRONMENT



## WE NEED 2PC PROTOCOL: AN EXAMPLE RUNDOWN(1)

### FOR DATABASES IN DISTRIBUTED ENVIRONMENT



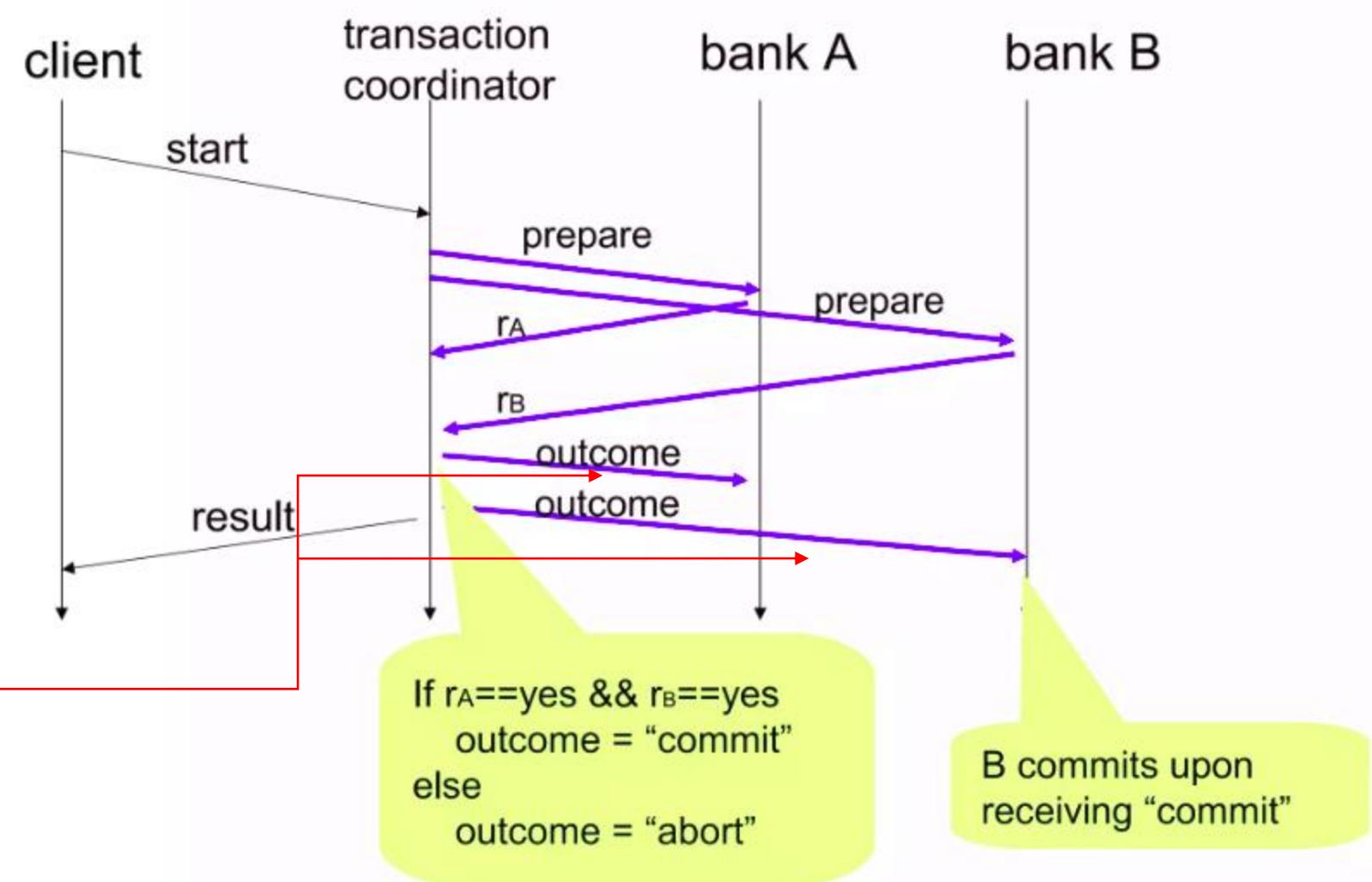
*Condition is checked for yes or not*

If  $r_A == \text{yes} \& r_B == \text{yes}$   
outcome = "commit"  
else  
outcome = "abort"

B commits upon receiving "commit"

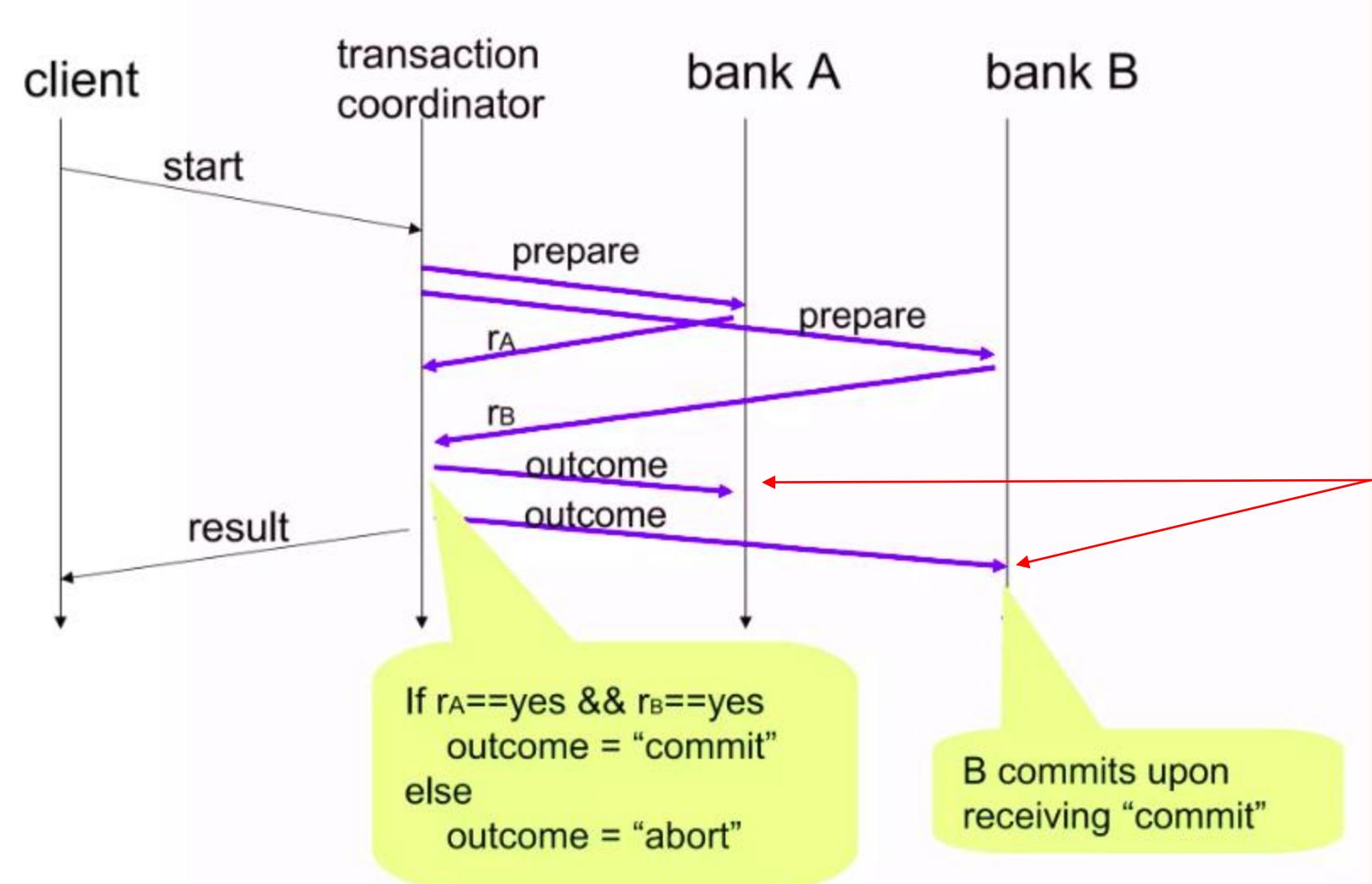
## WE NEED 2PC PROTOCOL: AN EXAMPLE RUNDOWN(1)

### FOR DATABASES IN DISTRIBUTED ENVIRONMENT



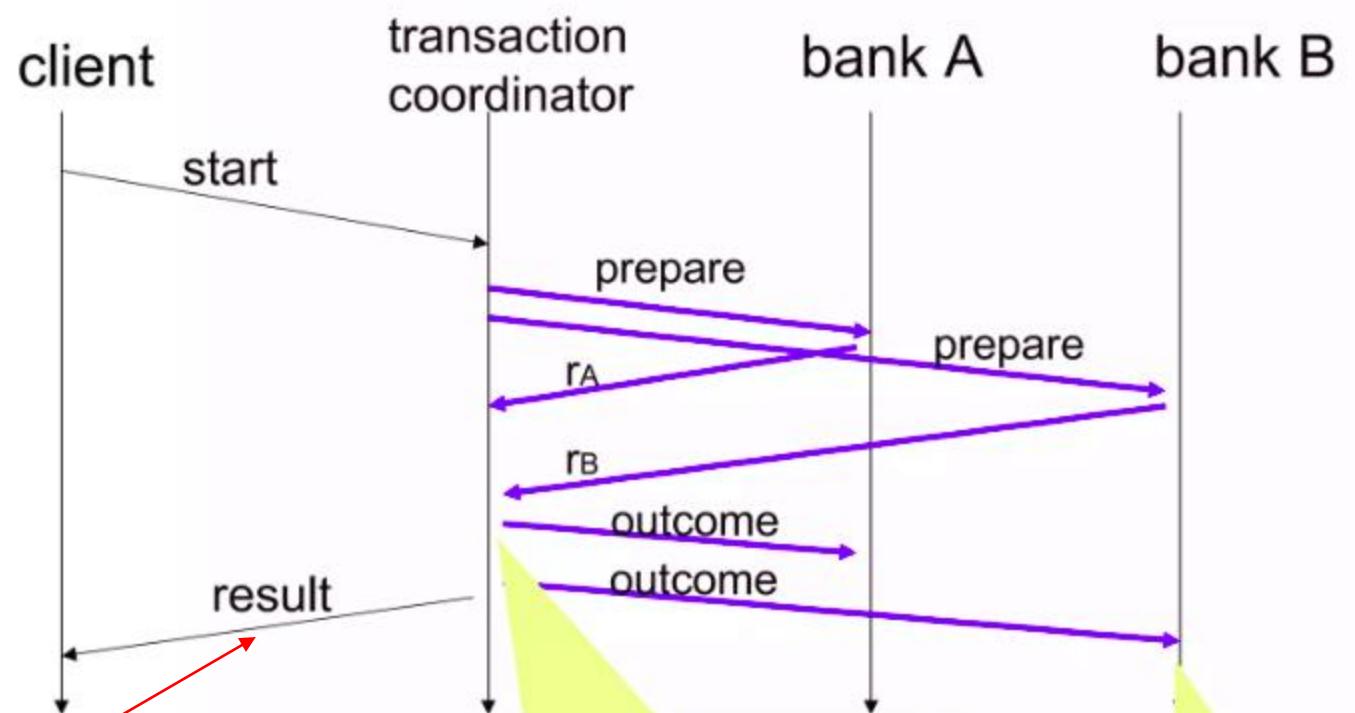
## WE NEED 2PC PROTOCOL: AN EXAMPLE RUNDOWN(1)

### FOR DATABASES IN DISTRIBUTED ENVIRONMENT



## WE NEED 2PC PROTOCOL: AN EXAMPLE RUNDOWN(1)

### FOR DATABASES IN DISTRIBUTED ENVIRONMENT



Coordinator  
sends result  
back to client

If  $r_A == \text{yes} \ \& \ r_B == \text{yes}$   
outcome = "commit"  
else  
outcome = "abort"

B commits upon  
receiving "commit"

# 2PC: THE PROBLEM

## Problem:

- If any service **fails**, the **entire transaction is blocked**, reducing availability.
- Applying **locks** on databases may compromise availability.
- In distributed systems, this leads to **delays, high latency, adding system downtime**.

## Short Example:

Say each database's uptime is 99.9%. Since we have two databases now, so total uptime is 99.8%. So, down time doubles.

Also, downtime may further increase due to 2PC fails, since most of the databases use locking while doing 2PC.

## **WE GOT CONSISTENCY WITH 2PC**

### **HOW THIS DESIGN IMPACTS AVAILABILITY?**

- The availability of any system is the product of the availability of the components required for operation.
- A transaction involving two databases in a 2PC commit will have the availability of the product of the availability of each database.
- For example, if we assume each database has 99.9 percent availability, then the availability of the transaction becomes 99.8 percent, or an additional downtime of 43 minutes per month.

# BASE

## AN ALTERNATIVE TO ACID: MORE AVAILABILITY

If ACID and 2PC provides the consistency choice for partitioned databases,

then how do you achieve availability instead?

One answer is **BASE**

(Basically Available, Soft state, Eventually consistent).

### Main Idea:

We want our systems to be more available (user experience better)  
and we can instead trade some consistency.

# BASE PRINCIPLE

- BASE is diametrically opposed to ACID.
- ACID is pessimistic and forces consistency at the end of every operation.
- BASE is optimistic and accepts that the database consistency will be in a state of flux.

Although this sounds impossible to cope with,  
in reality it is quite manageable and leads to levels of scalability  
that cannot be obtained with ACID.

# BASE IN DISTRIBUTED ENVIRONMENT

- The availability of BASE is achieved through supporting partial failures without total system failure.
- Here is a simple example:

If users are partitioned across five database servers, BASE design encourages crafting operations in such a way that one user database failure impacts only the 20 percent of the users on that particular host. There is no magic involved, but this does lead to higher perceived availability of the system.

# BASE EXAMPLE

## Example of BASE in Action: Amazon DynamoDB

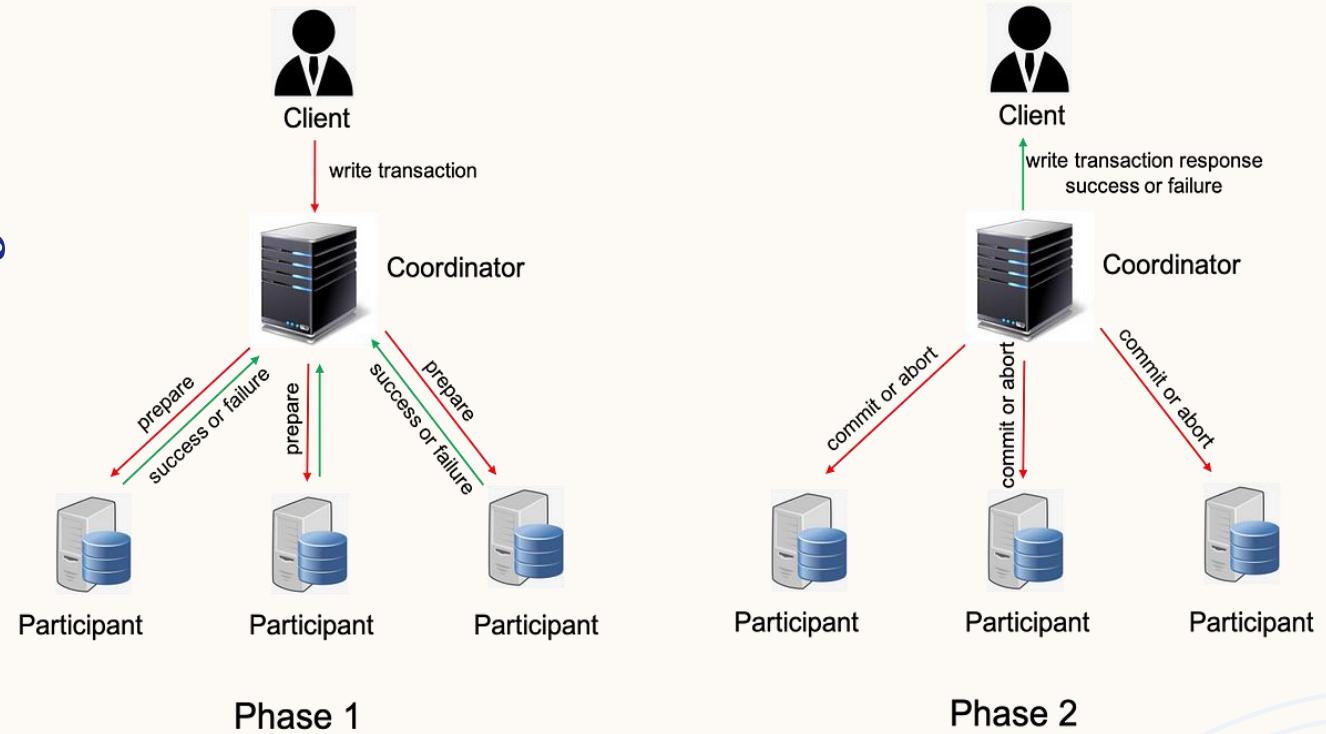
- Let's say an e-commerce site uses a **BASE-compliant** database like **DynamoDB**:
- A **customer places an order** and a write happens on Node A.
- Another **customer views orders** from Node B before the update propagates.
- The second customer **might not see the new order immediately**.
- However, after a few seconds (or minutes), the system **synchronizes**, and all nodes reflect the same data.

# IF WE GO FOR HARD CONSISTENCY,

- But now if we want to have consistency, we have to resort to 2PC.

## What is 2PC (Two-phase commit)?

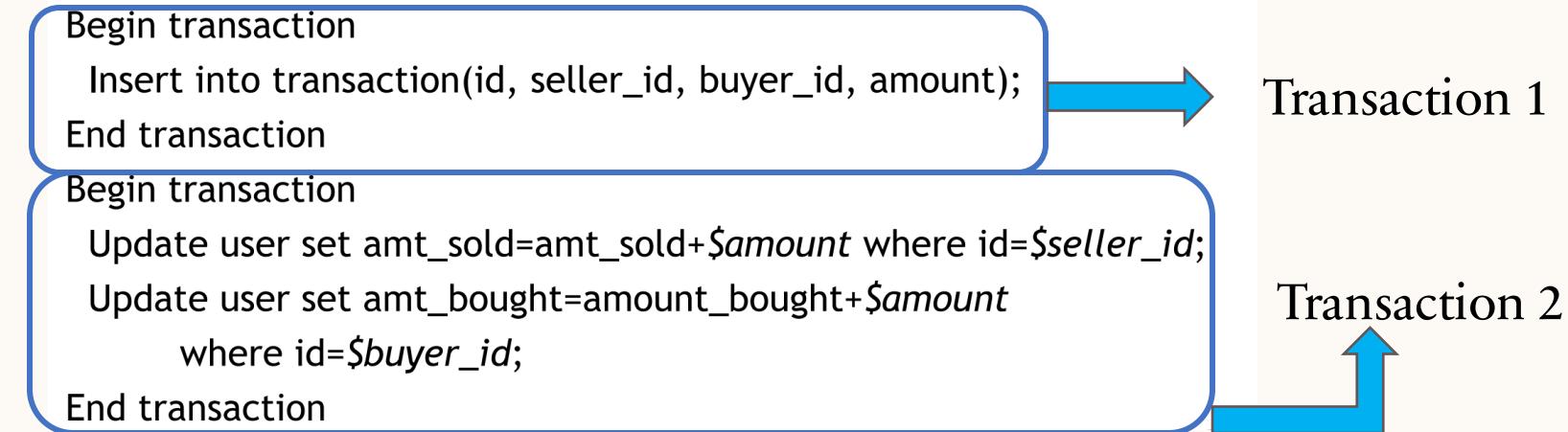
- In Part I, a coordinator sends all the update requests to all the databases.
- If all databases agree on the update then Part II starts and all updates are committed, otherwise they are rolled back.



## Problems with 2PC:

- If any service fails, the entire transaction is blocked, reducing availability.
- Applying locks on databases may compromise availability.
- In distributed systems, this leads to delays, high latency, adding system downtime.

# AVOIDING 2PC USING BASE



- The buyer and seller expectations can be set so their running balances do not reflect the result of a transaction immediately.
- This is not uncommon, and in fact people encounter this delay between a transaction and their running balance regularly (e.g., ATM withdrawals and cellphone calls)

# PROBLEM !

- If the **contract explicitly states** that the running totals are the estimates, then it may be considered **acceptable**.
- But **if not**, the changes in one table might not be consistent with the other one.
- A failure between the two transactions might result in permanent inconsistency in one of the tables.

# BASE

## PERSISTENT QUEUES

- Introducing a persistent message queue solves the problem of inconsistency.
- The most critical factor in implementing the queue: ensuring that the backing persistence is on the same resource as the database.
- Why? : to allow the queue to be transactionally committed without involving a 2PC.

```

Begin transaction
Insert into transaction(id, seller_id, buyer_id, amount);
Queue message "update user("seller", seller_id, amount)";
Queue message "update user("buyer", buyer_id, amount)";
End transaction
For each message in queue
Begin transaction
Dequeue message
If message.balance == "seller"
Update user set amt_sold=amt_sold + message.amount
where id=message.id;
Else
Update user set amt_bought=amt_bought + message.amount
where id=message.id;
End if
End transaction
End for

```

FIG

# BASE PERSISTENT QUEUES

Begin transaction

```
Insert into transaction(id, seller_id, buyer_id, amount);
Queue message "update user("seller", seller_id, amount)";
Queue message "update user("buyer", buyer_id, amount)";
```

End transaction



For each message in queue

```
Begin transaction
Dequeue message
If message.balance == "seller"
    Update user set amt_sold=amt_sold + message.amount
        where id=message.id;
```



Else

```
    Update user set amt_bought=amt_bought + message.amount
        where id=message.id;
```

End if

End transaction

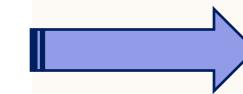
End for

# FIG

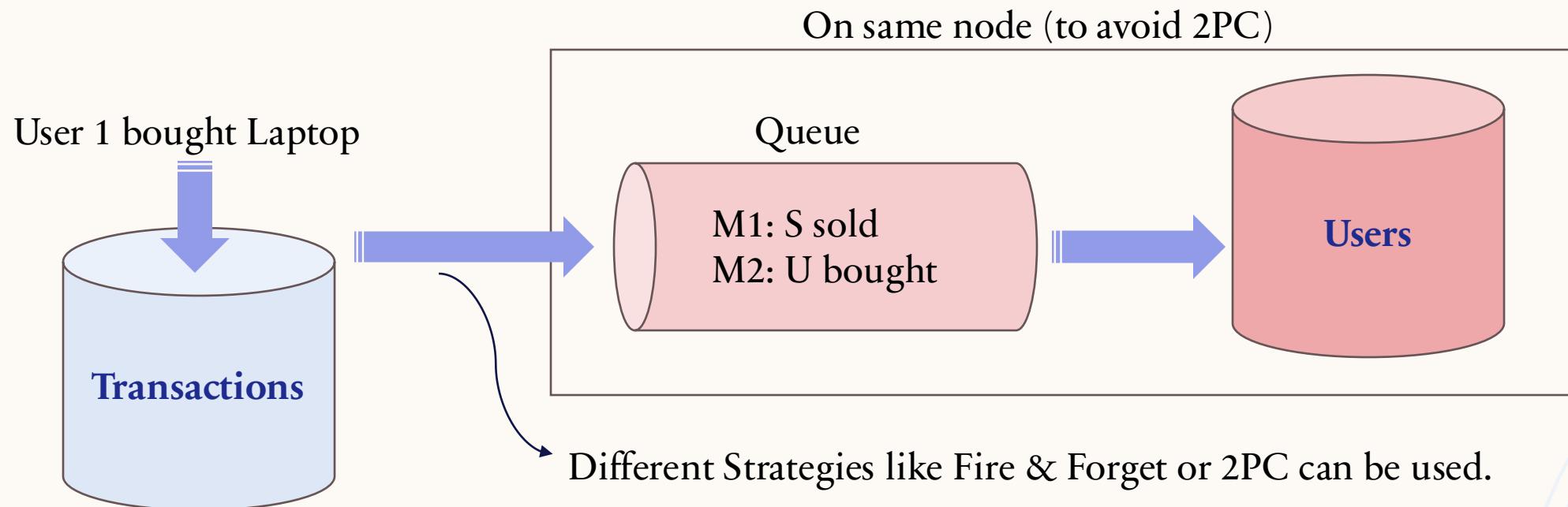
# BASE PERSISTENT QUEUES

Begin transaction

```
Insert into transaction(id, seller_id, buyer_id, amount);
Queue message "update user("seller", seller_id, amount)";
Queue message "update user("buyer", buyer_id, amount)";
End transaction
```



Inserts row in  
**'transaction'** table and  
sends update to the queue  
of **'user'** table



# BASE PERSISTENT QUEUES

For each message in queue

Begin transaction

Dequeue message

If message.balance == "seller"

    Update user set amt\_sold=amt\_sold + message.amount  
        where id=message.id;

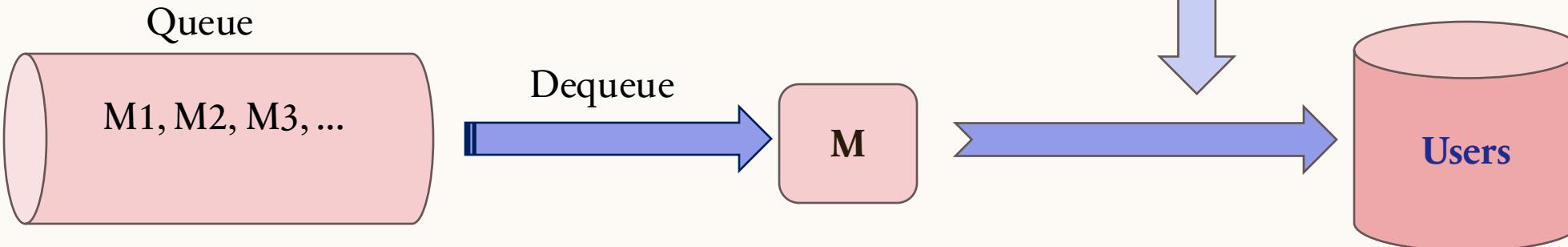
Else

    Update user set amt\_bought=amt\_bought + message.amount  
        where id=message.id;

End if

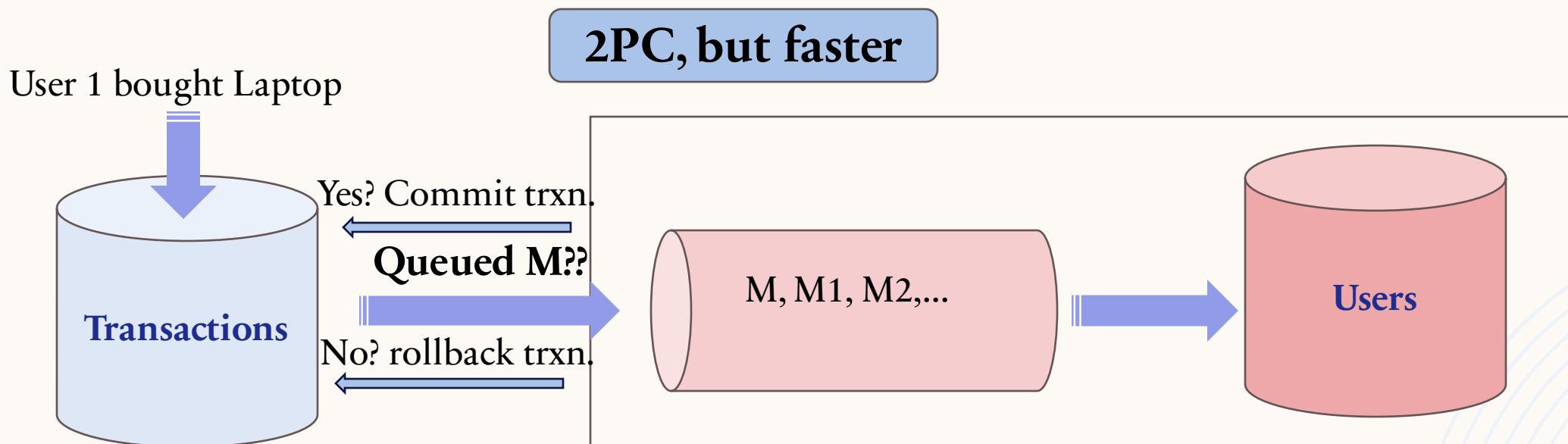
End transaction

End for



# ISSUE PERSISTS!

1. Messages are queued separately but processing them still requires coordination between the message queue and the database → Again 2PC problem.
2. If an update (e.g., inventory or payment) is processed within a transaction but the database fails, rollback is needed, leading to partial failures.



# SOLVING 2PC PROBLEM

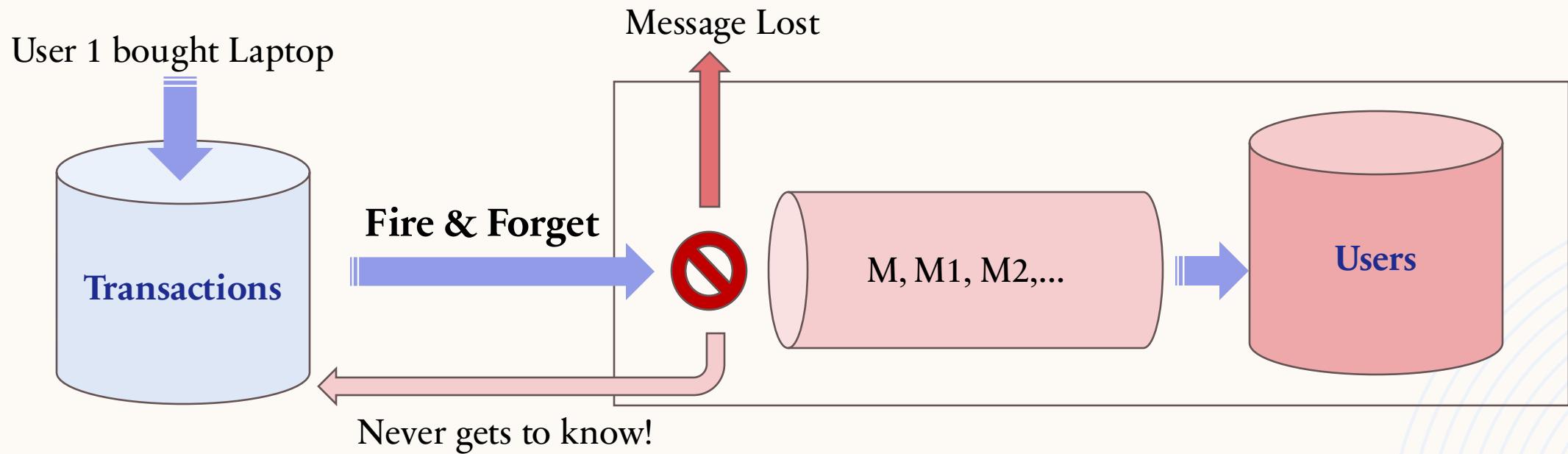
- One solution to the previous slow 2PC in the message-processing component is to do nothing; decouple the update into separate back-end component. Still 2PC but now lower availability for business requirements is acceptable.

## **Say 2PC is never acceptable: Fire and Forget**

- **Idempotence:** An operation is considered idempotent if it can be applied one time or multiple times with the same result. Idempotent operations are useful in that they permit partial failures, as applying them repeatedly does not change the final state of the system.

# ISSUE PERSISTS!

If we use Fire & Forget, then there can be issue of message loss.  
This creates an issue of permanent inconsistency.



# IDEMPOTENCE(1)

- Update operations are rarely idempotent.
- Increments and decrements are not idempotent as applying them more than once will not have same effect as applying once.  
Example: Wallet = Wallet -100
- Some updates can be idempotent.  
Example: Wallet = 1000
- However, in cases like above, we need to make sure that the order in which these events are received don't result in inconsistent data.

## IDEMPOTENCE(2)

- Using a table that records the transaction identifiers that have been applied can help.
- The table tracks the transactions ID, which balance has been updated, and the user ID where the balance was applied.
- To incorporate idempotence we update our example to contain a transaction id of the update for each update. So now, whenever an update is issued, we check the 'processed' field in the db for that update and apply it accordingly.
- We can keep a track of which updates have been applied.

updates_applied
trans_id
balance
user_id

# SOLVING F&F

- We peek a message in the queue and remove it once successfully processed.
- Queue operations are not committed unless database operations successfully commit (not 2PC since queue and db in same node).
- The algorithm now supports partial failures and still provides transactional guarantees without resorting to 2PC.

```

Begin transaction
Insert into transaction(id, seller_id, buyer_id, amount);
Queue message "update user("seller", seller_id, amount)";
Queue message "update user("buyer", buyer_id, amount)";
End transaction

For each message in queue
Peek message
Begin transaction
Select count(*) as processed where trans_id=message.trans_id
and balance=message.balance and user_id=message.user_id
If processed == 0
If message.balance == "seller"
Update user set amt_sold=amt_sold + message.amount
where id=message.id;
Else
Update user set amt_bought=amt_bought + message.amount
where id=message.id;
End if
Insert into updates_applied
(message.trans_id, message.balance, message.user_id);
End if
End transaction
If transaction successful
Remove message from queue
End if
End for

```

Block 1 remains  
same

FIG 7

# SOLVING F&F (2)

For each message in queue

Peek message

Begin transaction

Select count(\*) as processed where trans\_id=message.trans\_id  
and balance=message.balance and user\_id=message.user\_id

If processed == 0

```
If message.balance == "seller"
    Update user set amt_sold=amt_sold + message.amount
        where id=message.id;
Else
    Update user set amt_bought=amt_bought + message.amount
        where id=message.id;
End if
Insert into updates_applied
    (message.trans_id, message.balance, message.user_id);
End if
```

End transaction

If transaction successful

Remove message from queue

End if

End for

## Action for each msg in queue

Selects all transactions  
with same details as the  
message you peeked.  
**Counts them.**

If count is 0 i.e. no  
previous txn. with same  
details, then processes that  
message.  
Else, ends the db txn.

# FIG 7

(We already have a table  
containing the txn logs, we  
check that table.)

# MINOR ISSUES

## ORDERING ISSUE

We might want to preserve the ordering of the updates!

- Consecutive transaction ids may be produced by the msg producer.
- If we need something like 'last sale time', then we can attach a timestamp along with trxn. id for checking.
- Store last update's timestamp and compare with the current msg.

### SAMPLE CODE

For each message in queue:

    Peek message

    Begin Transaction

        Update user set last\_purchase=message.trans\_date where id=message.buyer\_id  
        and last\_purchase < message.trans\_time

    End Transaction

    If transaction successful

        Remove message from queue

End For

# MINOR ISSUES

## SOFT STATE: EVENTUAL CONSISTENCY

Up to this point, the focus has been on trading consistency for availability.

Consider a system where users can transfer assets to other users

There is a period of time where the asset has left one user and has not arrived at the other

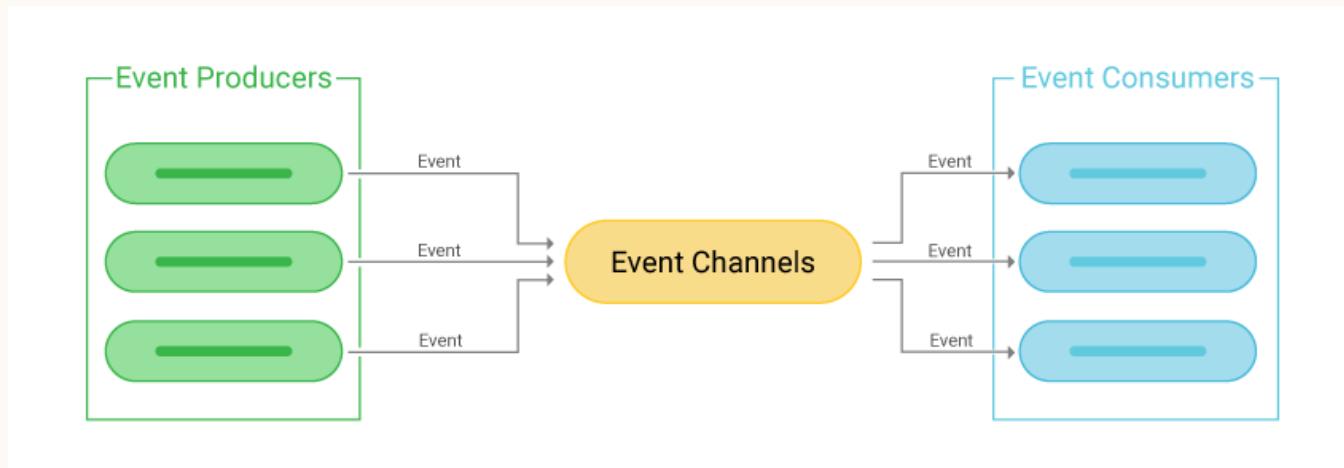
If the lag between sending and receiving is a few seconds, it will be invisible or certainly tolerable to users who are directly communicating about the asset transfer.

In this situation the system behavior is considered consistent and acceptable to the users, even though we are relying upon soft state and eventual consistency in the implementation.

# EVENT DRIVEN ARCHITECTURE

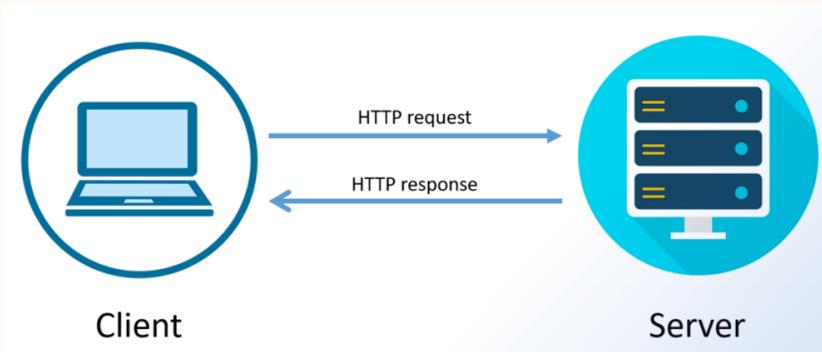
## TO THE RESCUE

- Event-Driven Architecture (EDA) is a **software design pattern** where systems **react to events** instead of following a predefined flow.
- It enables to communicate **asynchronously** through **events**.



## WHY REQUEST-RESPONSE MODEL IS NOT GOOD

- Request-Response architecture is the most popular and common messaging system .
- REST is one of the most common form of messaging protocols used in distributed systems.
- The client makes a *Request* which is serviced by the server to provide a *Response*.



### Issues:

- The problem arises as the distributed architecture gets *complex*.
- Introduction of *multiple services* creates a chain of strongly connected components.
- This increases the latency and wait time, and failure in one can lead to cascading failures.

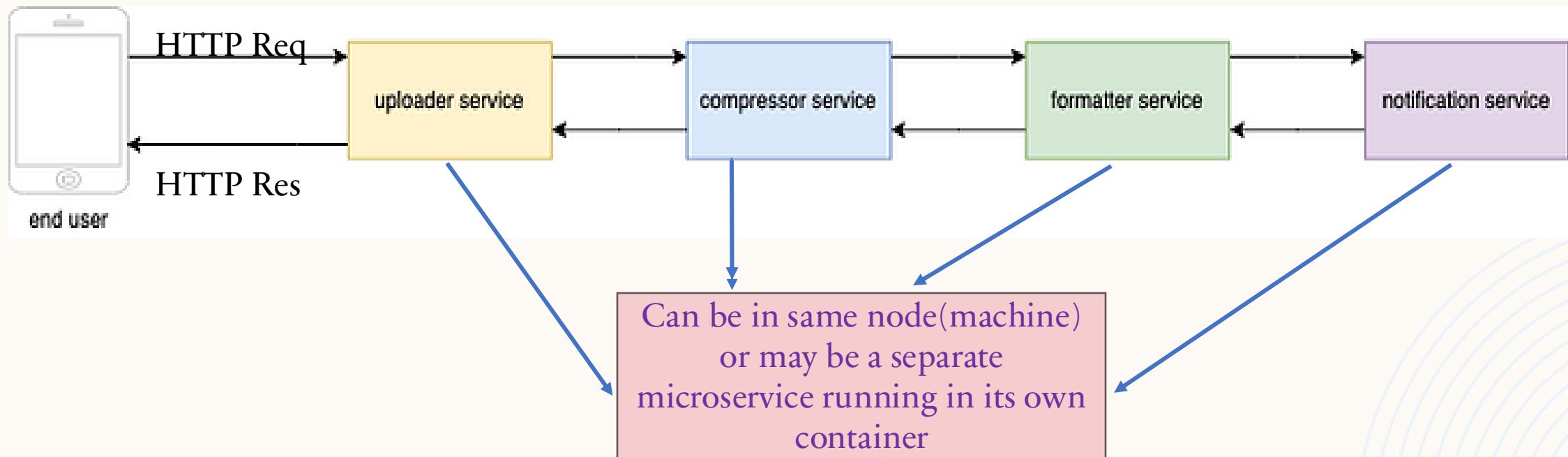
# NEED FOR EDA

60

## WHY REQUEST-RESPONSE MODEL IS NOT GOOD

Let's say we have a video processing service

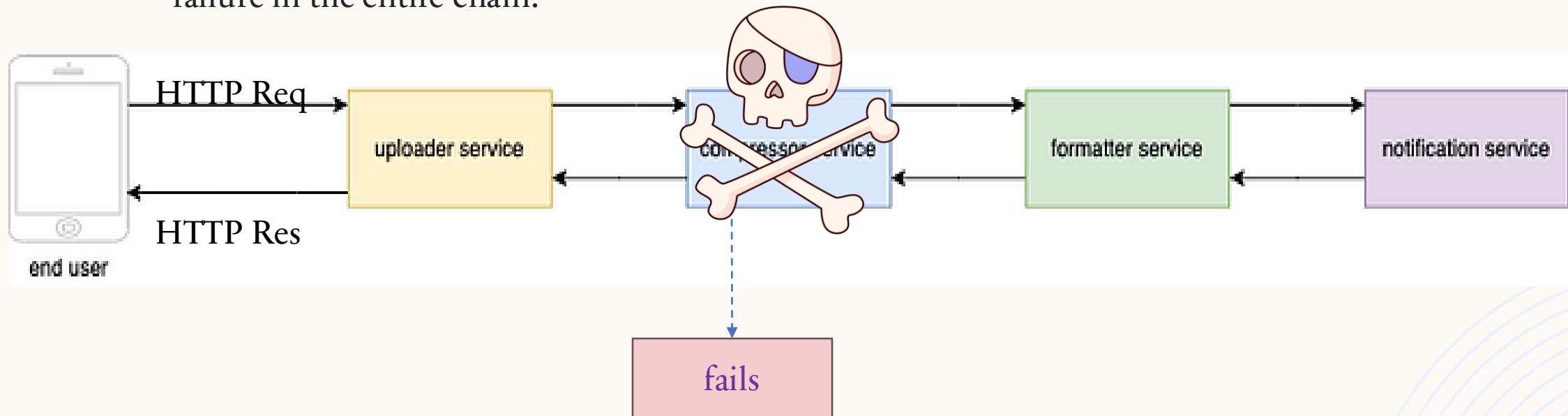
1. Processing has to happen in the given order
2. Only one service is front user facing



## WHY REQUEST-RESPONSE MODEL IS NOT GOOD

Let's say we have a video processing service

1. The user has to wait till each service processes the request and provides a response.
2. This would increase latency.
3. The system would also not be fault tolerant as a failure in any one of the services would lead to failure in the entire chain.



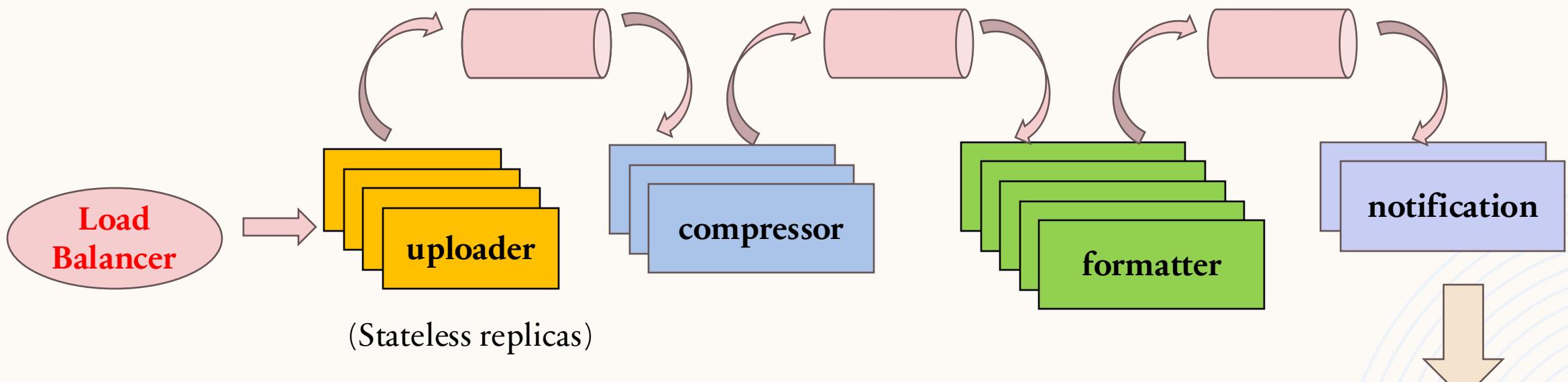
# NEED FOR EDA

62

## WHY EDA IS GOOD

Let's say we have a video processing service

1. Using this architecture we can scale each component individually and make them fault tolerant.
2. Since the messages are async we don't have to wait for response from downstream services.
3. Stateless services => Anyone can pick from the queue



# EVENT DRIVEN ARCHITECTURE

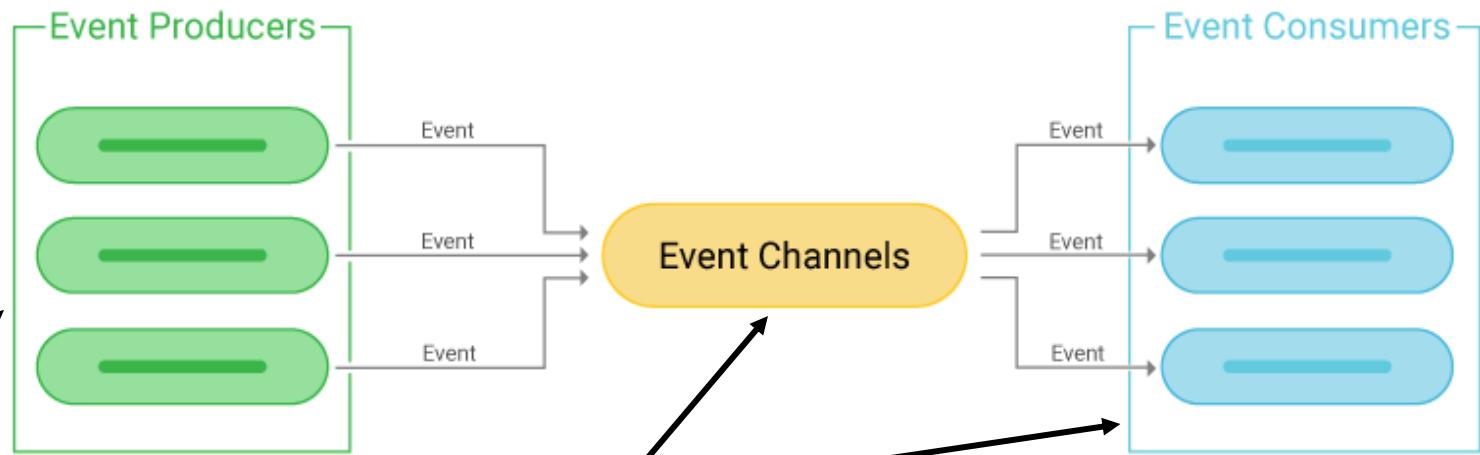
Why go for EDA?

## loose coupling

Producers and consumers can be separate microservices possibly running in their separate node/container.

Each producer/consumer may be in a container or a node in some cluster  
Also, not point 2 point connection

## TO THE RESCUE



They can be 3rd party service like AWS or self-hosted service

# EVENT DRIVEN ARCHITECTURE

## Why go for EDA?

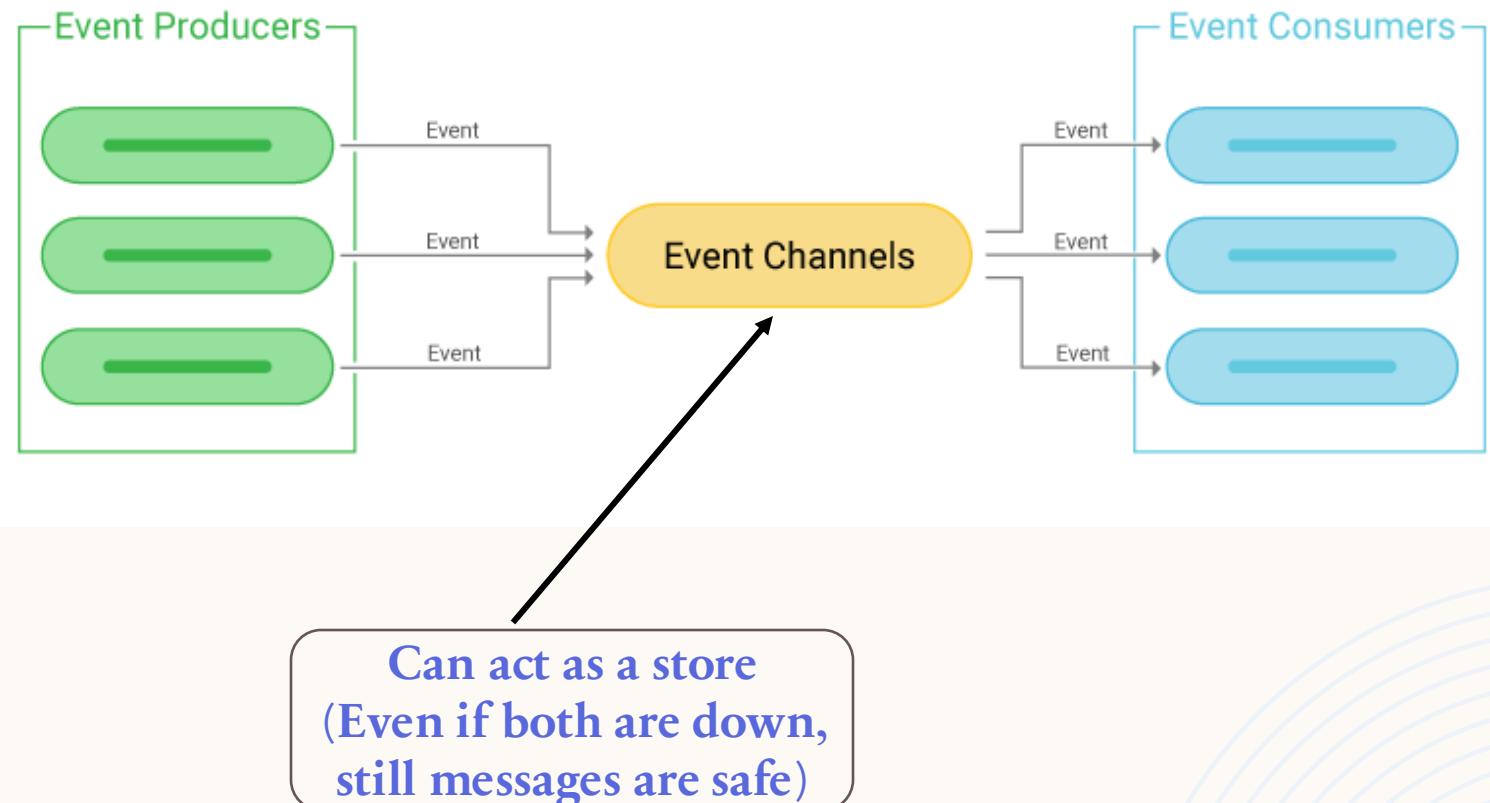
### scalability

A fast producer need not be blocked/waiting for a slower consumer.

The consumer can consume at its own pace/convinience.

Both need not be online at same time

### TO THE RESCUE



# EVENT DRIVEN ARCHITECTURE (1)

## TO THE RESCUE

### Events:

- An **event** is any significant change or occurrence in the system.
- Examples:
  - A new order is placed.
  - **Event Producers:**
- Components that **generate events** when something happens.
- Example: An e-commerce checkout system produces an event when an order is placed.

### Event Brokers (or Message Brokers):

- Acts as an intermediary that **routes events** from producers to consumers.
- Examples: **Kafka, RabbitMQ, AWS SNS, Google Pub/Sub.**

### Event Consumers:

- Components that **listen for and react to events.**
- Example: A **warehouse service** updates inventory when an order event is received.

### Event Store (Optional):

- A database that **persists events** for auditing, debugging, or replaying.
- Example: **Event Sourcing.**

# EVENT DRIVEN ARCHITECTURE (2)

## TO THE RESCUE

### Event-Driven Communication Patterns

- **Publish-Subscribe (Pub/Sub)**
  - Producers publish events to a message **broker**.
  - Multiple consumers **subscribe** and receive relevant **topics**.
  - Example: A payment service publishes a "**payment\_successful**" event, and multiple services (e.g., order service, notification service) react to it.
- **Event Streaming**
  - Continuous flow of events where consumers process them in **real time**.
  - Events are written to a log in a strictly ordered fashion.
  - Consumers can read from any point and maintain its own pointer/counter
  - Example: **Kafka Streams** processing financial transactions.

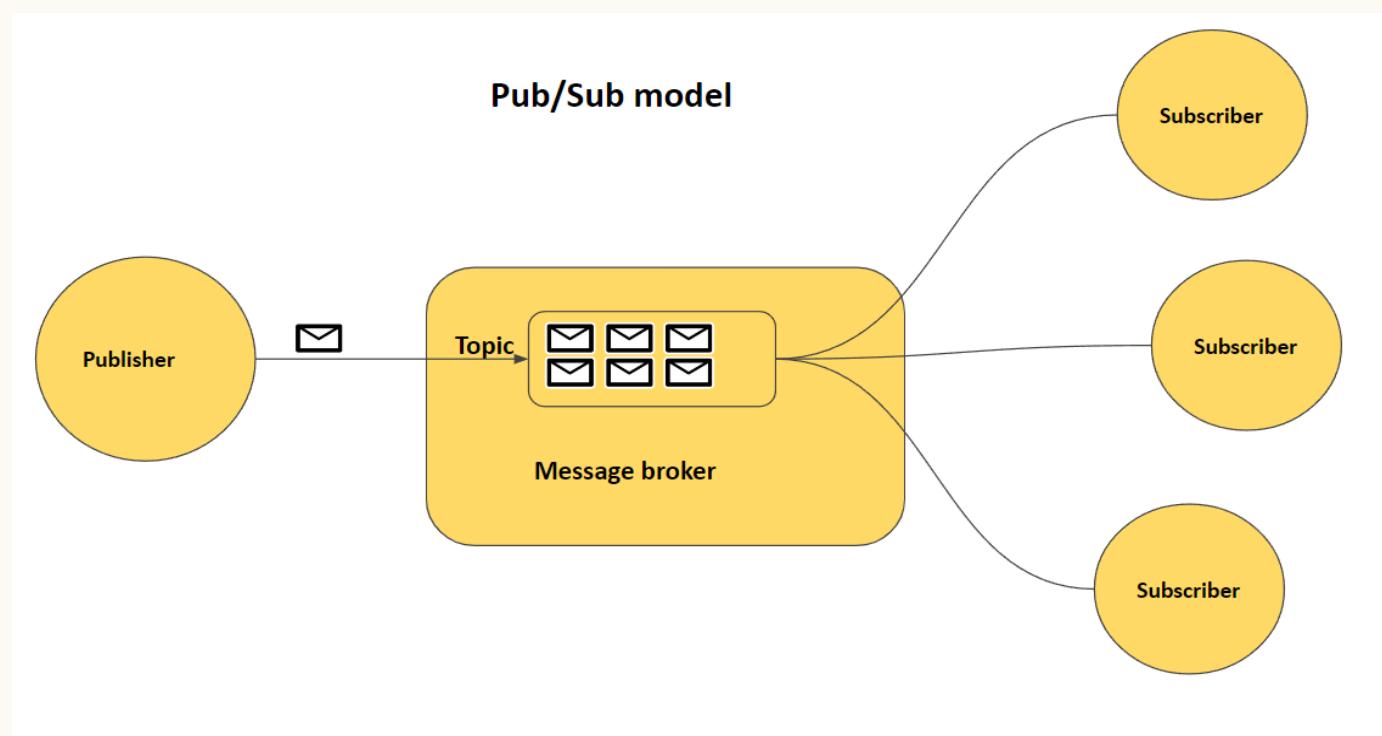
# EVENT DRIVEN ARCHITECTURE (2)

## TO THE RESCUE

### Event-Driven Communication Patterns

#### Publish-Subscribe (Pub/Sub) Model

e.g. whatsapp group notification



# HOW BASE ACHIEVABLE

## Eventual Consistency Mechanisms:

- **Asynchronous Replication:** Updates are propagated to other nodes over time.
- **Vector Clocks & Versioning:** Helps track changes in data across nodes (used in DynamoDB).
- **Conflict Resolution Strategies:** Ensures that outdated values are resolved appropriately.

## Partition-Tolerant and Available Systems (AP in CAP Theorem)

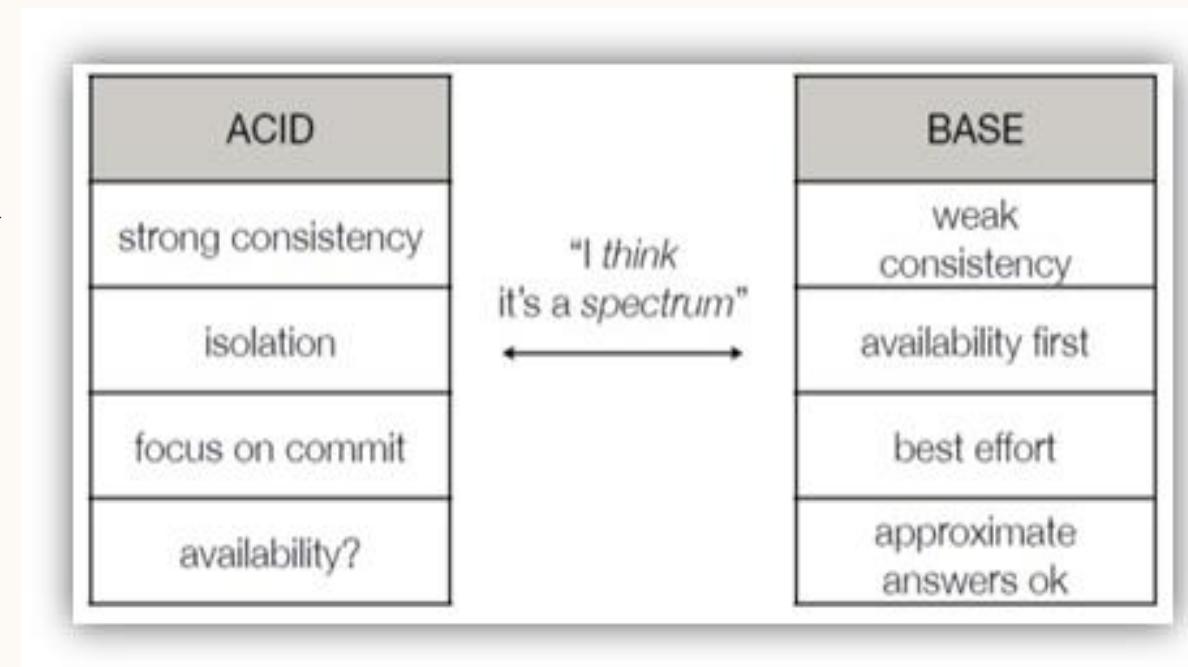
- BASE is common in NoSQL databases like **Cassandra, DynamoDB, CouchDB, and Riak**.
- Unlike ACID databases, these databases allow trade-offs in consistency to ensure high availability.

# THE BIG PICTURE ! OF BASE SYSTEMS

1. Immediately, this system feels nondeterministic and problematic.
2. There is a period of time where the asset has left one user and has not arrived at the other.
3. The size of this time window can be determined by the messaging system design.
4. Regardless, there is a lag between the begin and end states where neither user appears to have the asset.
5. We want to have higher availability and can sacrifice strong consistency.

# CONCLUSIONS

1. Scaling systems to dramatic transaction rates requires a new way of thinking about managing resources.
2. The traditional transactional models are **problematic** when loads need to be spread across a large number of components.
3. Decoupling the operations and performing them in turn provides for improved availability and scale at the cost of consistency.
4. BASE provides a model for thinking about this decoupling.



# **THANK YOU**

Debanjan Saha

Ikshita Pathak

Akash Maji