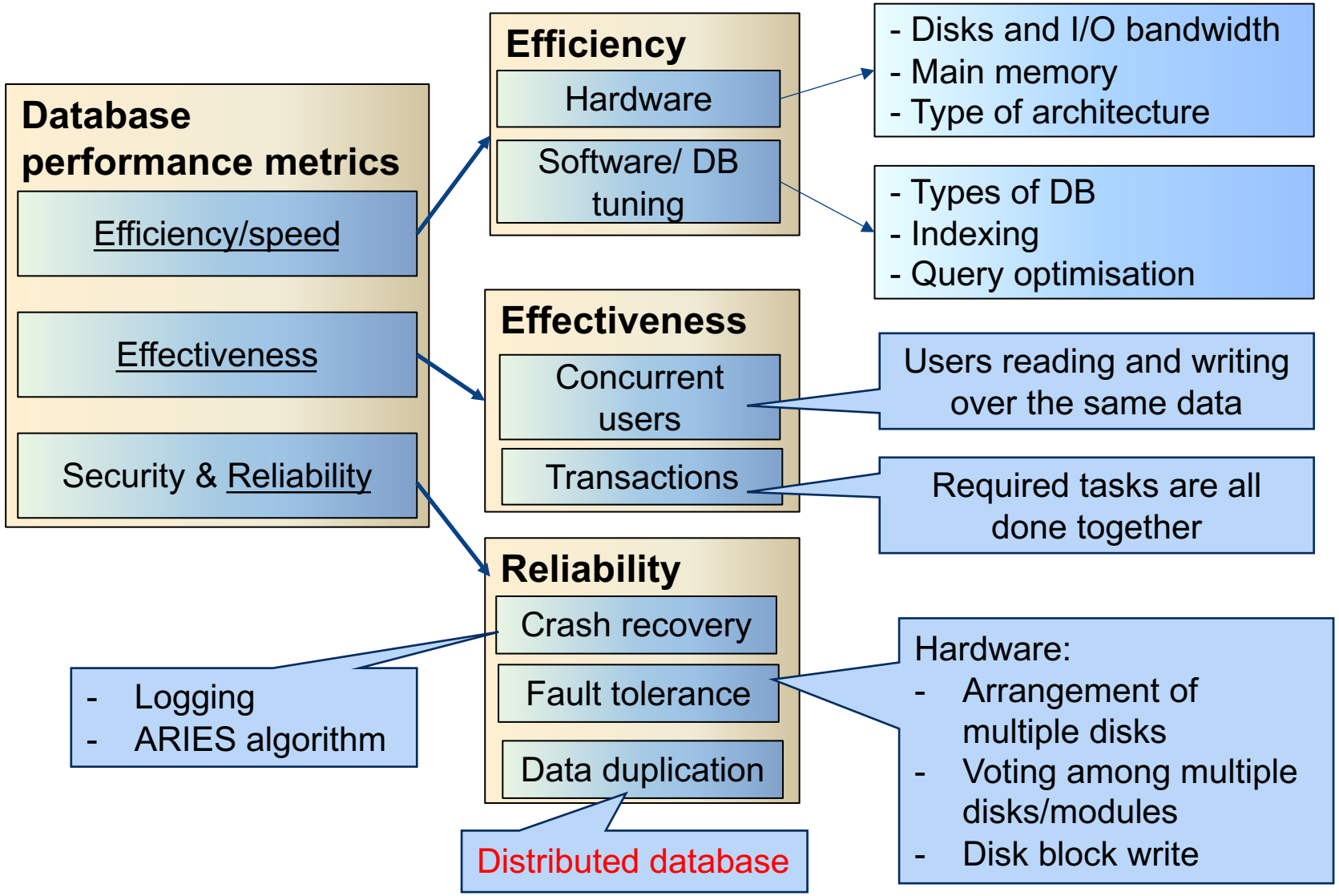
COMP90050: Advanced Database Systems

Lecturer: Farhana Choudhury (PhD)

Distributed databases

Week 11 part 1

Core Concepts of Database management system



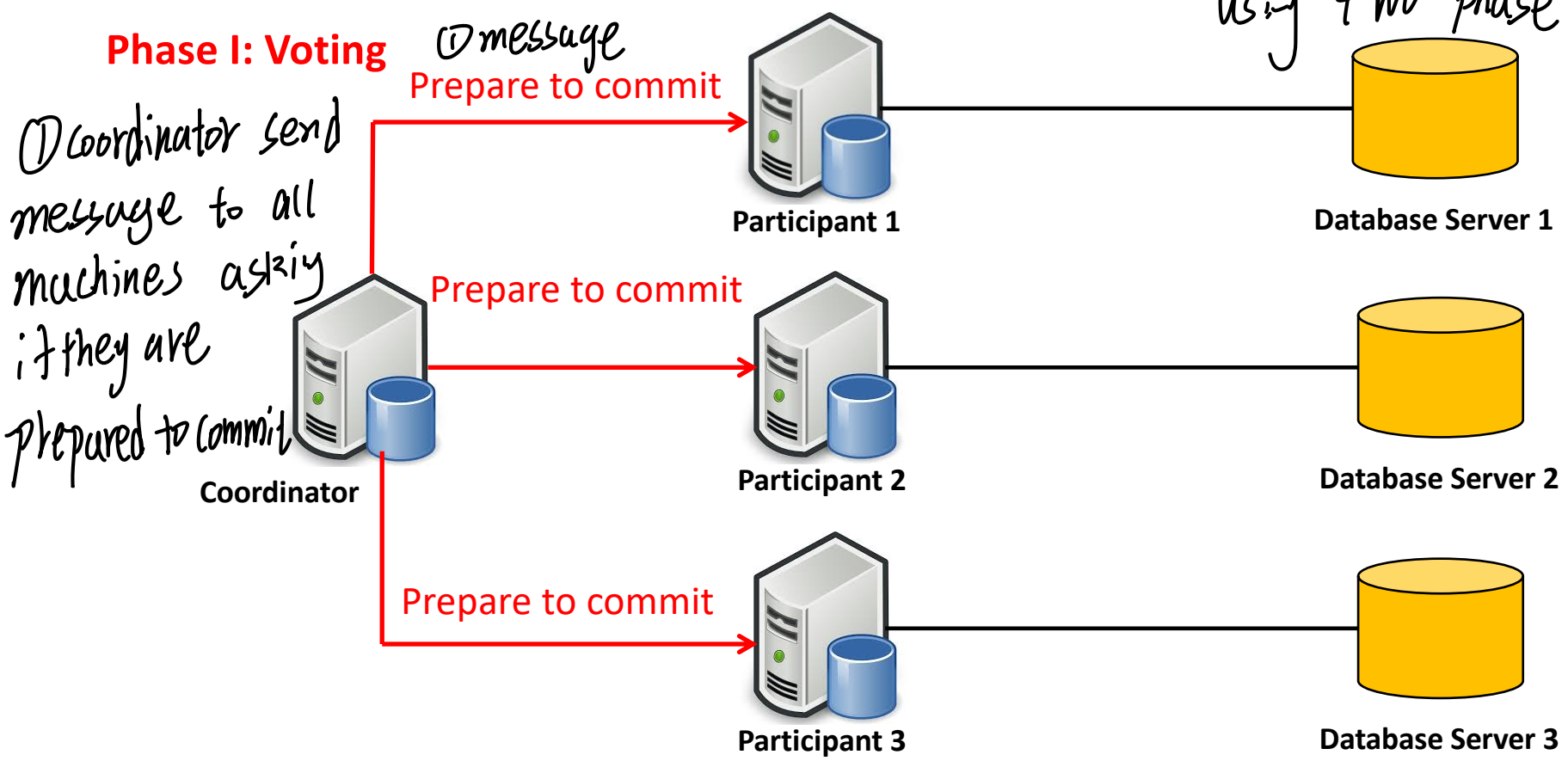
A related concept on distributed databases first - atomicity

Atomicity in distributed transaction processing

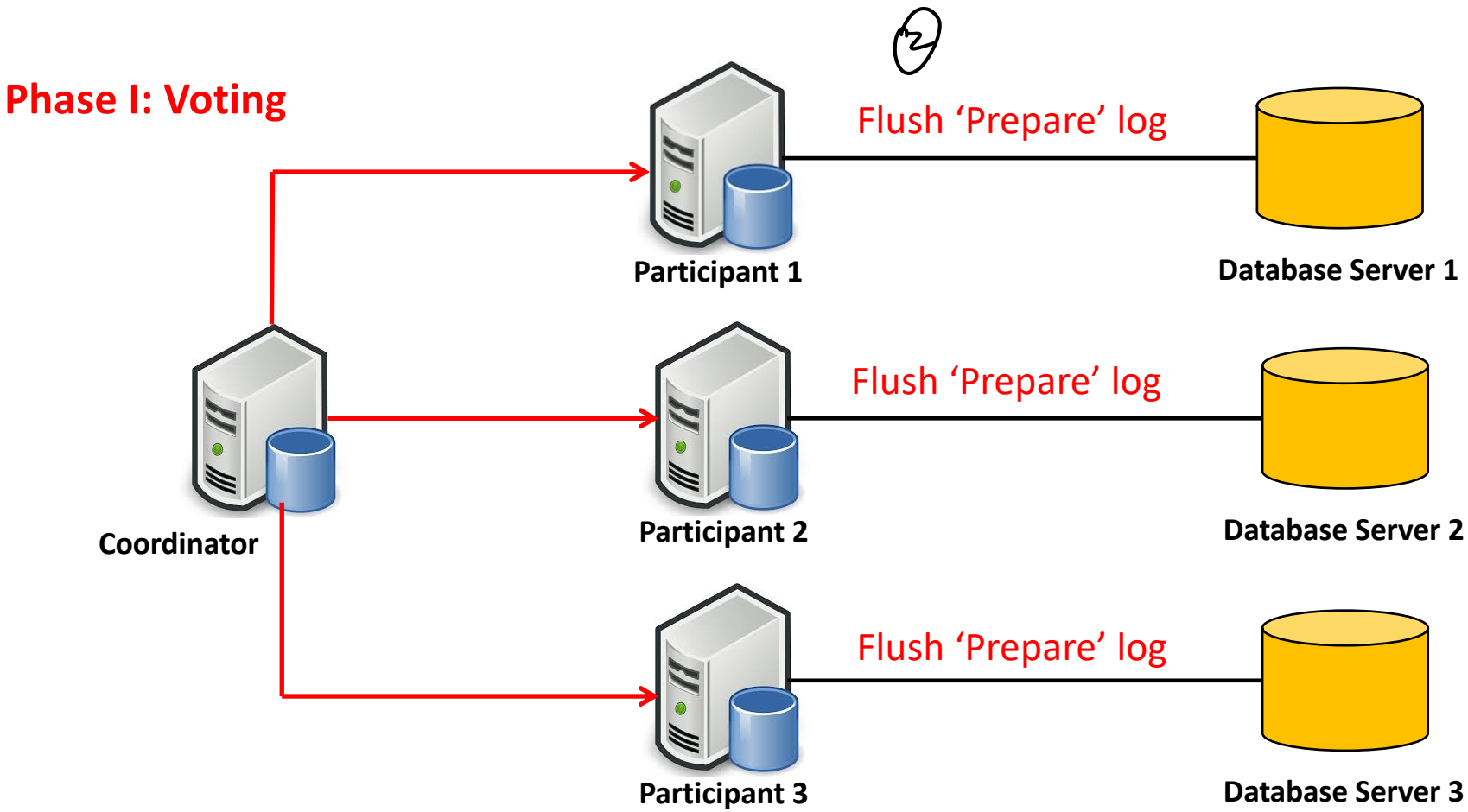
$T = \begin{pmatrix} T_1 \\ T_2 \\ T_3 \end{pmatrix}$ subtransaction.

How to maintain atomicity for each of subtransaction? Using two phase commit.

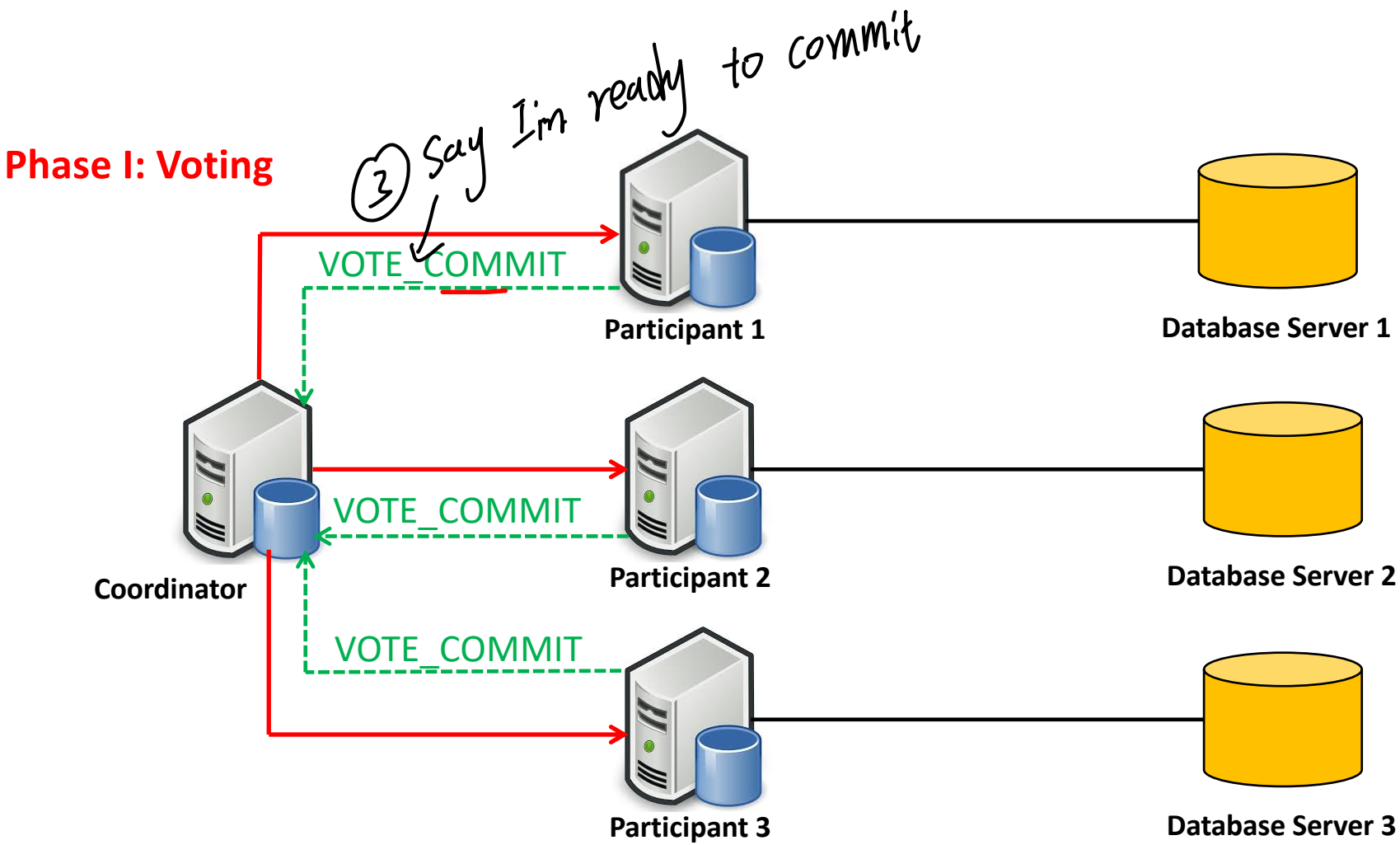
The two-phase commit protocol (2PC) can help achieve atomicity in distributed transaction processing



Two phase commit protocol



Two phase commit protocol

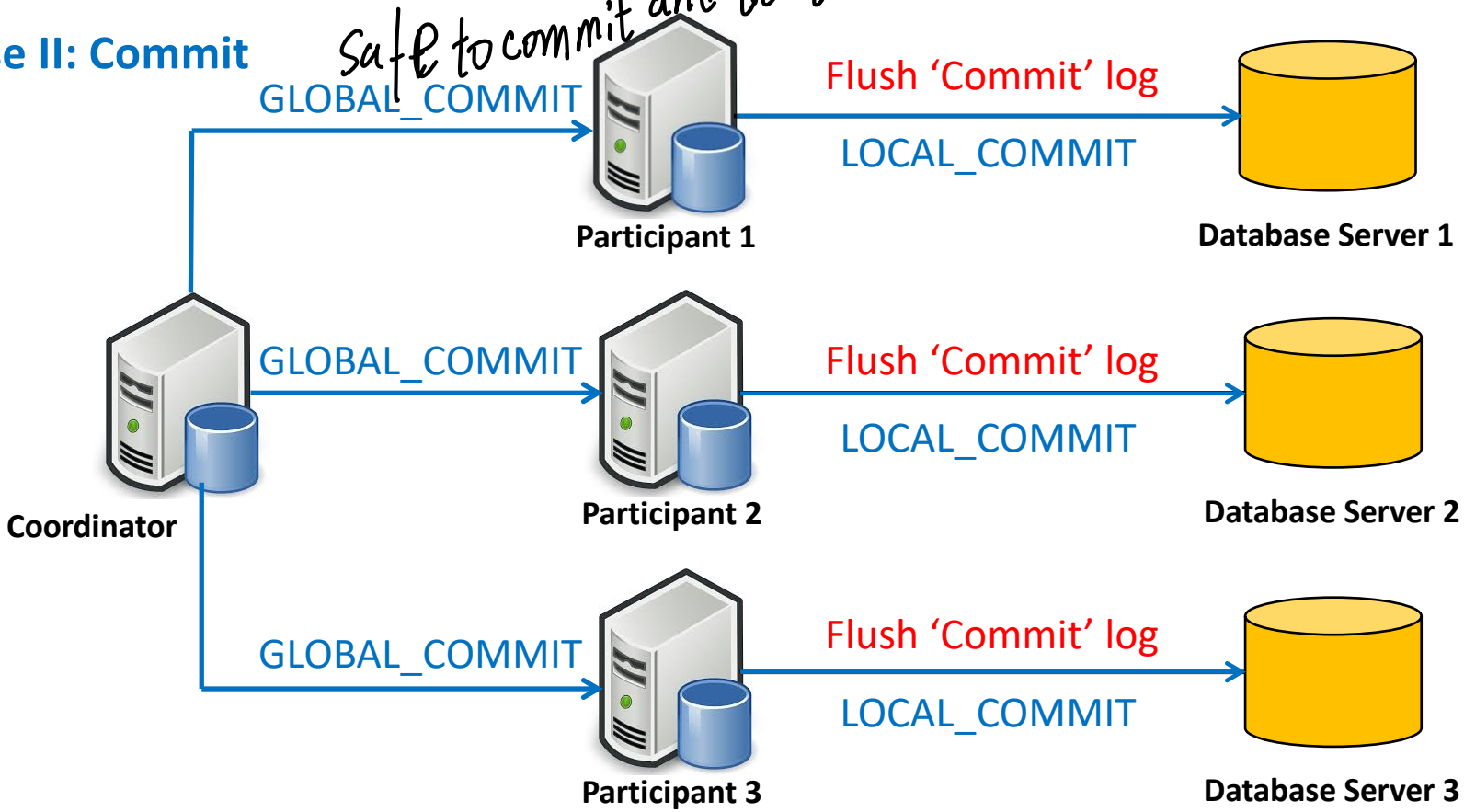


Two phase commit protocol

Coordinator receives 'ready to commit' from all participates from that distributed transaction

safe to commit and do commit

Phase II: Commit



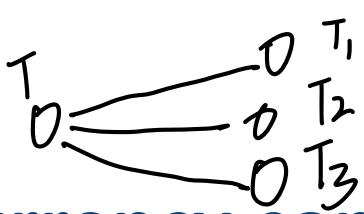
Two phase commit protocol

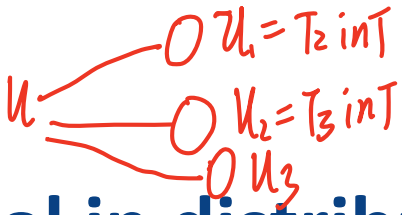
What happens if some participates abort? (participate发送abort给coordinator)

- Coordinator or participant can abort transaction
  - If a participant abort, it must inform coordinator (通知后)
  - If a participant does not respond within a timeout period, coordinator will abort (若超时)
- If abort, coordinator asks all participates to rollback (一损俱损 one sub failed => all subs failed.)
- If abort, abort logs are forced to disk at coordinator and all participants

Voting and commit all abort to make atomicity.

Another related concept on distributed databases - concurrency control





How to maintain concurrency ?

Concurrency control in distributed DBs

Using
① locks

- **Each server is responsible for applying concurrency control to its own objects**

② timestamps.

- The members of a collection of servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner

- ***BUT*** **servers independently acting would not work**

- If transaction ***T*** **is before transaction *U*** in their conflicting access to objects at one of the servers then:
  - They **must be in that order at all of the servers** whose objects are accessed in a conflicting manner by both *T* and *U*

- The central **Coordinator should assure this**

Other Considerations for Locking-based systems

- A local lock manager cannot release any locks until it knows that the transaction has been committed or aborted at all the servers involved in the transaction.

- The objects remain locked and are unavailable for other transactions during the commit protocol.
  - An aborted transaction releases its locks after phase 1 of the protocol.

Timestamp ordering concurrency control revisited

- The coordinator accessed by a transaction issues a **globally unique timestamp**
- The **timestamp is passed with each object access**
- The servers are jointly responsible for ensuring serial equivalence:
  - that is **if T access an object before U, then T is before U at all objects**

Optimistic concurrency control revisited

For distributed transactions to work:

1) **Validation takes place in phase 1 of 2PC protocol at each server**

2) **Transactions use a globally unique order for validation**

What if objects in different servers are replicas for increased availability

inconsistency: 若Client1在B上修改→未广播到A→Client2读到不一样的值
propagate

Client 1:	Client 2:
$setBalance_B(x,1)$	
$setBalance_A(y,2)$	
	$getBalance_A(y) \rightarrow 2$
	$getBalance_A(x) \rightarrow 0$

Server

Initial balance of x and y is $0

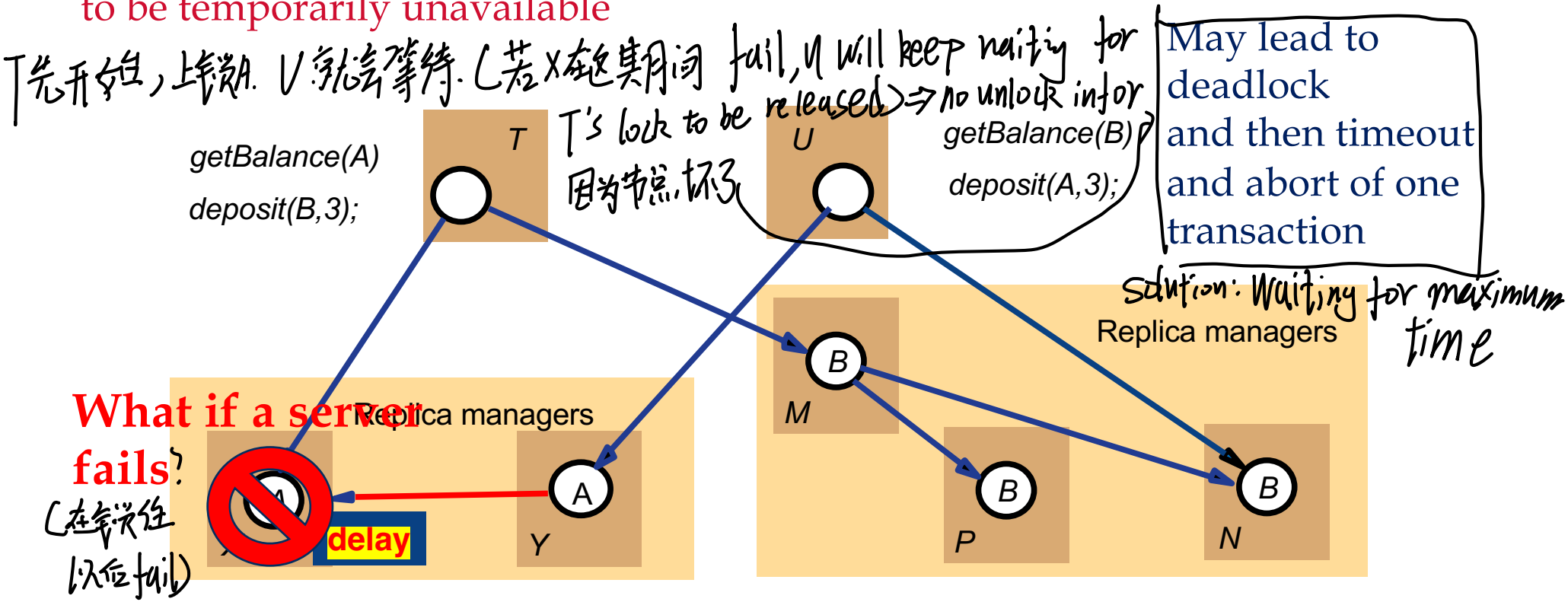
The **behaviour above cannot occur if A and B did not exist** (that is, if we had only one server)

Transactions with replicated data

- the effect of transactions on replicated objects should be the **same as if they had been performed one at a time on a single set of objects**
- this property is called ***one-copy serializability***
- **If all servers are available then no issue – but what if some servers are not available?**

Lets build the solution step by step

The **available copies replication** scheme is designed to allow some servers to be temporarily unavailable



At *X*, *T* has read *A* and has locked it. Therefore *U*'s *deposit* is delayed until *T* finishes. Normally, this leads to good concurrency control only if the servers do not fail….

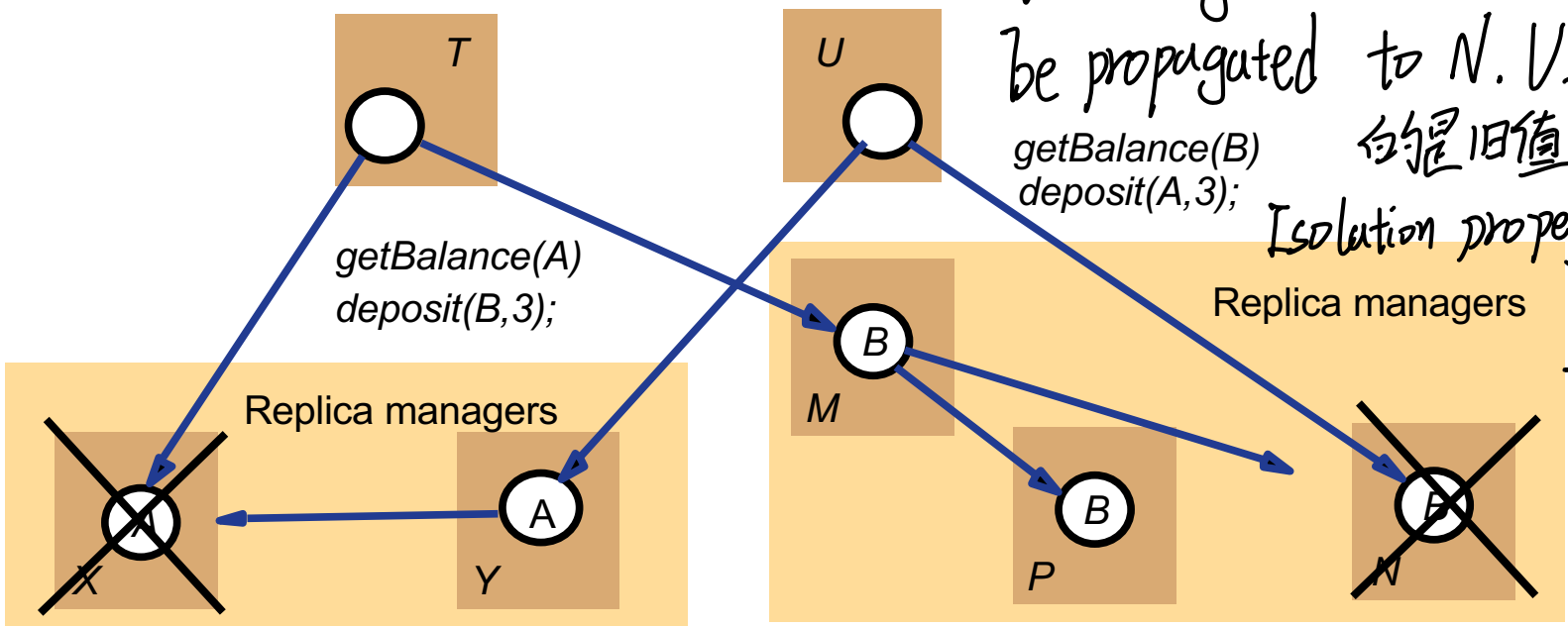
X坏了，为什么不能转而用其他好的？=x.?

Cannot the other servers simply be used?

assume that *X* fails just after *T* has performed *getBalance*

and *N* failing just after *U* has performed *getBalance*

按serilizabity：X先来 the change of B on M will not be propagated to N. U再读N的是旧值 (incosistency Isolation propetry will not be preserved.)



therefore *T*'s deposit will be performed at *M* and *P* (all available)

and *U*'s deposit will be performed at *Y* (all available)

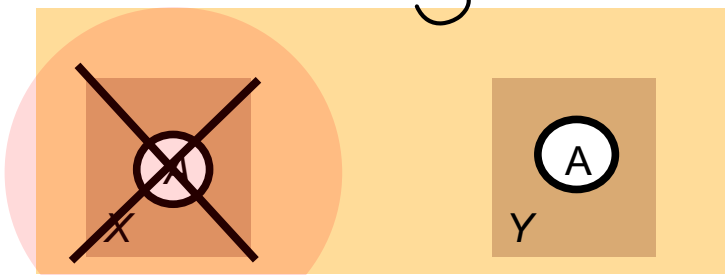
NOT GOOD!!

Available copies replication rule

Before a transaction commits, it checks for failed and available servers it has contacted, the set should not change during execution:

- E.g., *T* would check if X is still available among others.
- We said X fails before *T*'s *deposit*, in which case, T would have to abort.
- Thus no harm can come from this execution now.

① T has (x, y) during execution for A → if one node x or y failed then T should abort.

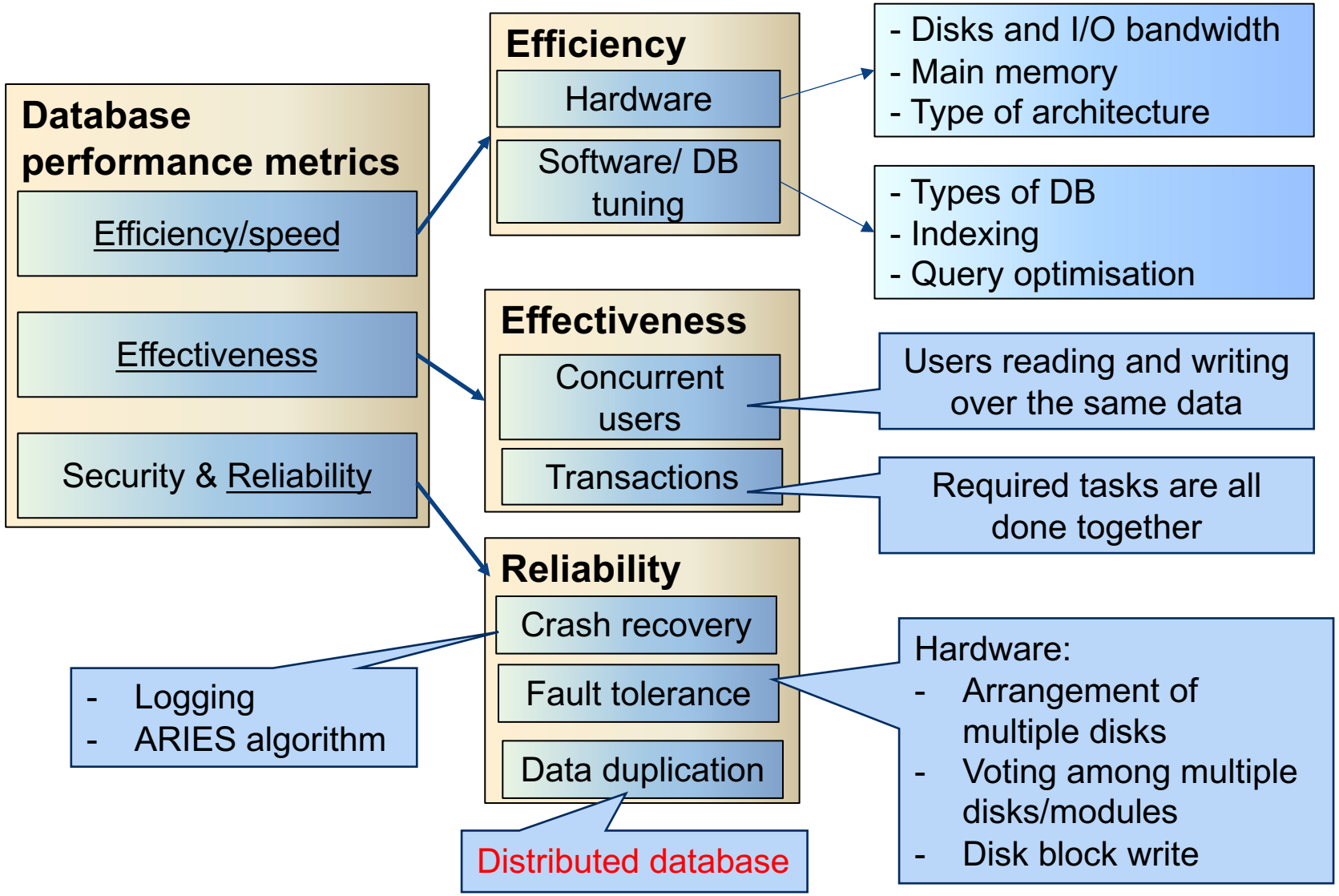


② T(x, y, z) → A

↓

node z is recovered during execution of T, T abort as well

Core Concepts of Database management system



CAP Theorem

Some slides from Mohammad Hammoud,
Dong Wang

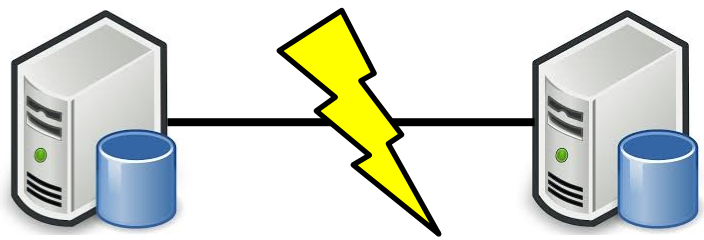
The CAP Theorem

- The limitations of distributed databases can be described in the so called the CAP theorem
  - **C**onsistency: every node always sees the same data at any given instance (i.e., strict consistency)
  - **A**vailability: the system continues to operate, even if nodes crash, or some hardware or software parts are down due to upgrades (still get data read, write operations for other nodes)
  - **P**artition Tolerance: the system continues to operate in the presence of network partitions

CAP theorem: any distributed database with shared data, can have *at most two* of the three desirable properties, C, A or P

The CAP Theorem (*Cont'd*)

- Let us assume two nodes on opposite sides of a network partition:



- Availability + Partition Tolerance forfeit Consistency as changes in place cannot be propagated when the system is portioned.

- Consistency + Partition Tolerance entails that one side of the partition must act as if it is unavailable, thus forfeiting Availability

↓ 读写都走一个 node.

- Consistency + Availability is only possible if there is no network partition, thereby forfeiting Partition Tolerance

Large-Scale Databases

- When companies such as Google and Amazon were designing large-scale databases, 24/7 Availability was a key
  - A few minutes of downtime means lost revenue

- With databases in 1000s of machines, the likelihood of a node or a network failure increases tremendously

- Therefore, in order to have strong guarantees on Availability and Partition Tolerance, they had to sacrifice "strict" Consistency (*implied by the CAP theorem*)

企业：网断了+nodes down → 剩的仍然工作，提供服务 ⇒ Availability + partition tolerance consistency 没了.

Types of Consistency

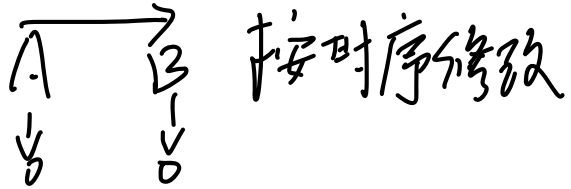
- Strong Consistency
  - After the update completes, **any subsequent access** will return the **same** updated value.
- Weak Consistency
  - It is **not guaranteed** that subsequent accesses will return the updated value.
- **Eventual Consistency**
  - Specific form of weak consistency
  - It is guaranteed that if **no new updates** are made to object, **eventually** all accesses will return the last updated value (e.g., *propagate updates to replicas in a lazy fashion*)

Eventual Consistency Variations

- Causal consistency
  - Processes that have causal relationship will see consistent data
- Read-your-write consistency
  - A process always accesses the data item after it's update operation and never sees an older value
- Session consistency
  - As long as session exists, system guarantees read-your-write consistency
  - Guarantees do not overlap sessions

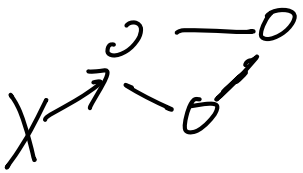
Eventual Consistency Variations

- Monotonic read consistency
  - If a process has seen a particular value of data item, any subsequent processes will never return any previous values
- Monotonic write consistency
  - The system guarantees to serialize the writes by the *same* process
- In practice
  - A number of these properties can be combined
  - Monotonic reads and read-your-writes are most desirable



Eventual Consistency
- A Facebook Example

- Bob finds an interesting story and shares with Alice by posting on her Facebook wall
- Bob asks Alice to check it out



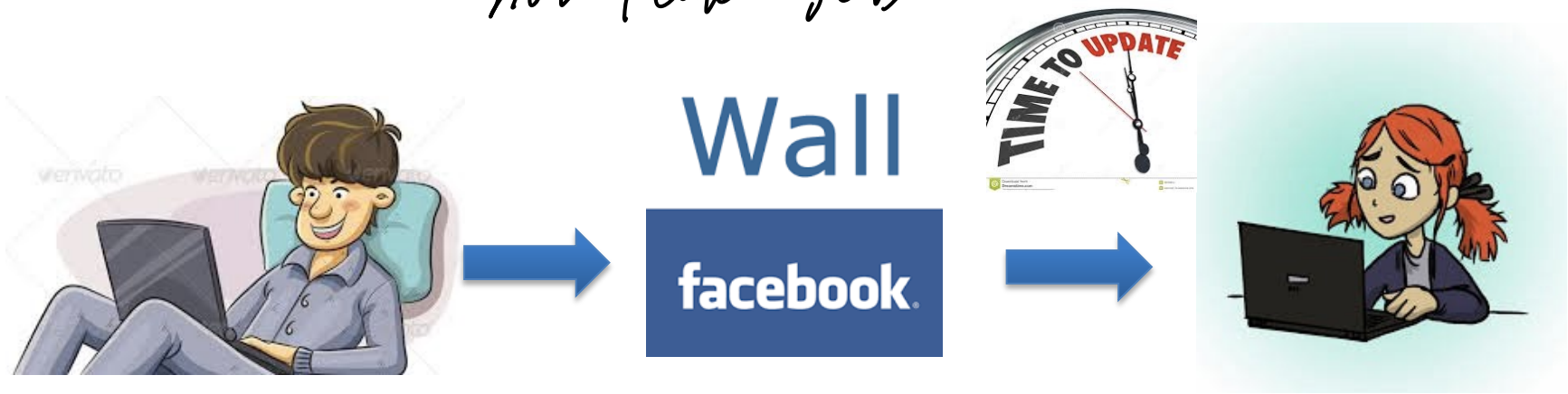
- Alice logs in her account, checks her Facebook wall but finds:
  - **Nothing is there!**

男的发post 女的当时看不见(更新其他节点需要时间)

Eventual Consistency
- A Facebook Example

- Bob tells Alice to wait a bit and check out later
- Alice waits for a minute or so and checks back:
  - **She finds the story Bob shared with her!**

二过3 就看到 3. Face book中，我朋友看到的新闻 我看不到，过3一就才看到
二eventual consistency. (the node 正 access from his not updated yet)



Eventual Consistency - A Facebook Example

- Reason: it is possible because Facebook uses an **eventual consistent model**
- Why Facebook chooses eventual consistent model over the strong consistent one?
  - Facebook has more than 1 billion active users
  - It is non-trivial to efficiently and reliably store the huge amount of data generated at any given time
  - Eventual consistent model offers the option to **reduce the load and improve availability**

Strong consistency not always needed.

preference of the availability over strong consistency 二者只能在使用同时，

若实现 strong consistency，若干人同时等待更新 annoying not desirable 体验差不好

for maintenance.

(unlike bank, for social media, a bit less recent data is acceptable)

Stock market . bank ⇒ Strong consistency is more desirable

Eventual Consistency - A Dropbox Example

- Dropbox enabled immediate consistency via synchronization in many cases.
- However, what happens in case of a network partition?

Eventual Consistency - A Dropbox Example

- Let's do a simple experiment here:
  - Open a file in your drop box
  - Disable your network connection (e.g., WiFi, 4G)
  - Try to edit the file in the drop box: can you do that?
  - Re-enable your network connection: what happens to your dropbox folder?

Eventual Consistency - A Dropbox Example

- Dropbox embraces eventual consistency:
  - Immediate consistency is impossible in case of a network partition
  - Users will feel bad if their word documents freeze each time they hit Ctrl+S , simply due to the large latency to update all devices across WAN
  - Dropbox is oriented to **personal syncing**, not on collaboration, so it is not a real limitation.

Eventual Consistency
- An ATM Example

- In design of automated teller machine (ATM):
  - Strong consistency appear to be a nature choice
  - However, in practice, **A beats C**
  - Higher availability means **higher revenue**
  - ATM will allow you to withdraw money *even if the machine is partitioned from the network*
  - However, it puts **a limit** on the amount of withdraw (e.g., $200)
  - The bank might also charge you a fee when a overdraft happens

尽管银行追求 strong consistency, However,

In bank personal account, strong consistency can be relaxed, 在网络分区发生时，银行也可以让你取钱（不满足保证 consistency）（限额/罚金）⇒ 为了服务客户（availability + Partition tolerance）⇒ although not consistent

Small risk (limit amount) ⇑ weak consistency

文件修改的例子：(AP)

availability + partition tolerance：当 network partition 发生 ⇒ 保证 availability (可操作文件修改) 但仅对你可见 ⇒ 图标变蓝 ⇒ 冲突要再对齐 (但不同冲突的值不同) ⇒ available.

但要保证 consistency ⇒ 当 network partition 发生时 ⇒ 就不能更新文件 不能操作文件

对 personal user 的例：more desirable 是不同用户提供不同服务.

⇒ still be able to update and access file when partition happen; (no freeze on my file).

Airline reservation：预订机票 ⇒ 强调 availability + consistency (等待) 取决于 余票量多少

(CA)

⇒ 很多座位余量：(AP) Weak consistency ⇒ dose matter when we see 40 tickets left or 349 left. ⇒ out of date data is acceptable.

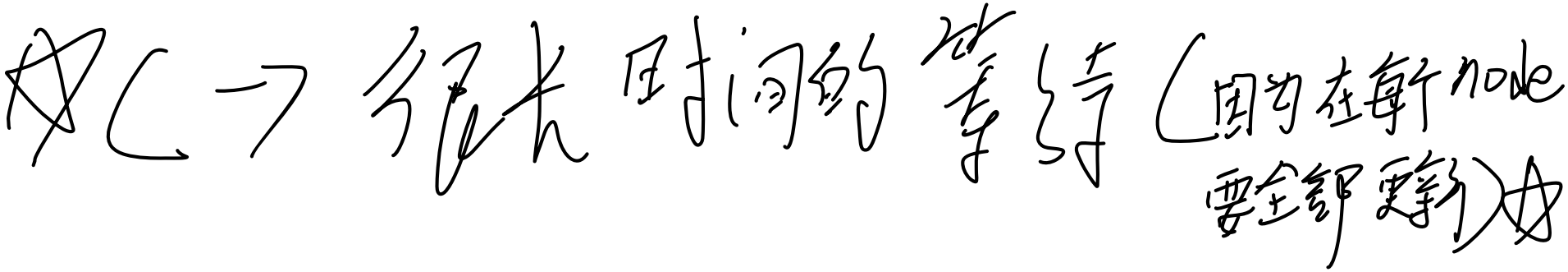
not strong consistency, ⇒ keep availability during network partition.

即使网络故障 用户也可下单 ⇒ 无需等待系统 maintain (in strong consistency) network.

⇒ 锁步 ⇒ strong consistency. (AP) ⇒ more desirable to see accurate data ⇒ ensure the plane is not overbooked. (over availability)

Dynamic Tradeoff between **C** and **A**

- An airline reservation system:
  - When most of seats are available: it is ok to rely on somewhat out-of-date data, availability is more critical
  - When the plane is close to be filled: it needs more accurate data to ensure the plane is not overbooked, consistency is more critical



Heterogeneity: Segmenting **C** and **A**

- No single uniform requirement (用户、服务 etc.)
  - Some aspects require strong consistency
  - Others require high availability
- Segment the system into different components
  - Each provides different types of guarantees
- Overall guarantees neither consistency nor availability
  - Each part of the service gets exactly what it needs
- Can be partitioned along different dimensions

Partitioning ⇒ 按偏好分区，有的EAL，有的AP，有的CP。

不同区有不同程度的 consistency、availability or tolerance

shopping cart high availability, responsive,

① web上的 products description、images : should be available, partition tolerance (无需等), 当admin修改时 用户看不见新的 (consistency) 但是可以接受 (短时间)

② 对 shopping cart: add products to cart (high availability) 不想等 (no consistency)

加购结帐时 有的 item not available (卖完) → OK, just remove from cart.

(inconsistency in terms #of available products) ⇐⇒ trade off between A vs C

在 locks 中: 商品数减少 (有用户买) ⇒ 下单不了，卖完了，尽管已加购。
⇒ 是一个原因. (只是不同角度)

{ locks: consistency of data.
CAP: shopping cart focus on more availability, not always the updated info (most recent) on the # of items available.

③ for checkout: { checkout - shipping record - financial billing } involved: paying money: more updated info on: Whether the item is available for user ⇒ so that I can make payment.

所以 OK for us to wait a little bit ⇒ the maintain consistency and safety

lockship: exclusive lock for checkout. Strong consistency over A when network partition happens

Partitioning Examples

Data Partitioning

- Different data may require different consistency and availability
- Example:
  - Shopping cart: high availability, responsive, can sometimes suffer anomalies
  - Product information need to be available, slight variation in inventory is sufferable
  - Checkout, billing, shipping records must be consistent

availability：网络断后用户仍然可access node只不过不同分区数据可能不同，有的不是最新的。

What if there are no partitions?

no network partition

- Tradeoff between **Consistency** and **Latency**:
- Caused by the **possibility of failure** in distributed systems
  - High availability -> replicate data -> consistency problem
- Basic idea:
  - Availability and latency are arguably **the same thing**: unavailable -> extreme high latency (即使system是available，某些节点down，也要等待时间，↓ Latency.)
  - Achieving different levels of consistency/availability takes different amount of time

Trading-Off Consistency

- Maintaining consistency should balance between the strictness of consistency versus availability/scalability
  - Good-enough consistency *depends on your application*

Trading-Off Consistency

- Maintaining consistency should balance between the strictness of consistency versus availability/scalability
  - Good-enough consistency *depends on your application*

Easier to implement,
and is efficient

Generally hard to implement,
and is inefficient

The BASE Properties

- The CAP theorem proves that it is impossible to guarantee strict Consistency and Availability while being able to tolerate network partitions

- This resulted in databases with relaxed ACID guarantees
  分区与有订票的例子 => relaxed ACID consistency is somewhat compromised.
- In particular, such databases apply the BASE properties: (现代数据库都是这个)
  - **B**asically **A**vailable: the system guarantees Availability
  - **S**oft-State: the state of the system may change over time
  - **E**ventual Consistency: the system will *eventually* become consistent

partition tolerance.
即使分区发生 => 不同桶的能提供服务.

why modern database rely on BASE rather than ACID:

① EC: ∵ large scale database are distributed ⇒ not possible to guarantee network tolerance ⇒ weaker consistency is applied to database. When system becomes eventually consistent, the all nodes are updated to the recend data, but it takes some time to propagate that info.

② soft-state.

State of system may change of time (whether focus more on consistency, or ignor constraint on data) State of system depends on application prefereence.

CAP -> PACELC

- A more complete description of the space of potential tradeoffs for distributed system:
  - If there is a **partition (P)**, how does the system trade off **availability and consistency (A and C)**; **else (E)**, when the system is running normally in the absence of partitions, how does the system trade off **latency (L) and consistency (C)**?

Abadi, Daniel J. "Consistency tradeoffs in modern distributed database system design." Computer-IEEE Computer Magazine 45.2 (2012): 37.

consistency ↑ → latency ↑

Examples

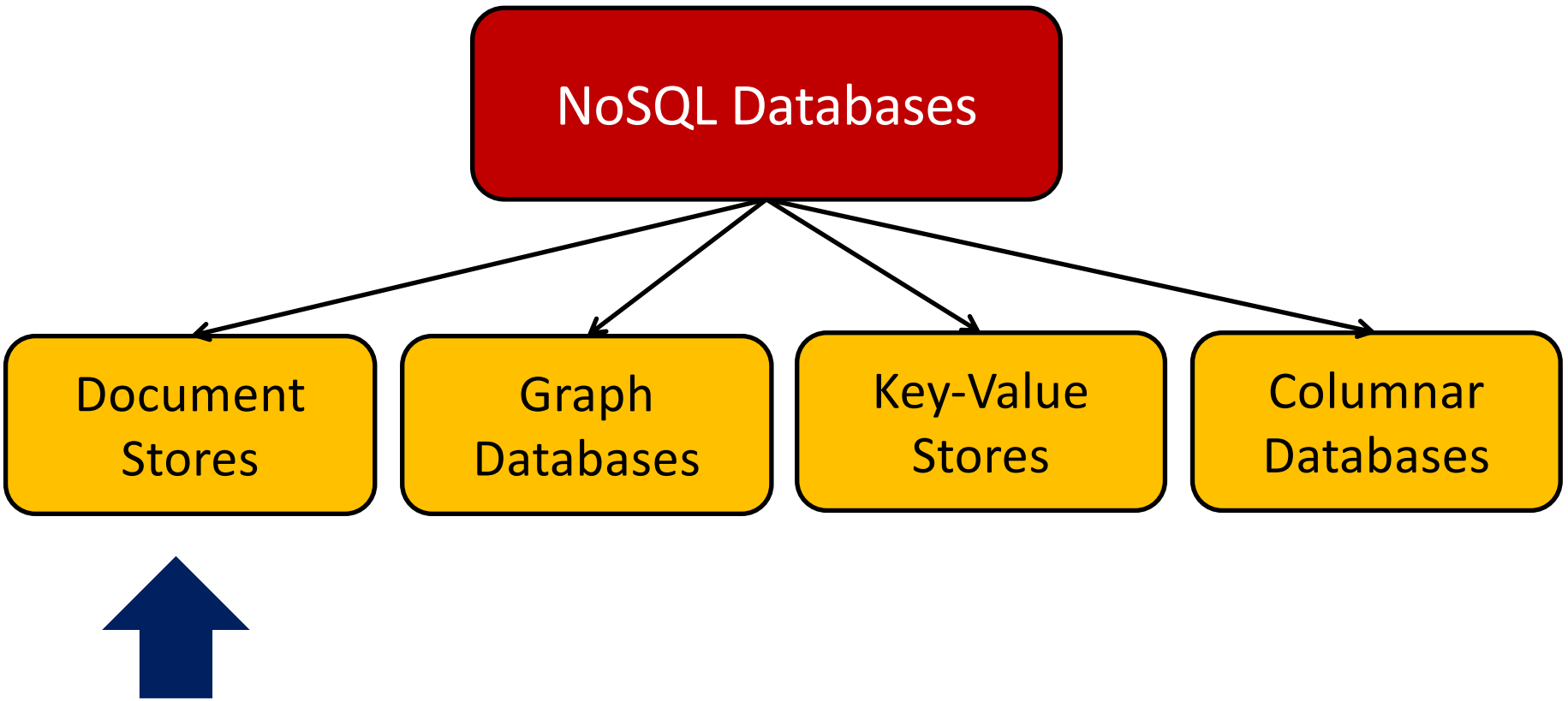
Partition tolerance always exists ⇒ network partion 不可避免

- **PA/EL Systems:** Give up both Cs for availability and lower latency : system prefer to answer of access data with a shorter time by compromising consistency.
  - Dynamo, Cassandra, Riak
- **PC/EC Systems:** Refuse to give up consistency and pay the cost of availability and latency
  - BigTable, Hbase, VoltDB/H-Store
- **PA/EC Systems:** Give up consistency when a partition happens and keep consistency in normal operations
  - MongoDB
- **PC/EL System:** Keep consistency if a partition occurs but gives up consistency for latency in normal operations
  - Yahoo! PNUTS

Types of NoSQL Databases

BASE instead of ACID

- Here is a limited taxonomy of NoSQL databases:

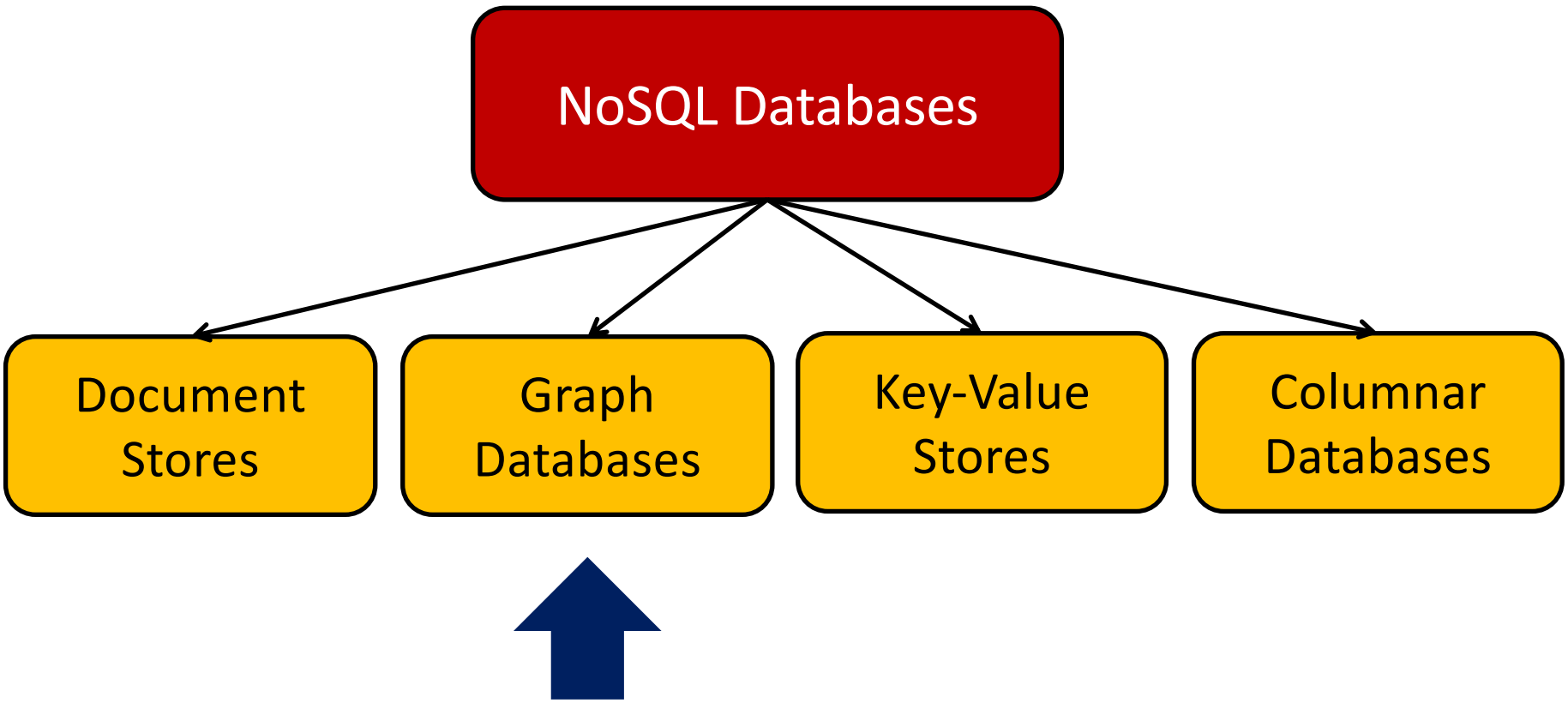


Document Stores

- Documents are stored in some standard format or encoding (e.g., XML, JSON, PDF or Office Documents)
  - These are typically referred to as Binary Large Objects (BLOBs)
- Documents can be indexed
  - This allows document stores to outperform traditional file systems
- E.g., MongoDB and CouchDB (both can be queried using MapReduce)

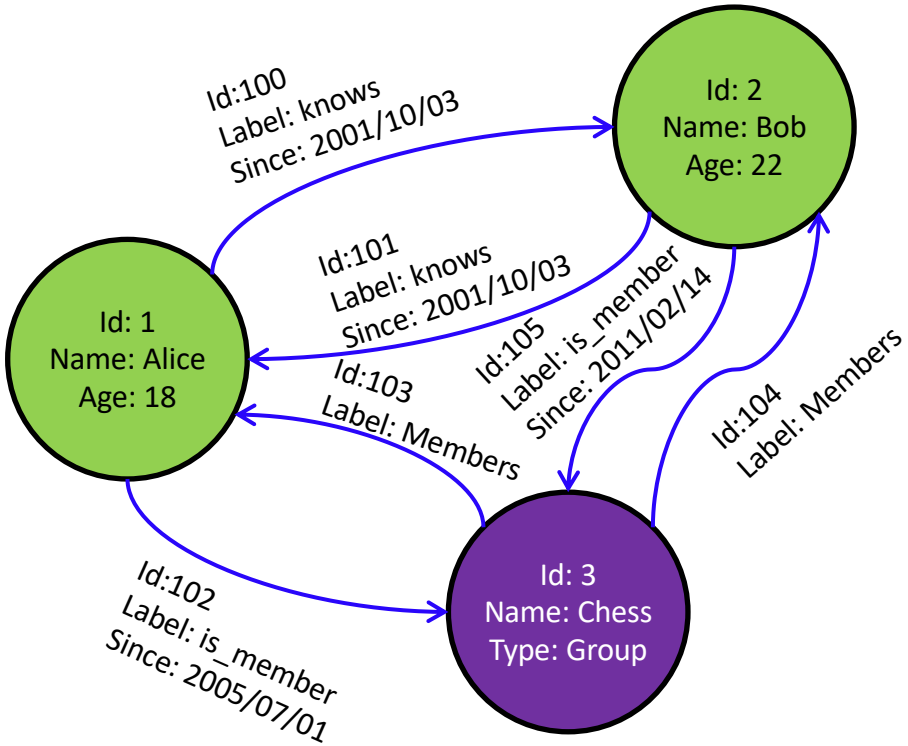
Types of NoSQL Databases

- Here is a limited taxonomy of NoSQL databases:



Graph Databases

- Data are represented as vertices and edges

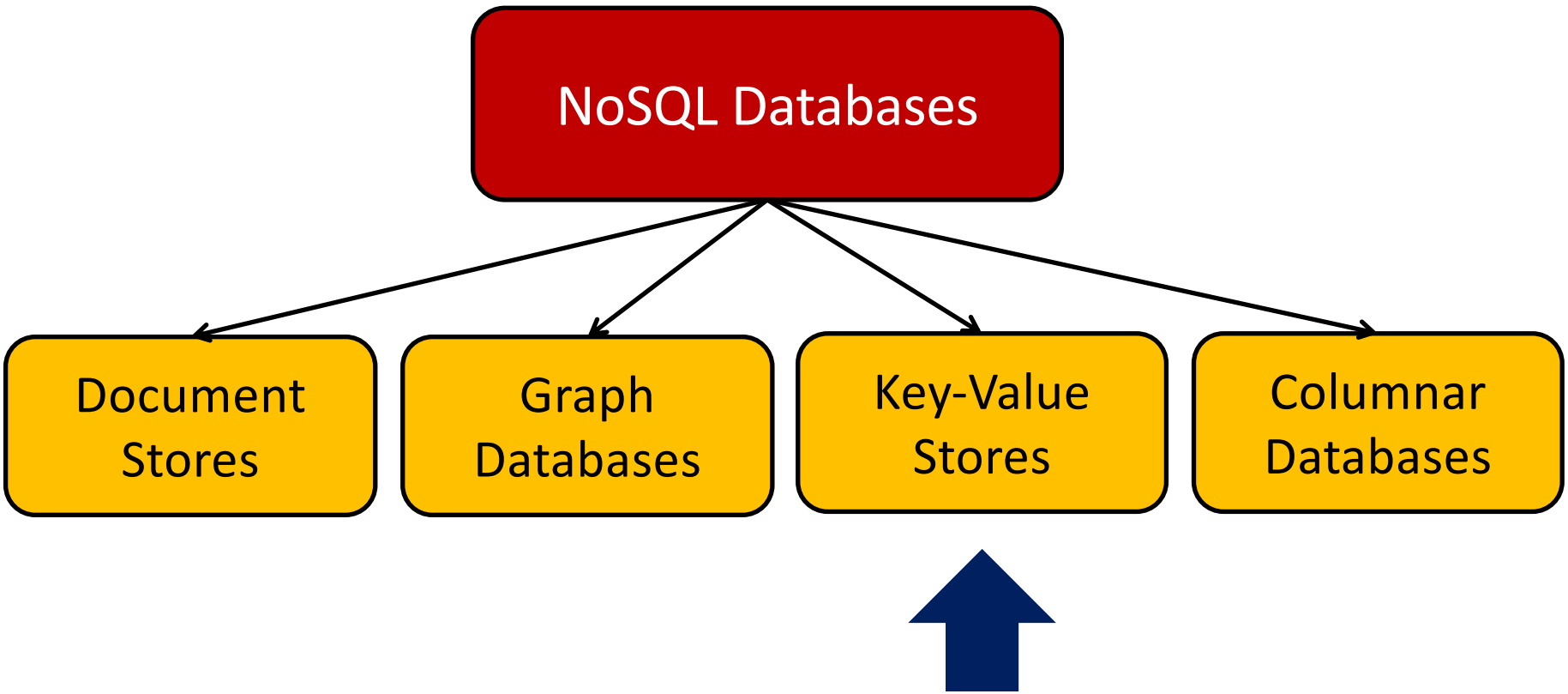


- Graph databases are powerful for graph-like queries (e.g., find the shortest path between two elements)

- E.g., Neo4j and VertexDB

Types of NoSQL Databases

- Here is a limited taxonomy of NoSQL databases:

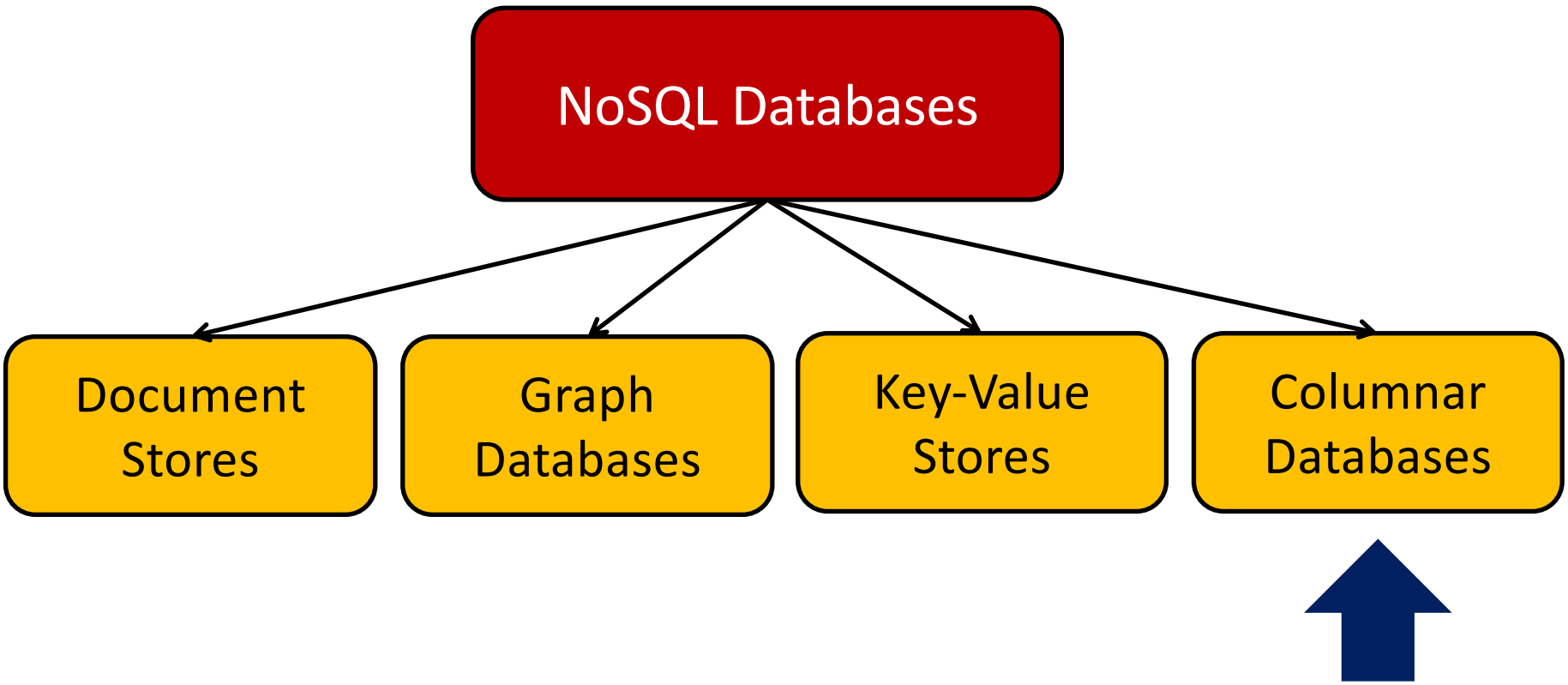


Key-Value Stores

- Keys are mapped to (possibly) more complex value (e.g., lists)
- Keys can be stored in a hash table and can be distributed easily
- Such stores typically support regular CRUD (create, read, update, and delete) operations
  - That is, no joins and aggregate functions
- E.g., Amazon DynamoDB and Apache Cassandra

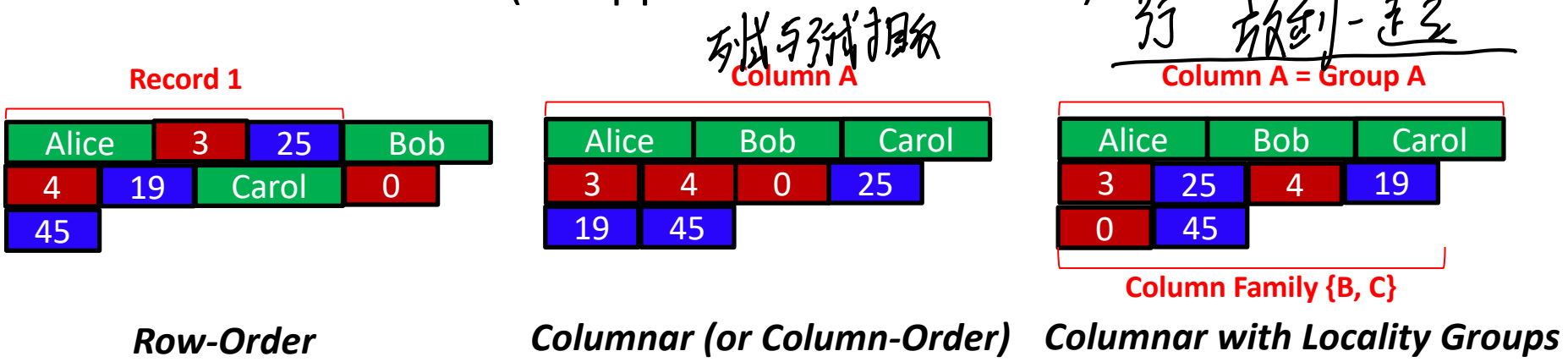
Types of NoSQL Databases

- Here is a limited taxonomy of NoSQL databases:



Columnar Databases

- Columnar databases are a hybrid of RDBMSs and Key-Value stores
  - Values are stored in groups of zero or more columns, but in Column-Order (as opposed to Row-Order)



Row-Order ***Columnar (or Column-Order)*** ***Columnar with Locality Groups***

  - Values are queried by matching keys

- E.g., HBase and Vertica

Summary

- The *CAP theorem* states that any distributed database with shared data can have at most two of the three desirable properties:
  - **C**onsistency
  - **A**vailability
  - **P**artition Tolerance

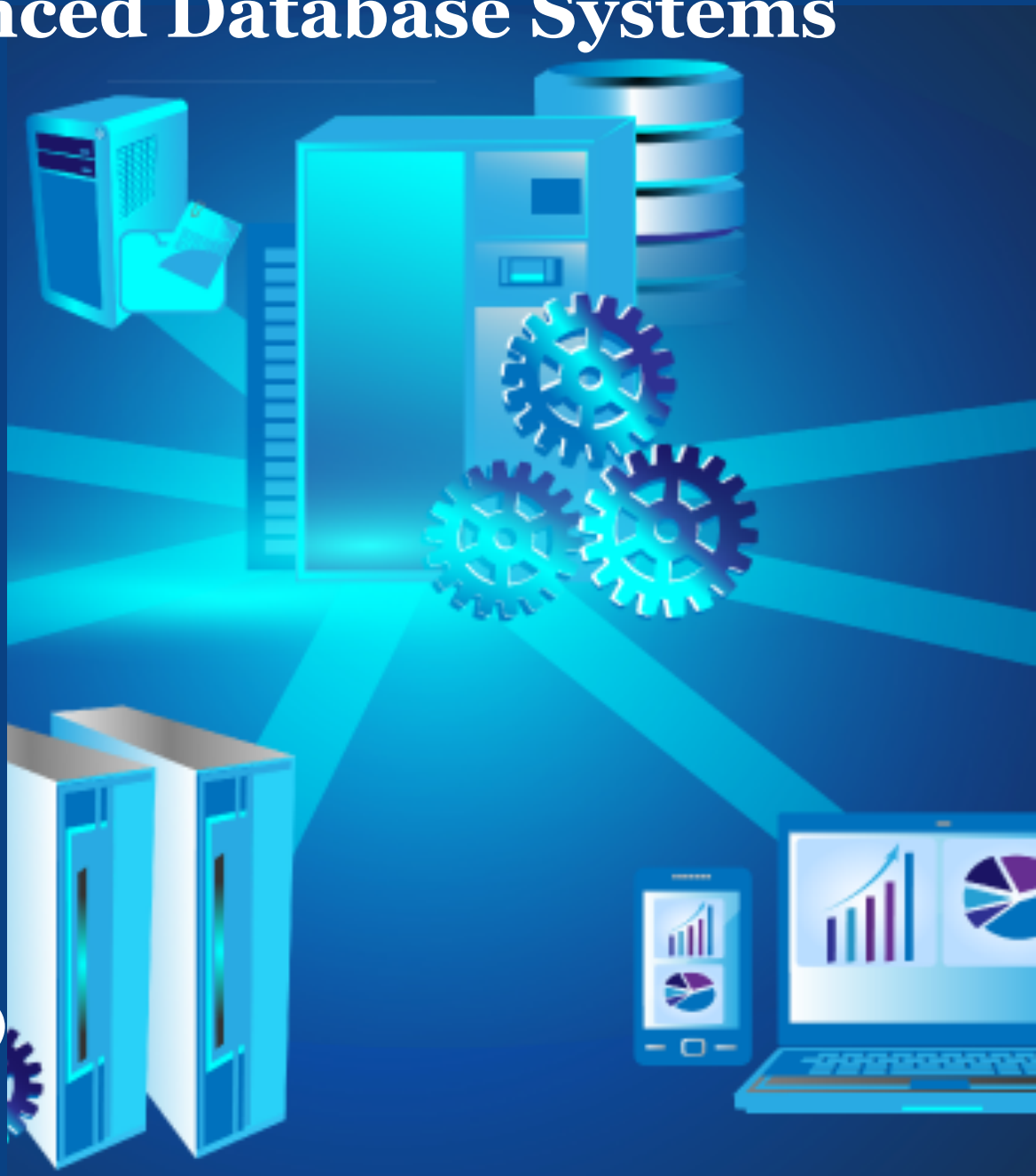
- The CAP theorem leads to various designs of databases with *relaxed* ACID guarantees

Summary (*Cont'd*)

- *NoSQL* (or *Not-Only-SQL*) databases follow the *BASE properties*:
  - **B**asically **A**vailable
  - **S**oft-State
  - **E**ventual Consistency

THE UNIVERSITY OF MELBOURNE

COMP90050 Advanced Database Systems



Semester 2, 2024

Lecturer: Farhana Choudhury (PhD)

Live lecture – Week 11

Today's topics

- Activities on distributed databases
- Q/A
  - Project, final exam, quiz questions

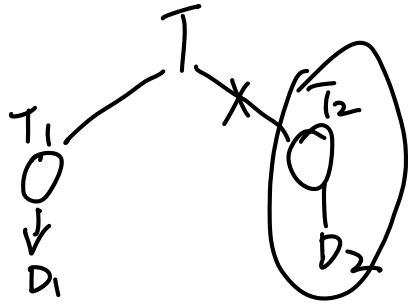
DBMSs in the era of Networking

For distributed transaction processing:

- Atomicity: two-phase commit protocol

Atomicity in distributed DB

In a distributed DB, a transaction T is being executed as T1 in node D1 and as T2 in node D2 (T1 and T2 are parts of T). Both T1 and T2 successfully finished their operations in their own nodes, but D2 lost connection with the coordinator, and couldn't send the 'yes' vote in phase 1. Can T successfully commit?



everything in here down completely still

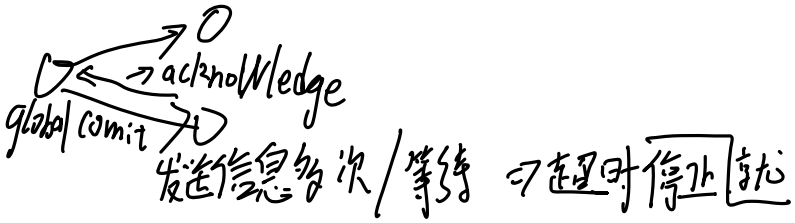
Coordinator has no way to know whether it is successful or not.

Time for a poll –
Pollev.com/farhanachoud585

To maintain Atomicity, both T1 and T2 need to be completed because overall its one transaction

So even if T1 commited, both of T1 and T2

will be aborted to maintain atomicity.
信息传递时她们也无法 commit.



DBMSs in the era of Networking

For distributed transaction processing:

- Atomicity: two-phase commit protocol
- Concurrency:
  - Lock release
  - Timestamping
  - Optimistic locking

Concurrency in distributed DB

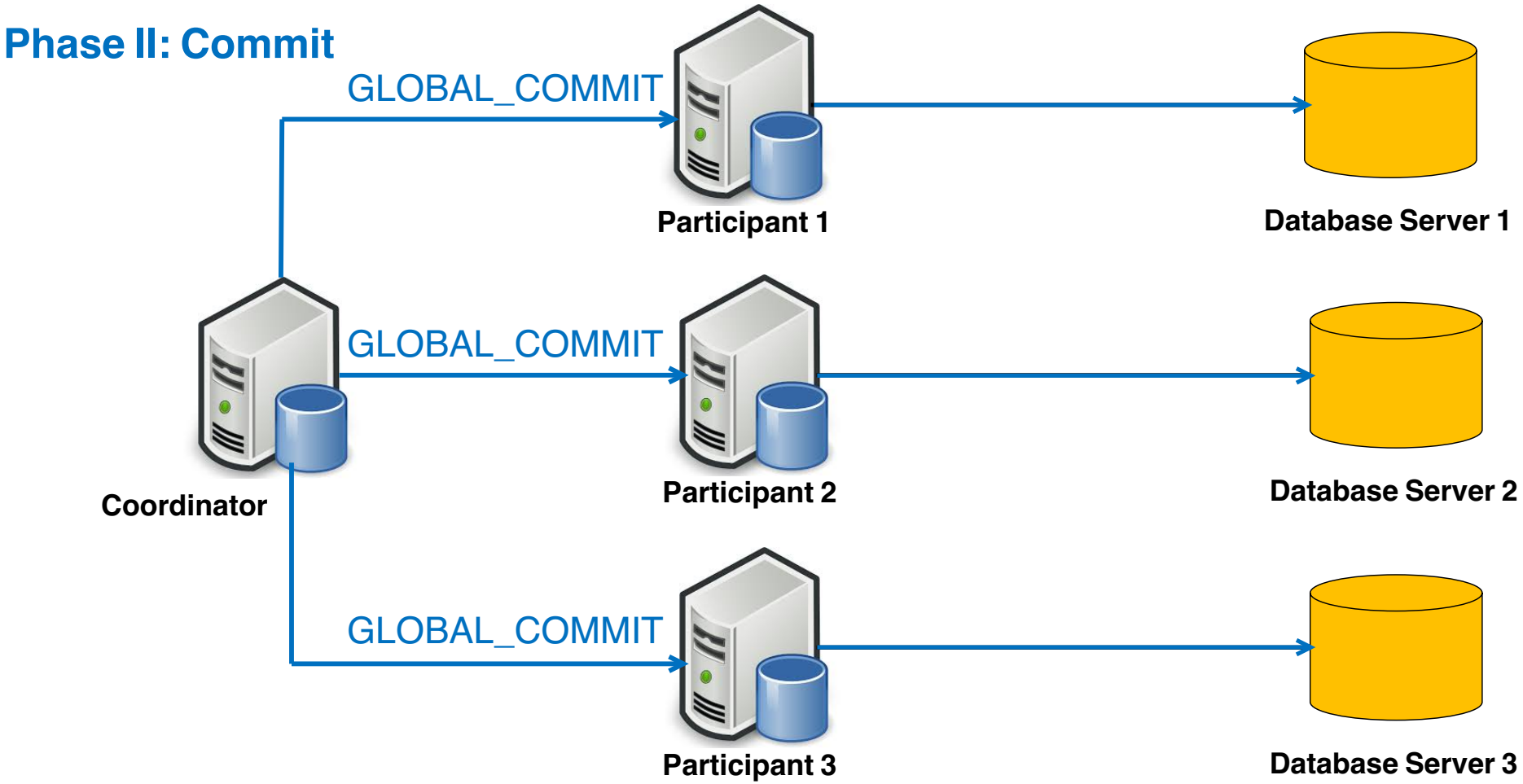
A local lock manager cannot release any locks until it knows that the transaction has been committed or aborted at all the servers involved in the transaction.

When/how will a node know when to release the locks of a transaction?

when global_commit sent, it means that coordinator tells the participate: others have successfully finished their part of the transaction; if they receive that, then each of them can release their own locks for that transaction.

当一个node down了，这个问题如何回答：When any of sub transactions abort, others will also abort as well. So In phase 1, the down node(T_1) sending 'no' message at that time, T_1 releases its lock in phase 1, others will know this in phase 11, and then they will release their locks for T

When/how will a node know when to release the locks of a transaction?



What if a sub-transaction aborts in a node?

DBMSs in the era of Networking

For distributed transaction processing:

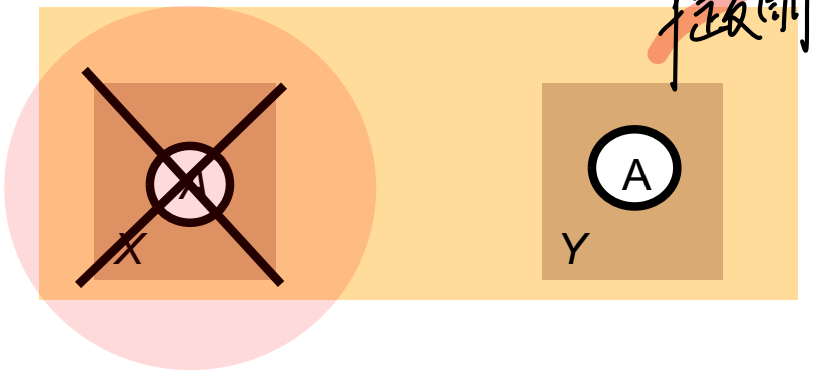
- Atomicity: two-phase commit protocol
- Concurrency:
  - Lock release
  - Timestamping
  - Optimistic locking
- Data replication
  - Rule for one-copy serializability
  - CAP

However, not always desirable because data repetition come with some benefits.

Available copies replication rule

Before a transaction commits, it checks for failed and available servers it has contacted, the set should not change during execution：

- E.g., *T* would check if X is still available among others.
- We said X fails before *T*'s *deposit*, in which case, T would have to abort.
- Thus no harm can come from this execution now.



Available copies replication rule

In a distributed DB, a transaction T only needs to read an object o1. o1 is replicated in the nodes D1, D2, and D3 of this distributed system. There is another node D4, but D4 does not store any copies of o1.

When T started its execution, the nodes D1, D2 are available, and the other nodes are unavailable. When T commits, the nodes D1, D2, D4 are available and D3 is unavailable. Can T successfully commit?

Time for a poll –

Pollev.com/farhanachoud585

The CAP Theorem

- The limitations of distributed databases can be described in the so called the CAP theorem

CAP theorem: any distributed database with shared data, can have *at most two* of the three desirable properties, Consistency, Availability, or Partition tolerance

Different design choices based on CAP theorem: Examples

- Different data may require different consistency and availability
- Example:
  - Shopping cart: high availability, responsive, can sometimes suffer anomalies/inconsistencies
  - Product information need to be available, slight variation in inventory is sufferable
  - Checkout, billing, shipping records must be consistent

different part of application have differet level of consistency.

For Browsing the product: inconsistency / weak consistency on # of items available
; description on items are cuceptable and tolerable.

For checking out, billing: Strong consistency, orther wise we'll lose money
⇒ locking mechanism
more strict locking mechanism V.S relaxed locking mechanism.

Strict: even finish transaction don't release locks (write, making changes).
relaxed locking: ready, 甚至读到不同的值 (for reading more relaxed locks used).

For adding items into shopping cart: Some times we'll find there's items are unavailable,
(so we just remove it from shopping cart -> (available) by the time of checking out,
Someone has bought. (inconsistency)

ready lock 中 relaxed lock: 别的都无法 read 这个 product and allowed to add to
the shopping cart, (inconsistency)

在 distributed system 中 有 node 存着 item count, 如果 item 被别的买家 node has
not updated immediately by coordinator ⇒ We'll also meet the same situation
with ready lock.

Different design choices based on CAP theorem: Examples

More real-life applications/examples where high availability is important, can sometimes tolerate some inconsistency?

For social media Apps, usually there are millions of users, their data are stored across many nodes. The nodes you are reading from maybe not the same node that your friends are reading from. And the new comment hasn't been immediately propagated to all those nodes. So In reality, if we change or edit some post, those changes may not get immediately propagated to everyone ⇒ inconsistency but AP.

Time for a poll –
Pollev.com/farhanachoud585

Flight prices, hotel prices 数据库的机制可以随应用状态数据切换. (emphasize)

inconsistency → 房间多or我先订了 → 但看到的还是 old price ⇒ the new price hasn't been propagated to it. ⇒ 等一会再看就更新了, 但不影响, 可以暂时使用 (waited consistency)

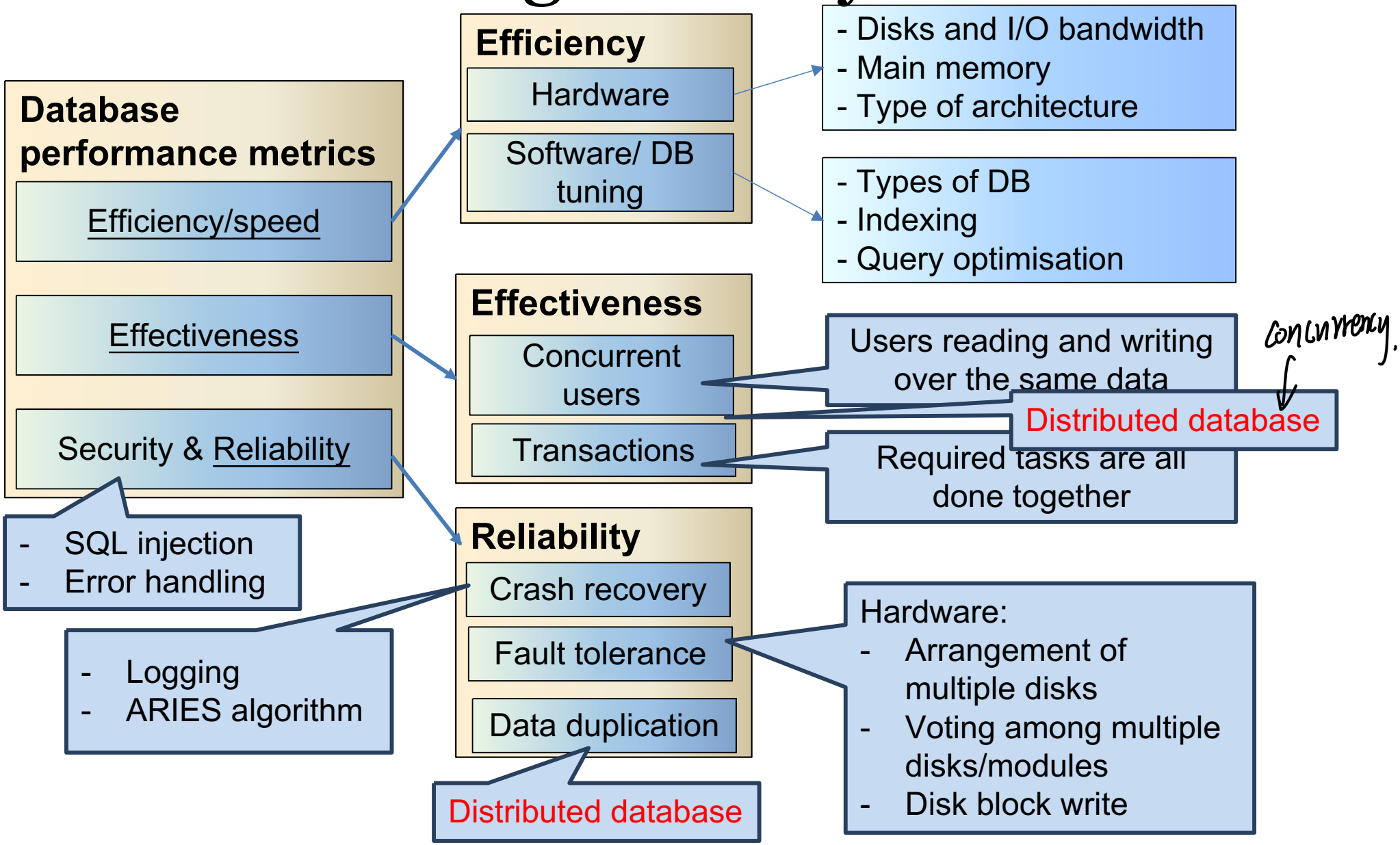
although the updated info would be propagated to all nodes eventually but give some higher priority for some thing.

When room is empty ⇒ It will keep consistency eventually, but it goes down in the priority list.

Inconsistency: 我看见、我朋友看还没见 、(没被propogated).

If rooms have been almost fully booked ⇒ emphasize on consistency

Core Concepts of Database management system



RAID 0 with 3 disks no replica, no additional info.

flushLSN of page LSN.

DPT records the oldest LSN for each page.

Q/A