CAP Theorem

Rao Kotagiri
Some slides from Mohammad Hammoud,
Dong Wang

Types of Data

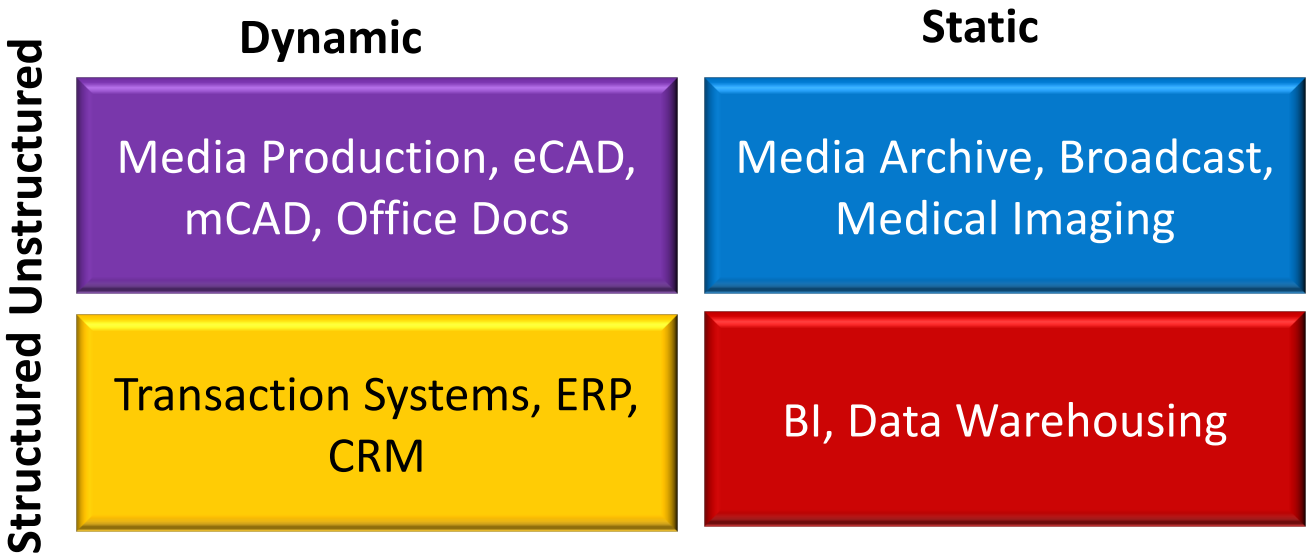
- Data can be broadly classified into four types:
  1. Structured Data:
     - Have a predefined model, which organizes data into a form that is relatively easy to store, process, retrieve and manage
     - E.g., relational data
  2. Unstructured Data:
     - Opposite of structured data
     - E.g., Flat binary files containing text, video or audio
     - Note: data is not completely devoid of a structure (e.g., an audio file may still have an encoding structure and some metadata associated with it)

Types of Data

- Data can be broadly classified into four types:
  3. Dynamic Data:
     - Data that changes relatively frequently
     - E.g., office documents and transactional entries in a financial database
  4. Static Data:
     - Opposite of dynamic data
     - E.g., Medical imaging data from MRI or CT scans

Why Classifying Data?

- Segmenting data into one of the following 4 quadrants can help in designing and developing a pertaining storage solution



- Relational databases are usually used for structured data

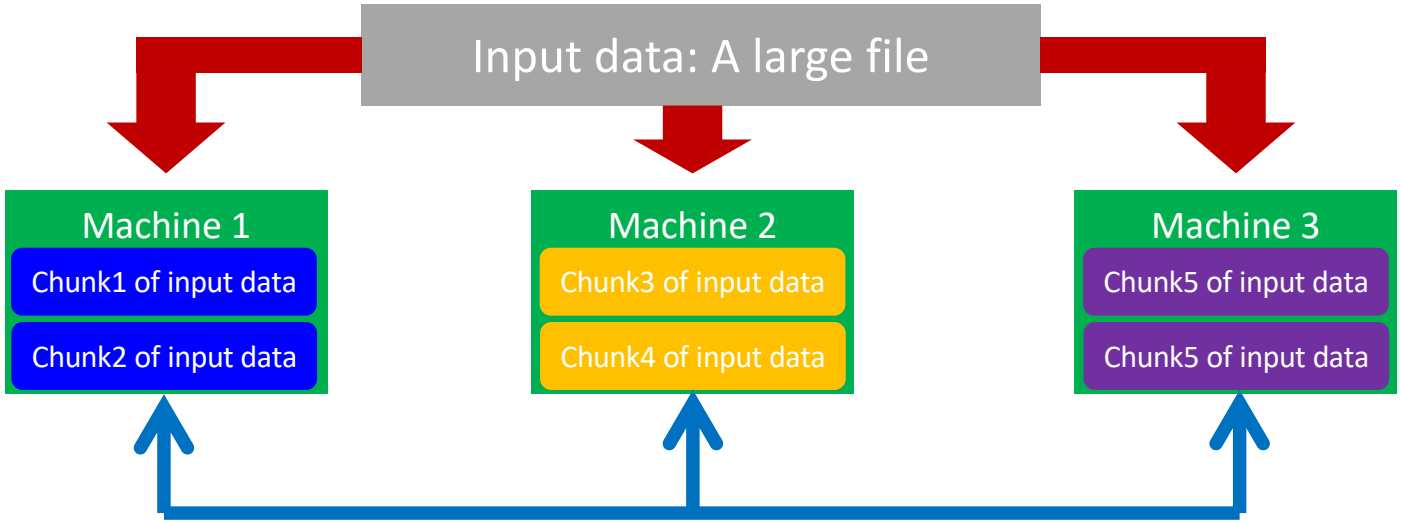
- File systems or *NoSQL databases* can be used for (static), unstructured data (*more on these later*)

Scaling Traditional Databases

- Traditional RDBMSs can be either scaled:
  - Vertically (or Up)
    - Can be achieved by hardware upgrades (e.g., faster CPU, more memory, or larger disk)
    - Limited by the amount of CPU, RAM and disk that can be configured on a single machine
  - Horizontally (or Out)
    - Can be achieved by adding more machines
    - Requires database *sharding* and probably *replication*
    - Limited by the Read-to-Write ratio and communication overhead

Why Sharding Data?

- Data is typically *sharded* (or *striped*) to allow for concurrent/parallel accesses



E.g., Chunks 1, 3 and 5 can be accessed in parallel

Amdahl's Law

- How much faster will a parallel program run?
  - Suppose that the sequential execution of a program takes T_1 time units and the parallel execution on p processors/machines takes T_p time units
  - Suppose that out of the entire execution of the program, ***s*** fraction of it is not parallelizable while ***1-s*** fraction is parallelizable
  - Then the speedup (***Amdahl's formula***):

$$\frac{T_1}{T_p} = \frac{T_1}{(T_1 \times s + T_1 \times \frac{1-s}{p})} = \frac{1}{s + \frac{1-s}{p}}$$

Amdahl's Law: An Example

- Suppose that:
  - 80% of your program can be parallelized
  - 4 machines are used to run your parallel version of the program
- The speedup you can get according to Amdahl's law is:

$$\frac{1}{s+\frac{1-s}{p}} = \frac{1}{0.2+\frac{0.8}{4}} = 2.5 \text{ times}$$

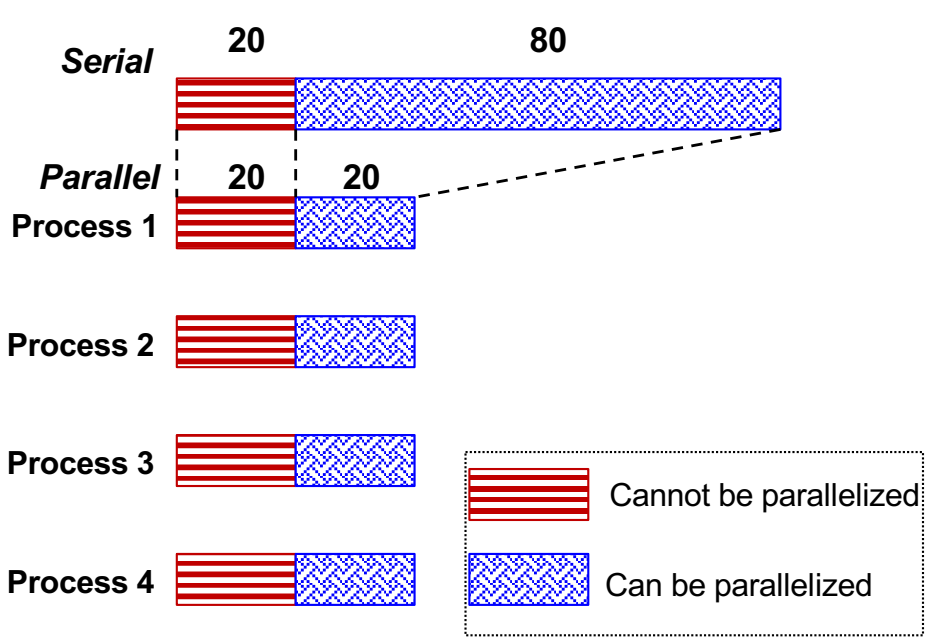
Although you use 4 processors you cannot get a speedup more than 2.5 times!

Efficiency of parallelism = achieved speedup/ maximum possible speedup = 2.5/4 = 0.625 = 62.5%

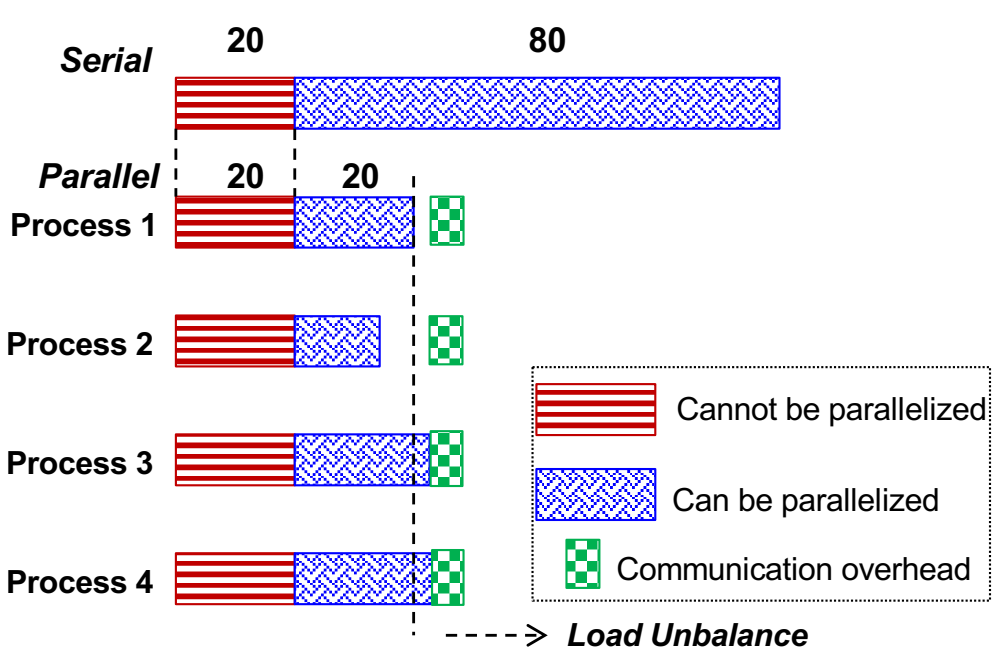
Real Vs. Actual Cases

- Amdahl's argument is too simplified

- In reality, communication overhead and potential workload imbalance exist upon running parallel programs



1. Parallel Speed-up: An Ideal Case



2. Parallel Speed-up: An Actual Case

Some Guidelines

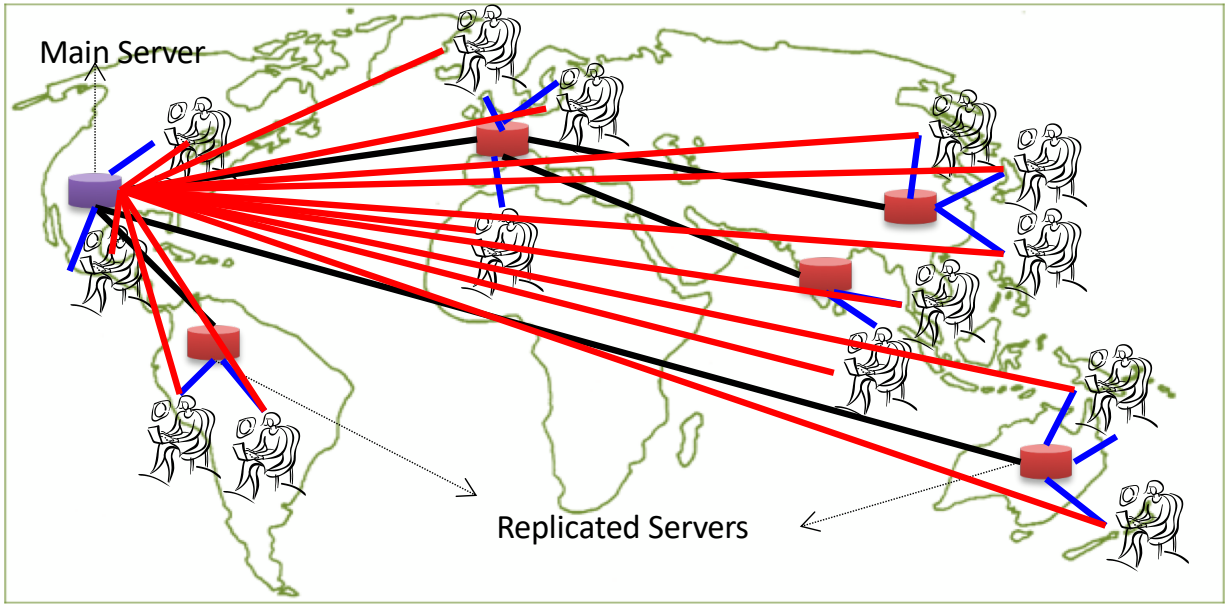
- Here are some guidelines to effectively benefit from parallelization:
  1. Maximize the fraction of your program that can be parallelized
  2. Balance the workload of parallel processes
  3. Minimize the time spent for communication

Why Replicating Data?

- Replicating data across servers helps in:
  - Avoiding performance bottlenecks
  - Avoiding single point of failures
  - And, hence, enhancing scalability and availability

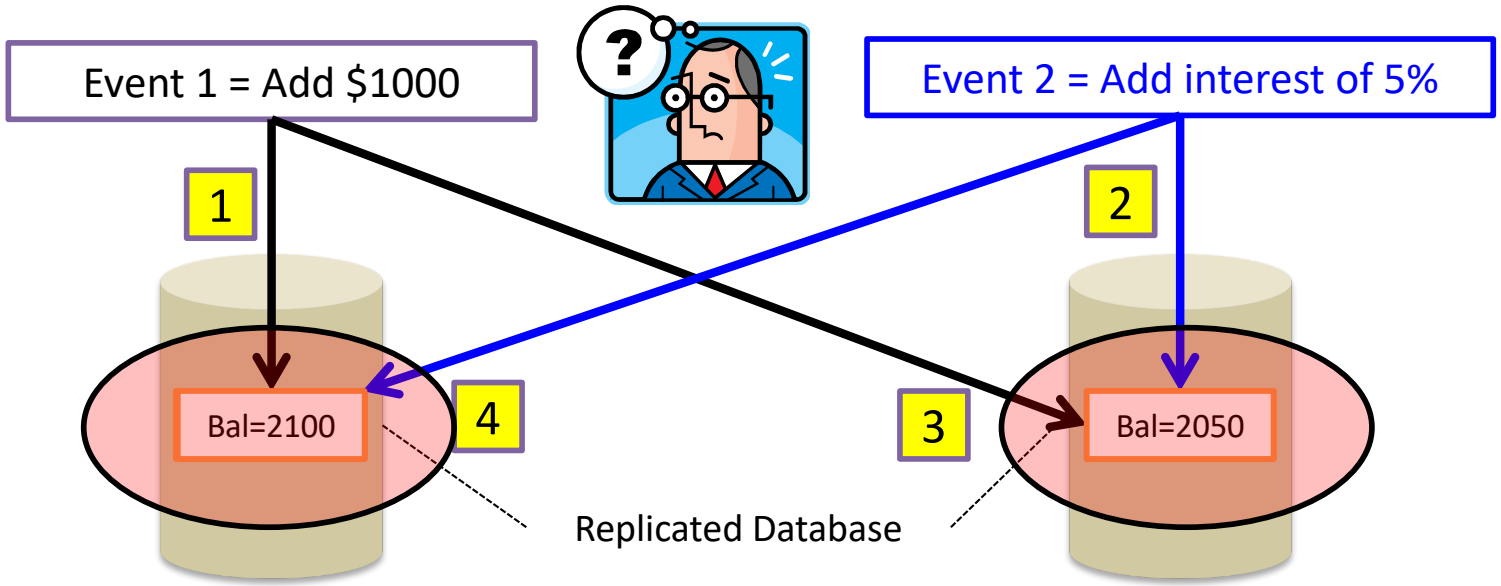
Why Replicating Data?

- Replicating data across servers helps in:
  - Avoiding performance bottlenecks
  - Avoiding single point of failures
  - And, hence, enhancing scalability and availability



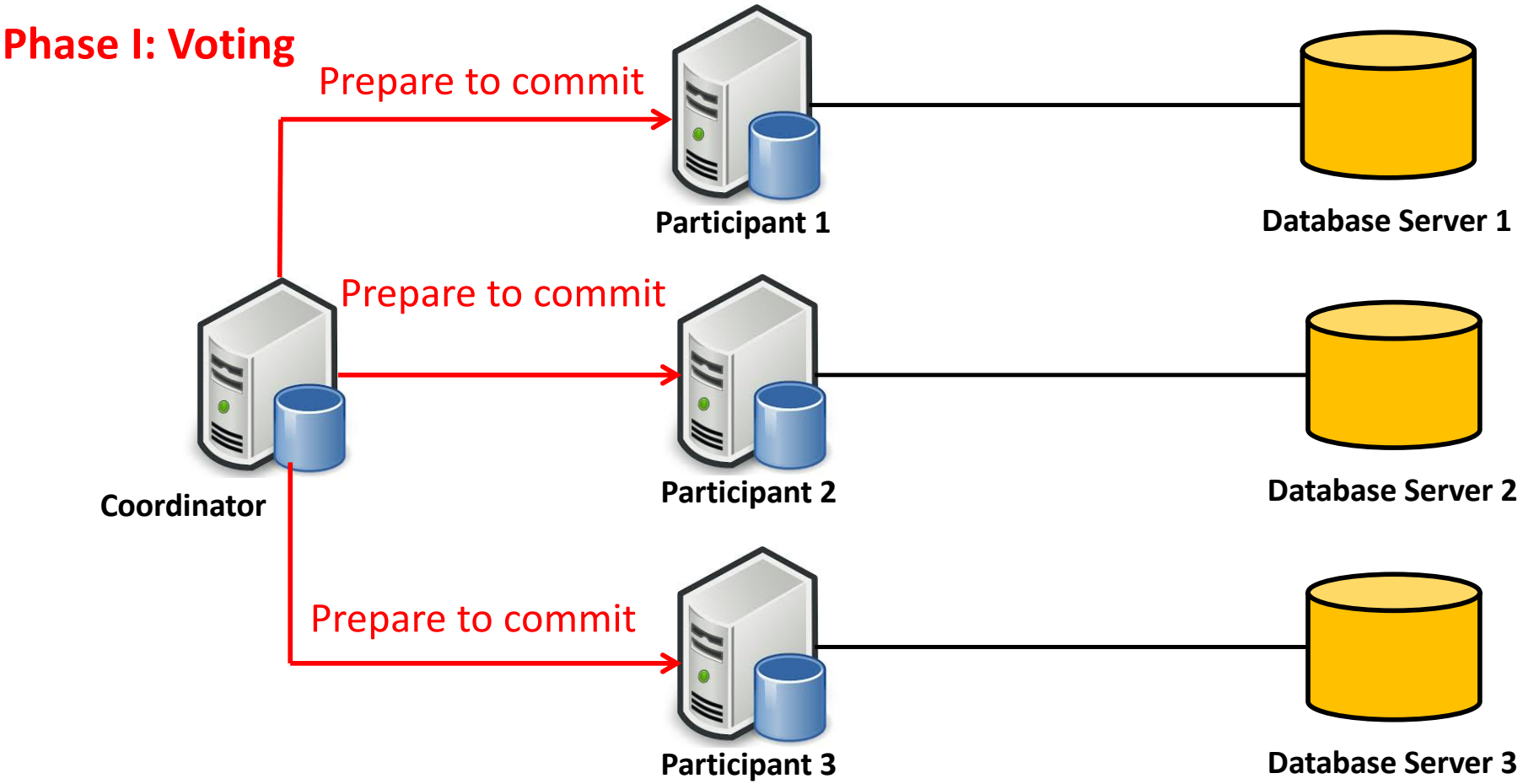
But, Consistency Becomes a Challenge

- An example:
  - In an e-commerce application, the bank database has been replicated across two servers
  - Maintaining consistency of replicated data is a challenge



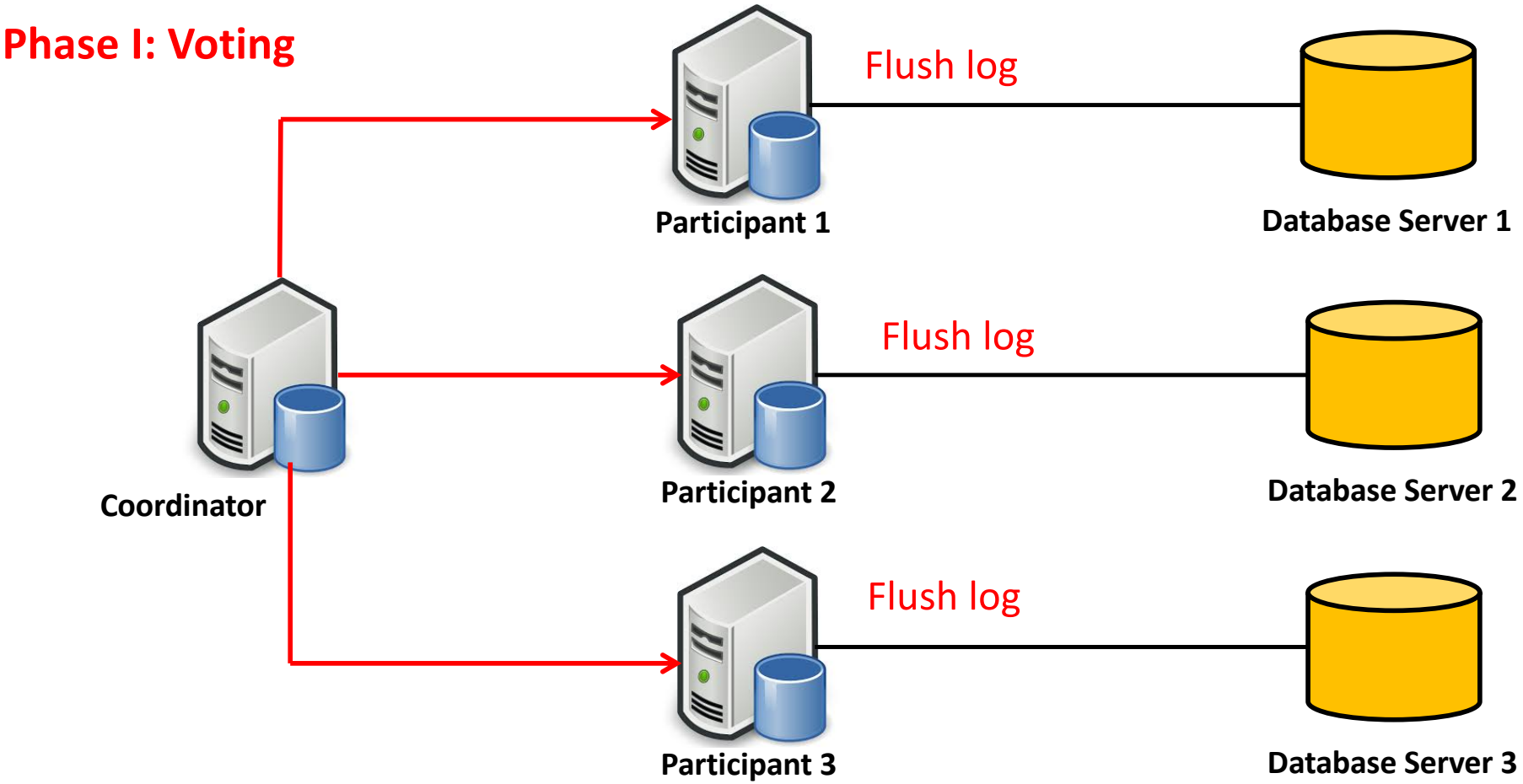
The Two-Phase Commit Protocol

- The two-phase commit protocol (2PC) can be used to ensure atomicity and consistency



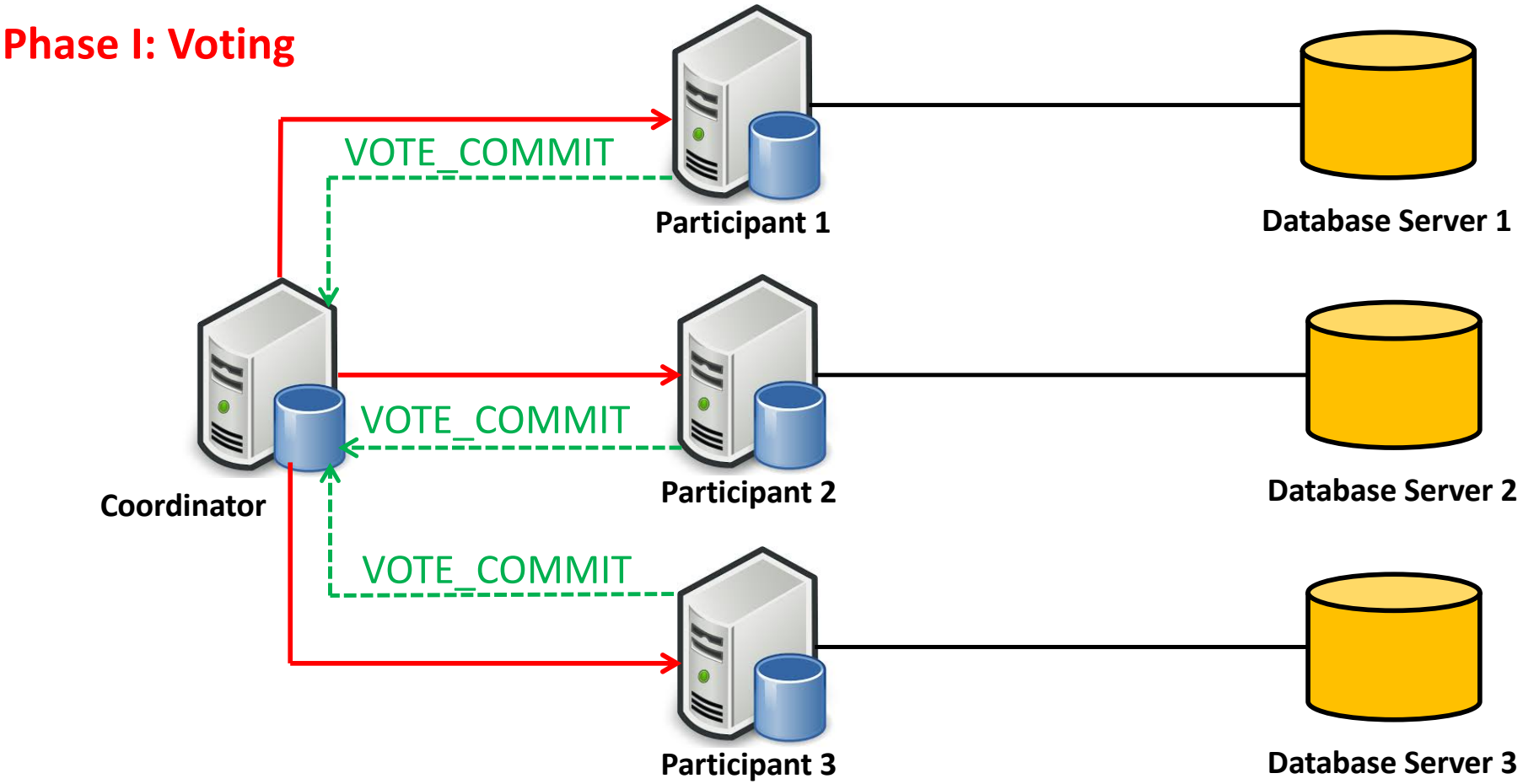
The Two-Phase Commit Protocol

- The two-phase commit protocol (2PC) can be used to ensure atomicity and consistency



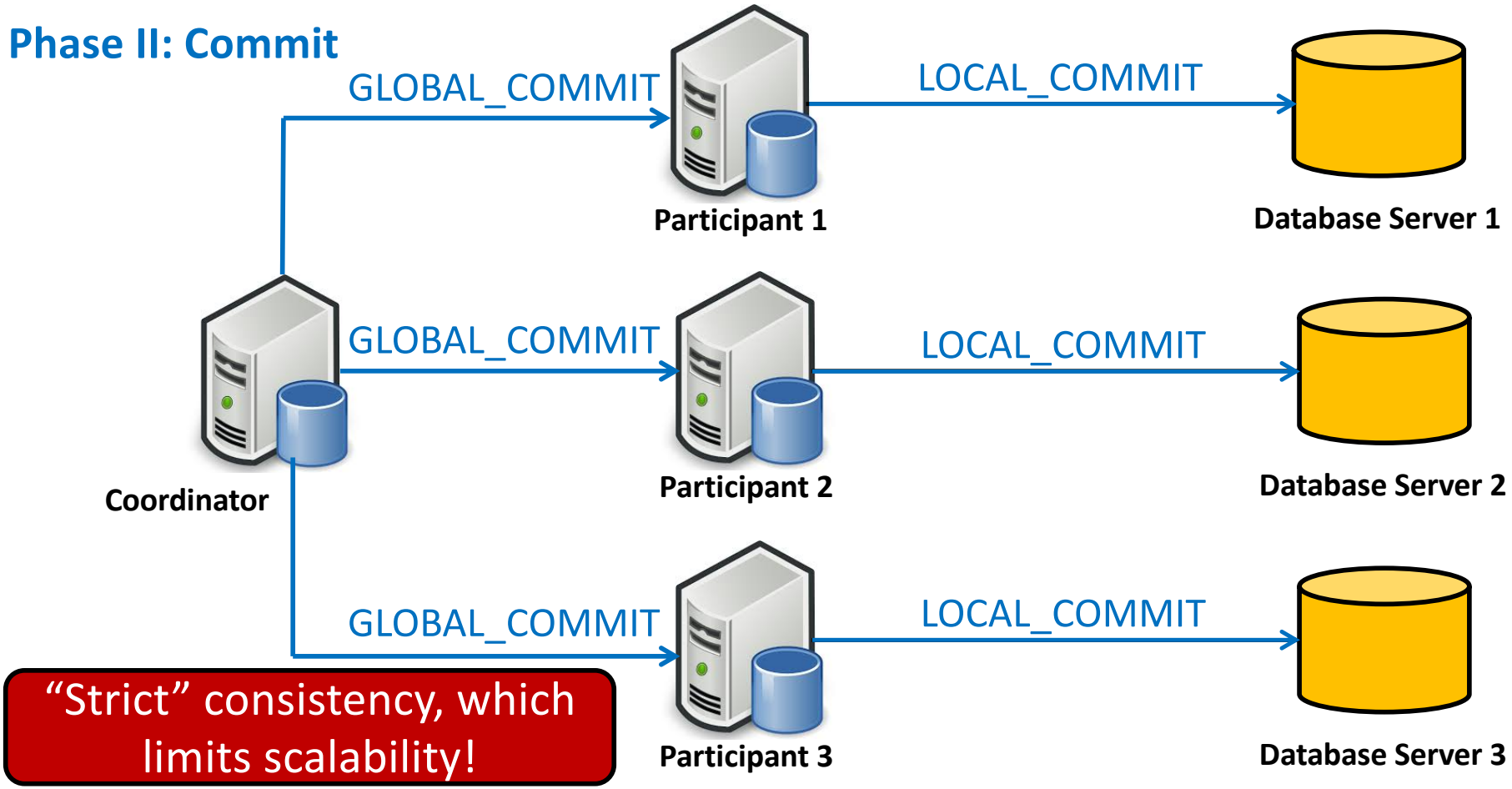
The Two-Phase Commit Protocol

- The two-phase commit protocol (2PC) can be used to ensure atomicity and consistency



The Two-Phase Commit Protocol

- The two-phase commit protocol (2PC) can be used to ensure atomicity and consistency



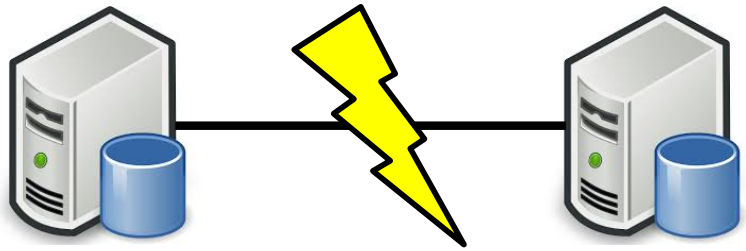
The CAP Theorem

- 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 in a cluster crash, or some hardware or software parts are down due to upgrades
  - **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

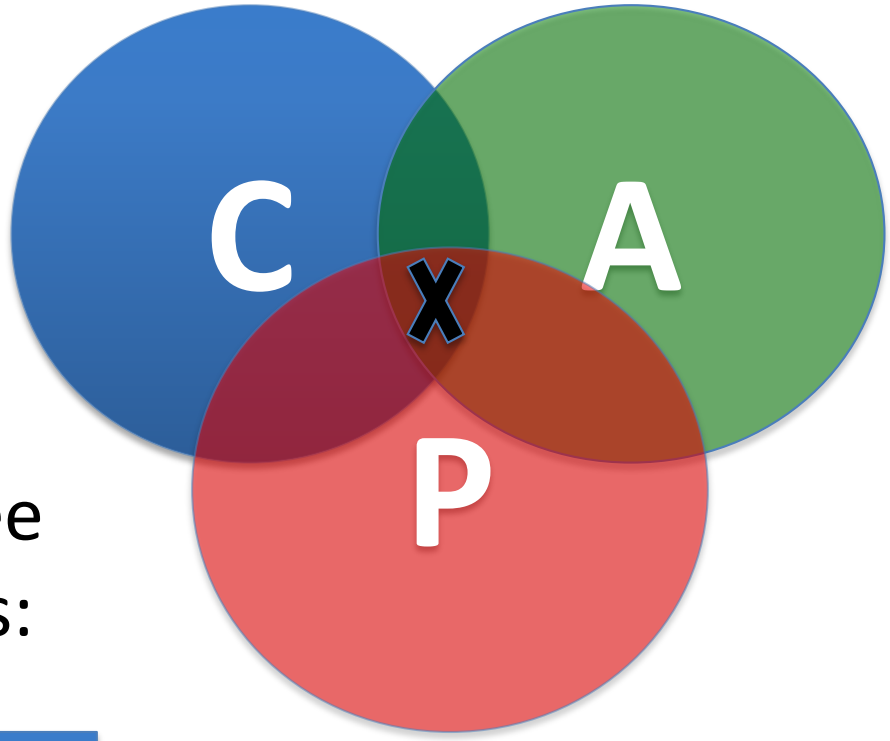
- Consistency + Availability is only possible if there is no network partition, thereby forfeiting Partition Tolerance

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
- When *horizontally* scaling databases to 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*)

Revisit CAP Theorem

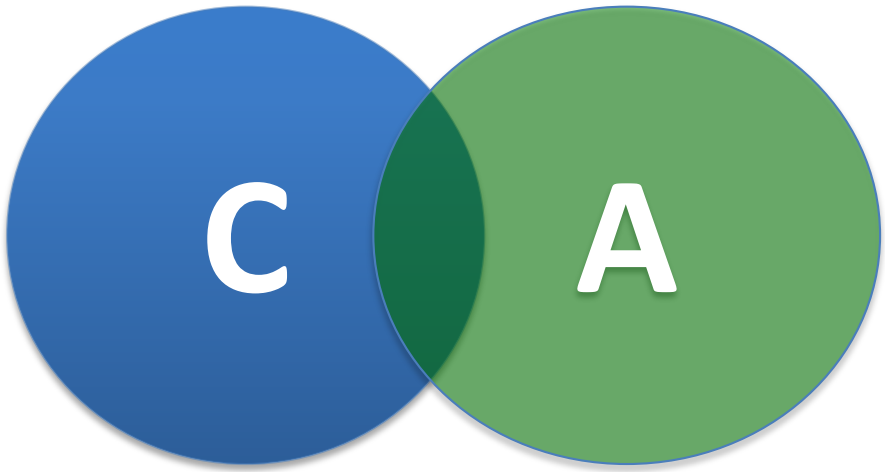
- Of the following three guarantees potentially offered a by distributed systems:
  - Consistency
  - Availability
  - Partition tolerance
- Pick two
- This suggests there are three kinds of distributed systems:
  - CP
  - AP
  - CA



Any problems?

A popular misconception: 2 out 3

- How about CA?
- Can a distributed system (with unreliable network) really be not tolerant of partitions?



A few witnesses

- Coda Hale, Yammer software engineer:
  - "Of the CAP theorem's Consistency, Availability, and Partition Tolerance, **Partition Tolerance is mandatory in distributed systems**. You cannot not choose it."

http://codahale.com/you-cant-sacrifice-partition-tolerance/

A few witnesses

- Werner Vogels, Amazon CTO
  - "An important observation is that in larger distributed-scale systems, network partitions are a given; therefore, **consistency and availability cannot be achieved at the same time**."

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

A few witnesses

- Daneil Abadi, Co-founder of Hadapt
  - So in reality, there are only two types of systems ... I.e., if there is a partition, **does the system give up availability or consistency?**

http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html

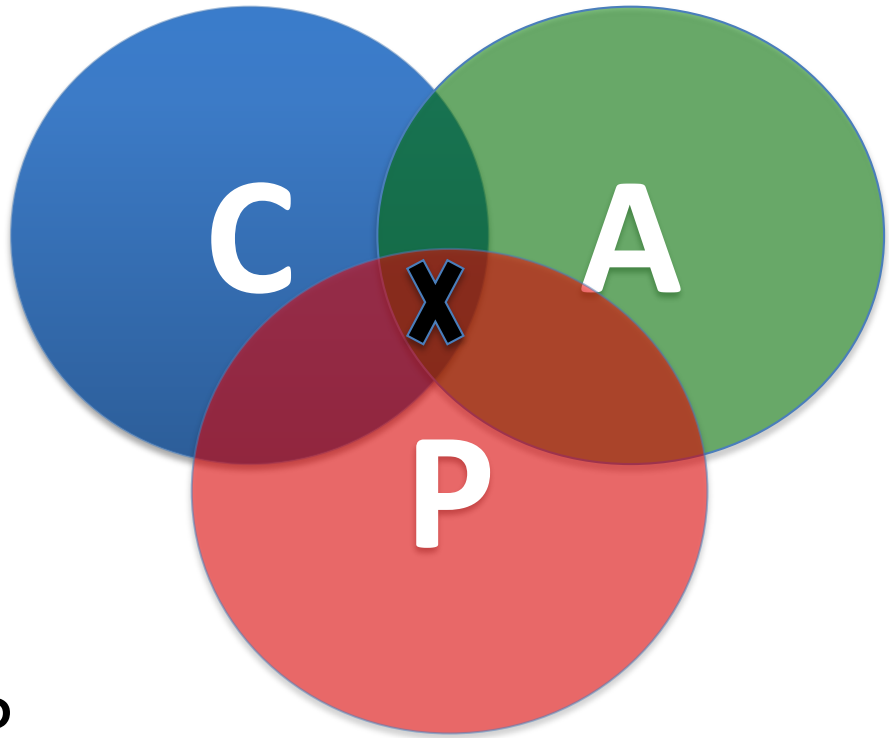
CAP Theorem 12 year later

- Prof. Eric Brewer: father of CAP theorem
  - "The "2 of 3" formulation was always **misleading** because it tended to oversimplify the tensions among properties. ...
  - **CAP prohibits only a tiny part of the design space**: *perfect availability and consistency in the presence of partitions*, which are rare."

http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

Consistency or Availability

- Consistency and Availability is not “binary” decision
- AP systems relax consistency in favor of availability – but are not inconsistent
- CP systems sacrifice availability for consistency- but are not unavailable
- This suggests both AP and CP systems can offer a degree of consistency, and availability, as well as partition tolerance



AP: Best Effort Consistency

- Example:
  - Web Caching
  - DNS
- Trait:
  - Optimistic
  - Expiration/Time-to-live
  - Conflict resolution

CP: Best Effort Availability

- Example:
  - Majority protocols
  - Distributed Locking (Google Chubby Lock service)
- Trait:
  - Pessimistic locking
  - Make minority partition unavailable

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

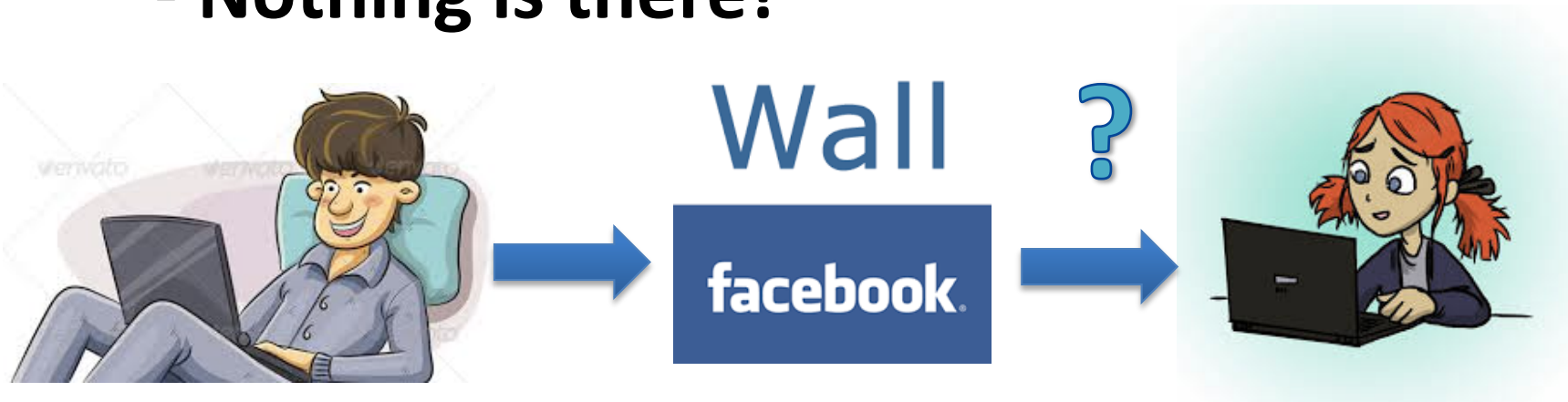
Eventual Consistency Variations

- 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

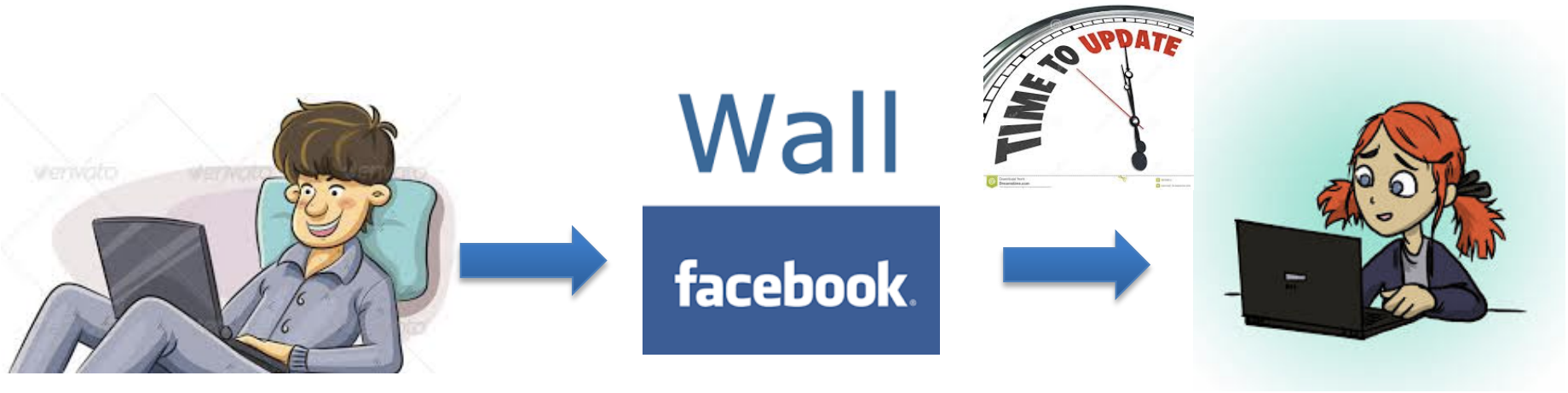
- Bob finds an interesting story and shares with Alice by posting on her Facebook wall
- Bob asks Alice to check it out
- Alice logs in her account, checks her Facebook wall but finds:

  - **Nothing is there!**



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!**



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**

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

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
- Neither strong consistency nor guaranteed availability, but it may significantly increase the tolerance of network disruption

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

- No single uniform requirement
  - 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

Discussion

- In an e-commercial system (e.g., Amazon, e-Bay, etc.), what are the trade-offs between consistency and availability you can think of? What is your strategy?
- Hint -> Things you might want to consider:
  - Different types of data (e.g., shopping cart, billing, product, etc.)
  - Different types of operations (e.g., query, purchase, etc.)
  - Different types of services (e.g., distributed lock, DNS, etc.)
  - Different groups of users (e.g., users in different geographic areas, etc.)

Partitioning Examples

- Data Partitioning
- Operational Partitioning
- Functional Partitioning
- User Partitioning
- Hierarchical Partitioning

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

Partitioning Examples

Operational Partitioning

- Each operation may require different balance between consistency and availability
- Example:
  - Reads: high availability; e.g.., “query”
  - Writes: high consistency, lock when writing; e.g., “purchase”

Partitioning Examples

Functional Partitioning

- System consists of sub-services
- Different sub-services provide different balances
- Example: A comprehensive distributed system
  - Distributed lock service (e.g., Chubby) :
    - Strong consistency
  - DNS service:
    - High availability

Partitioning Examples

User Partitioning

- Try to keep related data close together to assure better performance
- Example: Craglist
  - Might want to divide its service into several data centers, e.g., east coast and west coast
    - Users get high performance (e.g., high availability and good consistency) if they query servers closet to them
    - Poorer performance if a New York user query Craglist in San Francisco

Partitioning Examples

Hierarchical Partitioning

- Large global service with local “extensions”
- Different location in hierarchy may use different consistency
- Example:
  - Local servers (better connected) guarantee more consistency and availability
  - Global servers has more partition and relax one of the requirement

What if there are no partitions?

- 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
  - 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

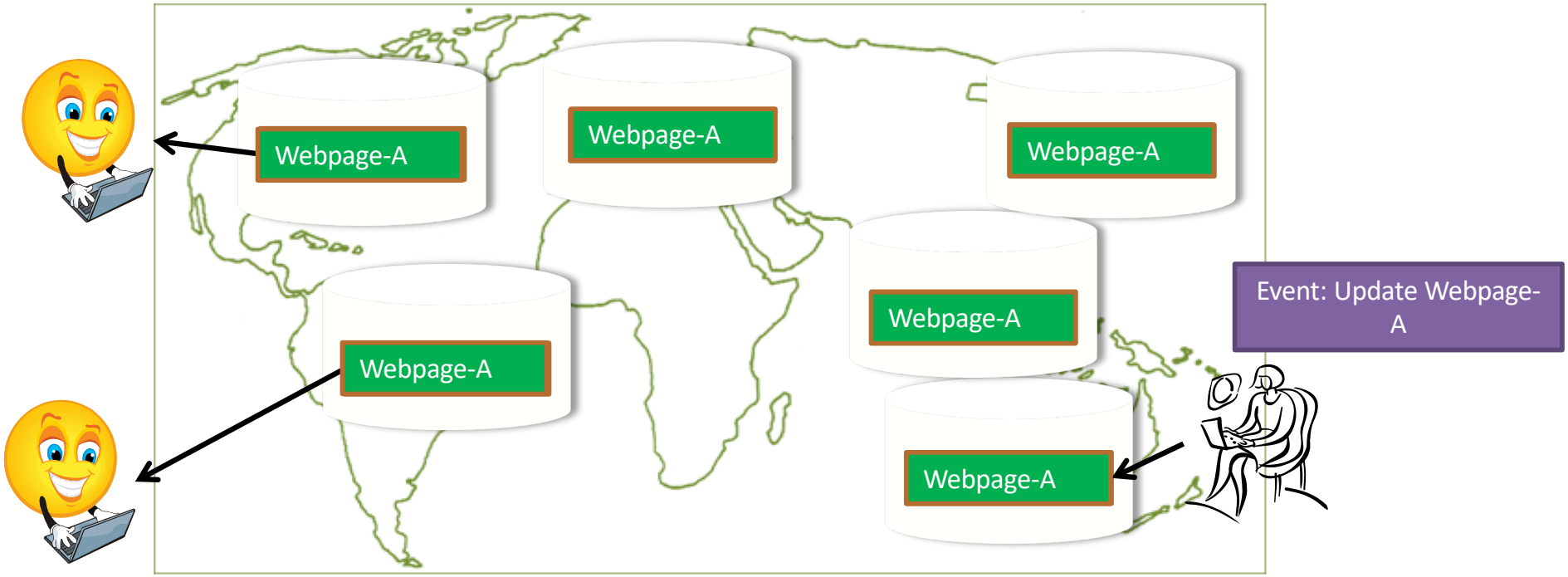
- In particular, such databases apply the BASE properties:
  - **B**asically **A**vailable: the system guarantees Availability
  - **S**oft-State: the state of the system may change over time
  - **E**ventual Consistency: the system will *eventually* become consistent

Eventual Consistency

- A database is termed as *Eventually Consistent* if:
  - All replicas will *gradually* become consistent in the absence of updates

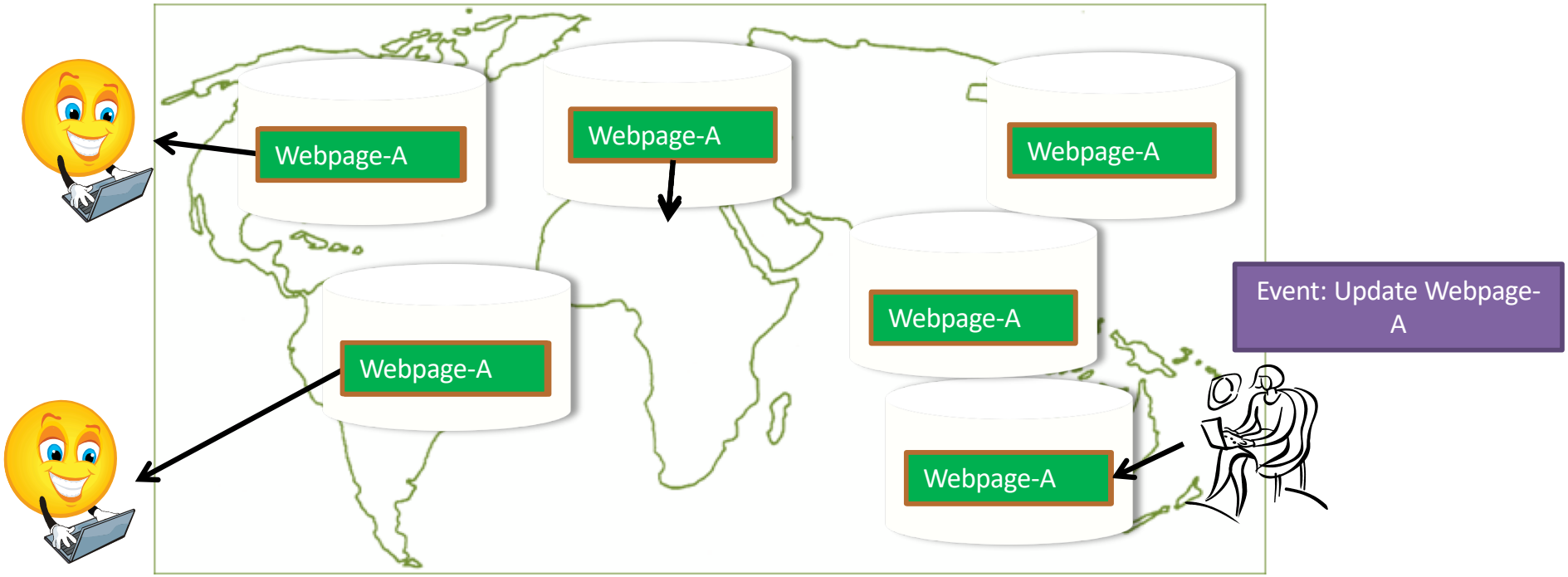
Eventual Consistency

- A database is termed as *Eventually Consistent* if:
  - All replicas will *gradually* become consistent in the absence of updates



Eventual Consistency: A Main Challenge

- But, what if the client accesses the data from different replicas?



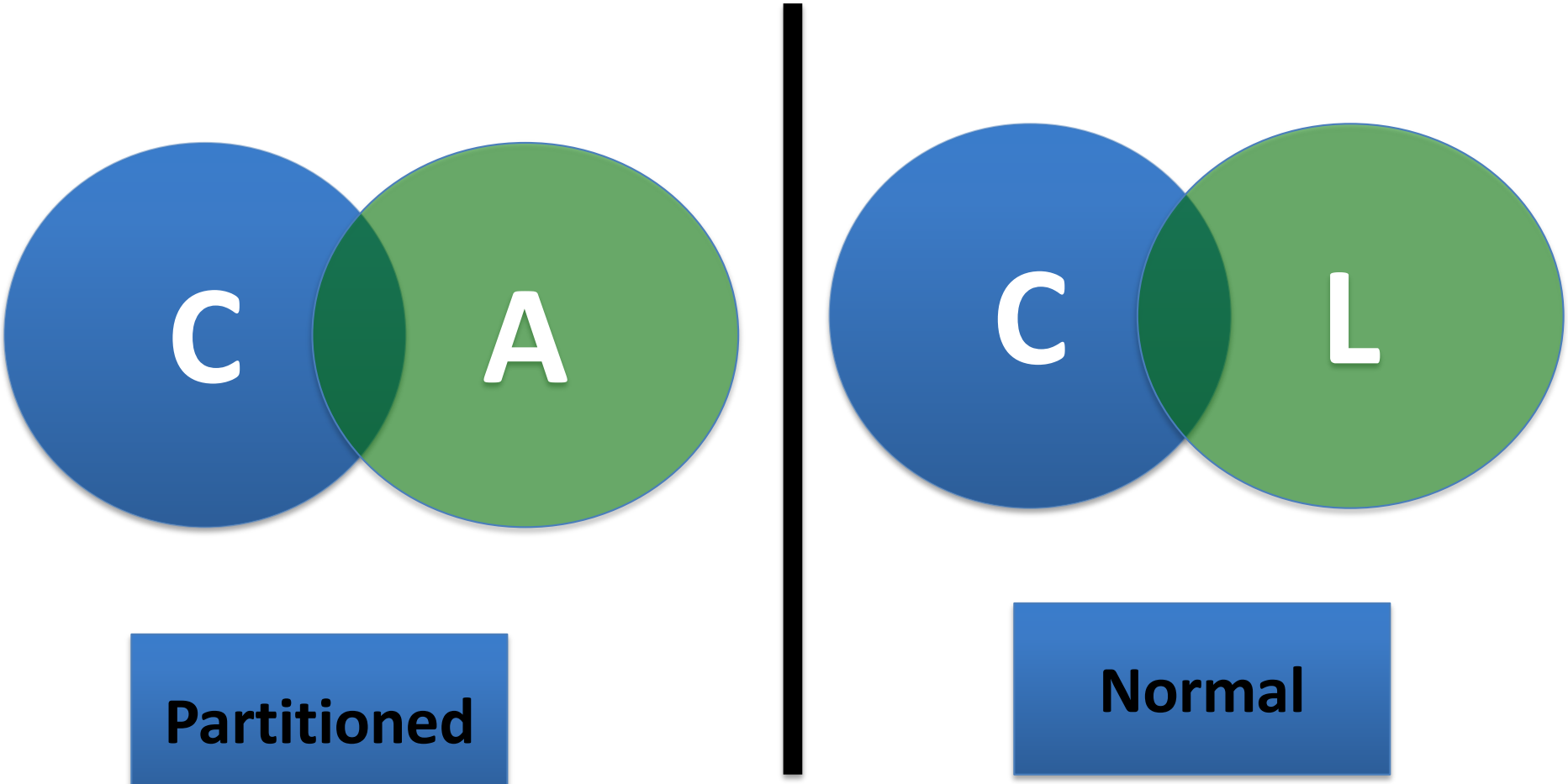
Protocols like Read Your Own Writes (RYOW) can be applied!

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.

PACELC



Examples

- **PA/EL Systems:** Give up both Cs for availability and lower latency
  - 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

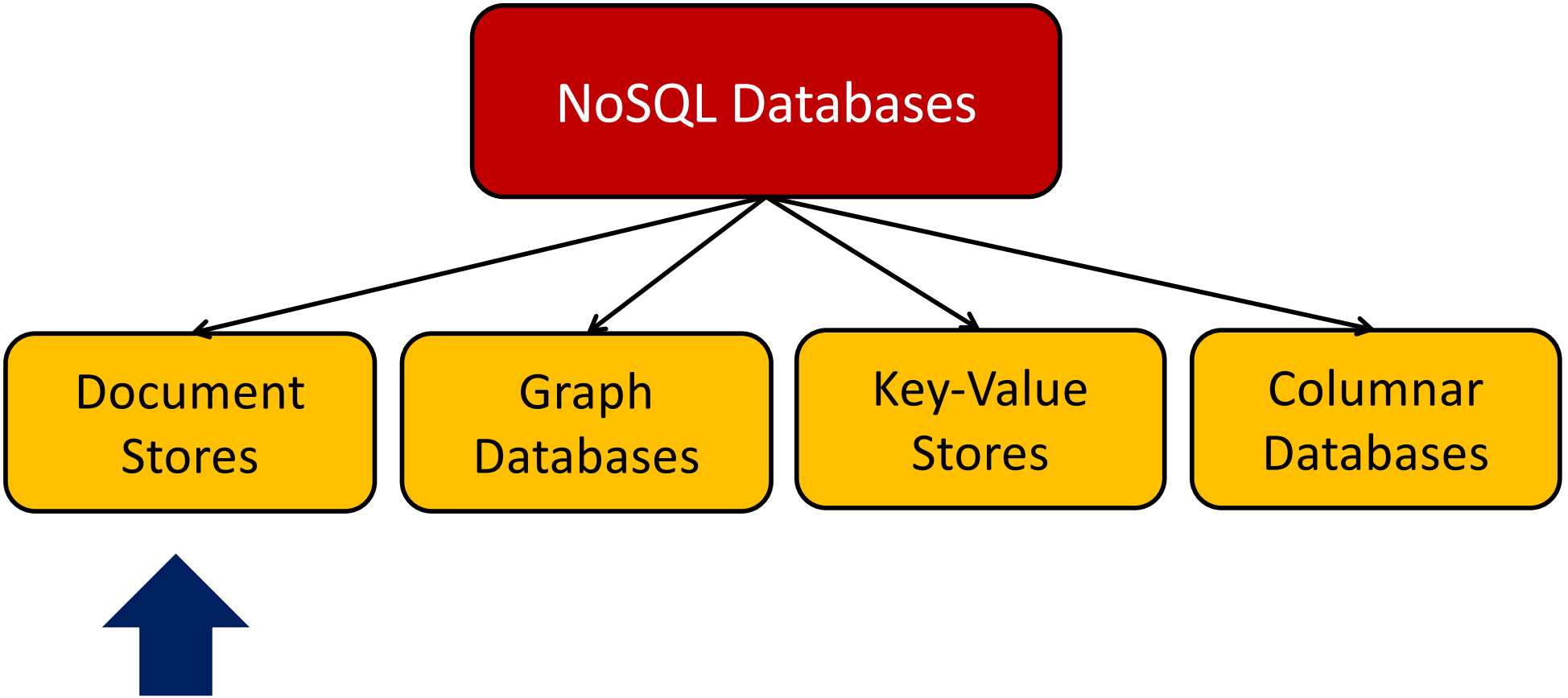
NoSQL Databases

- To this end, a new class of databases emerged, which mainly follow the BASE properties
  - These were dubbed as NoSQL databases
  - E.g., Amazon's Dynamo and Google's Bigtable

- Main characteristics of NoSQL databases include:
  - No strict schema requirements
  - No strict adherence to ACID properties
  - Consistency is traded in favor of Availability

Types of NoSQL Databases

- Here is a limited taxonomy of NoSQL databases:

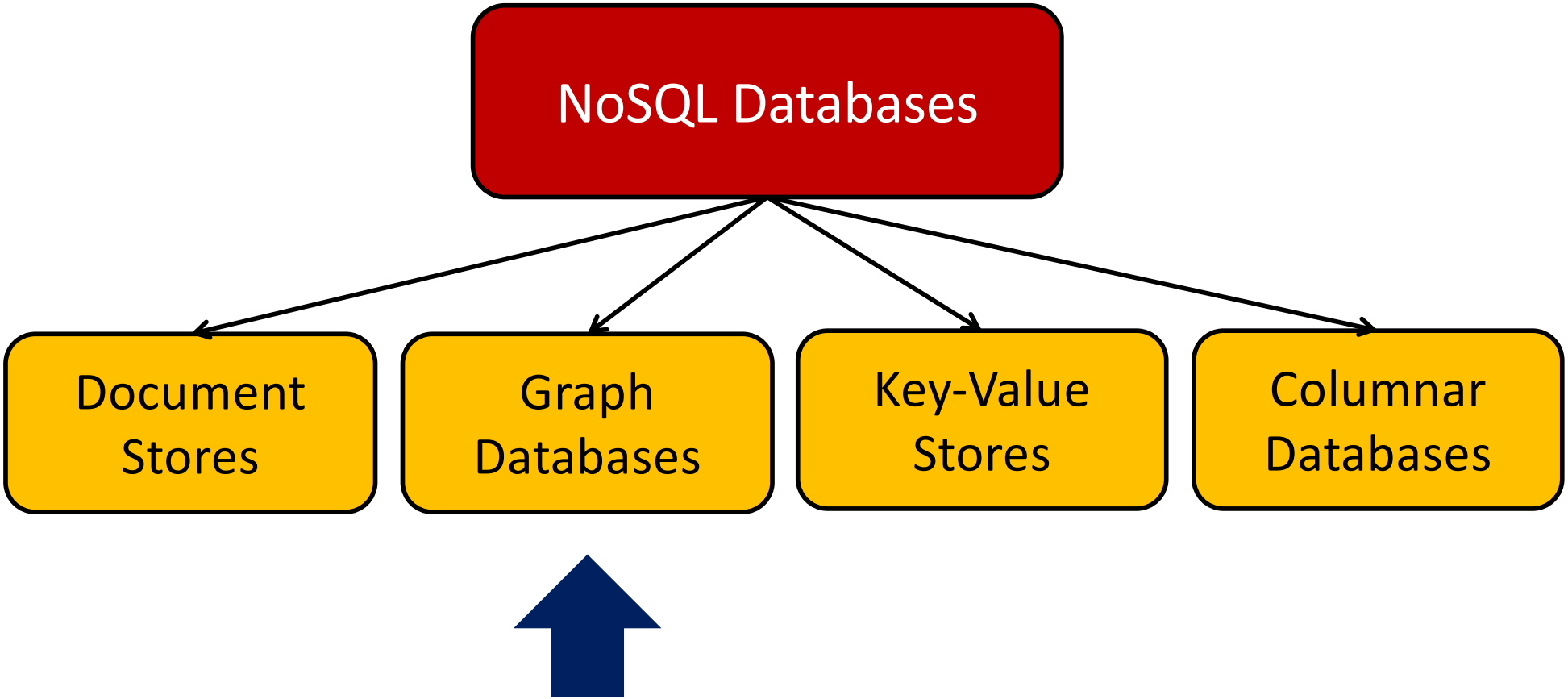


Document Stores

- 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)

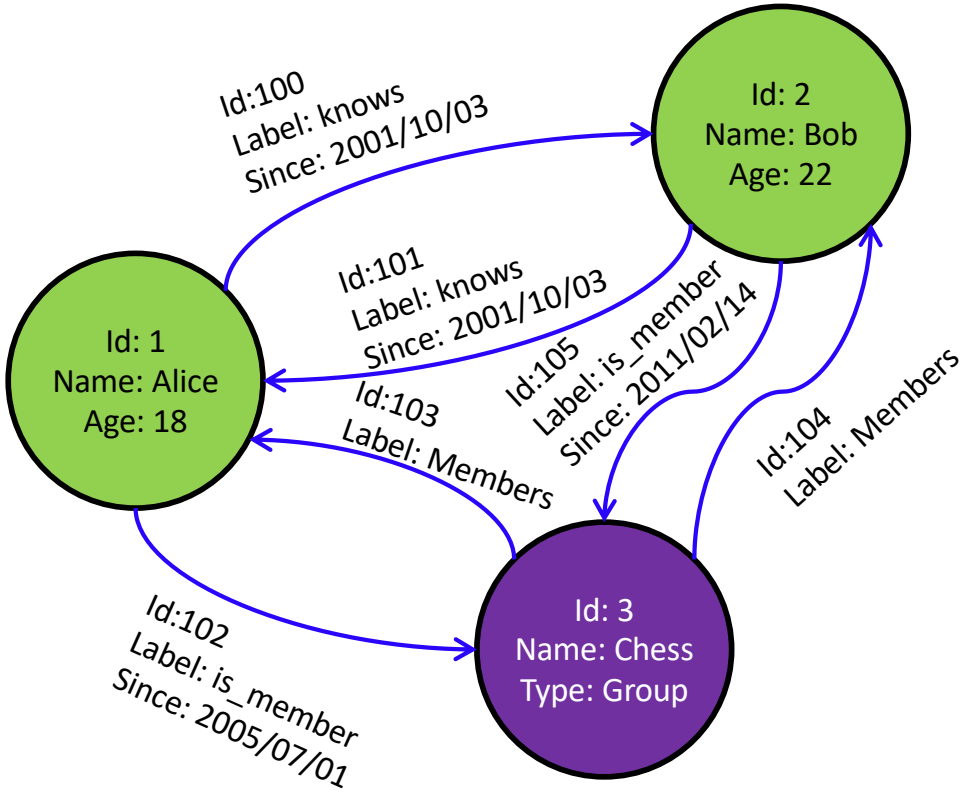
Types of NoSQL Databases

- Here is a limited taxonomy of NoSQL databases:



Graph Databases

- Data are represented as vertices and edges

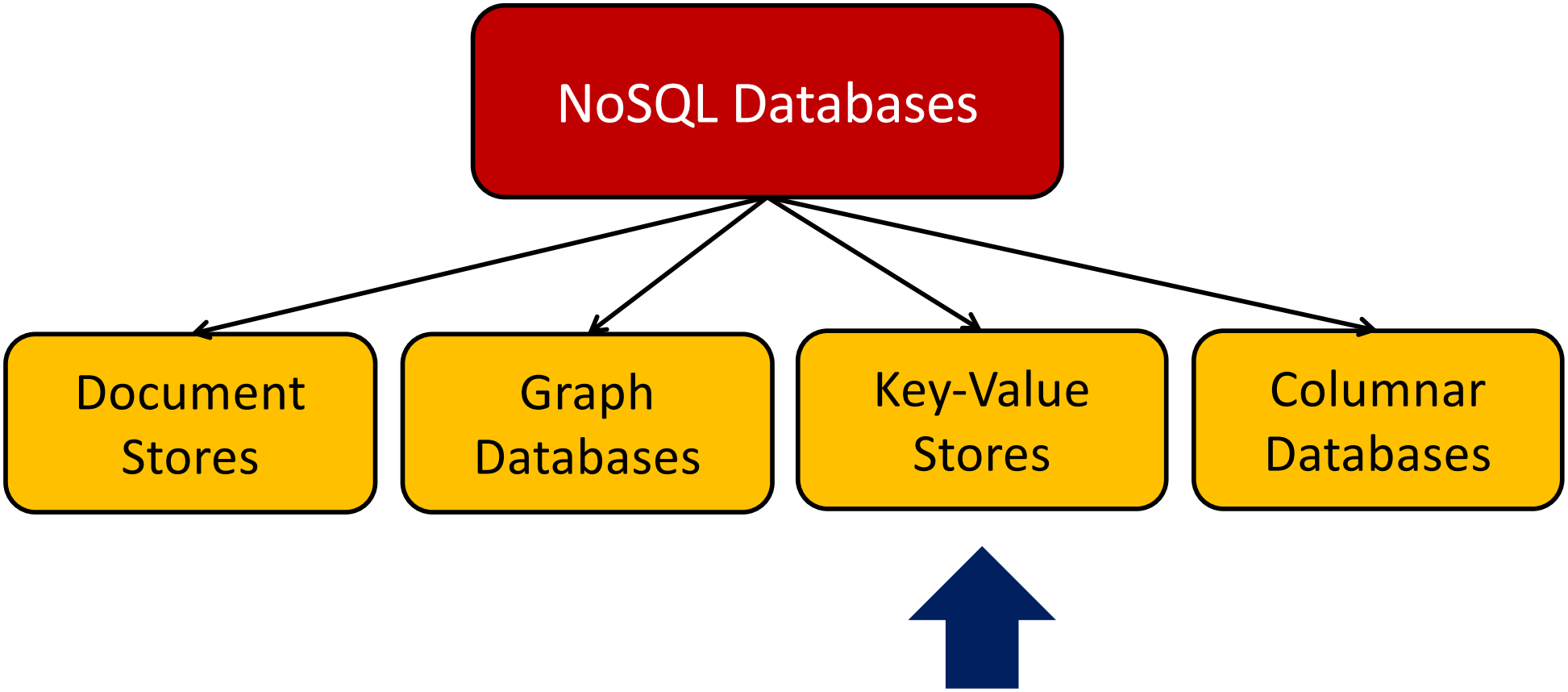


- Graph databases are powerful for graph-like queries (e.g., find the shortest path between two elements)

- E.g., Neo4j and VertexDB

Types of NoSQL Databases

- Here is a limited taxonomy of NoSQL databases:

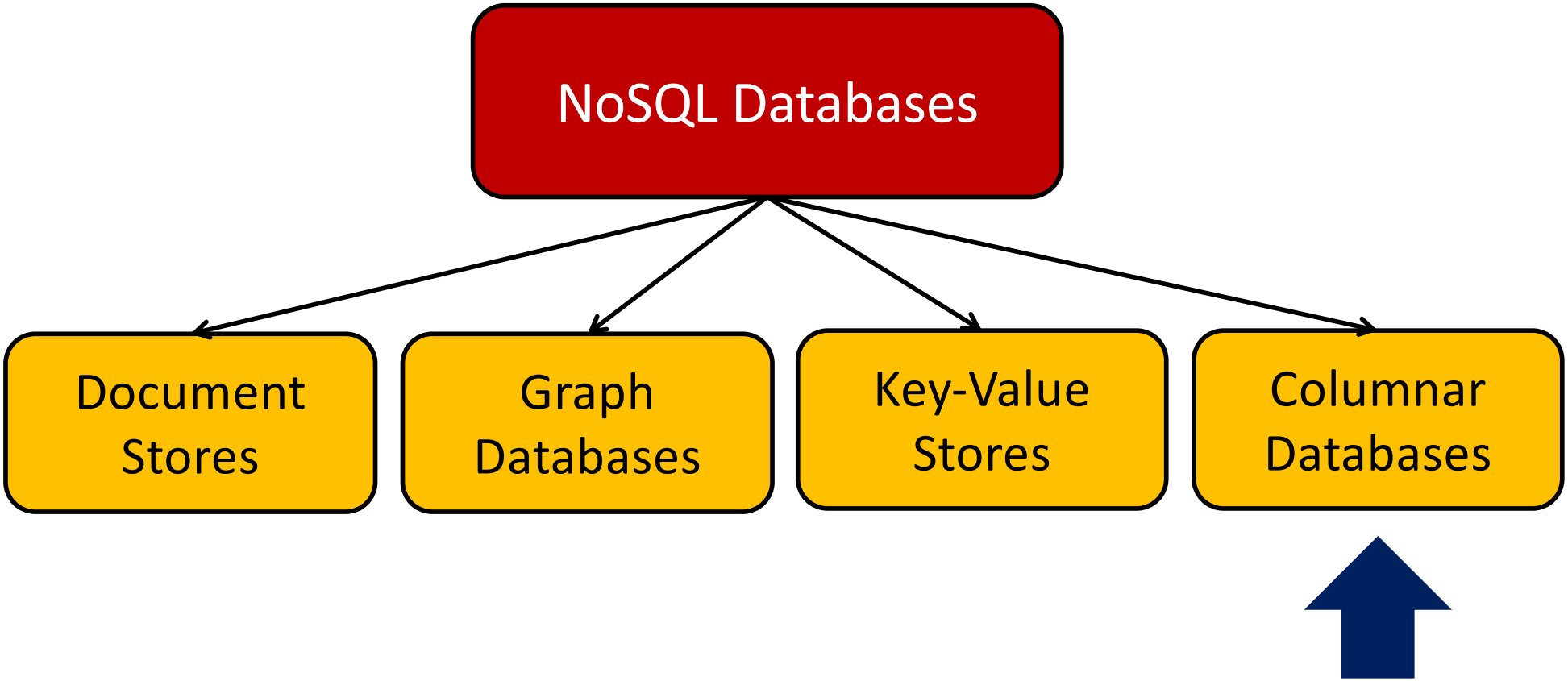


Key-Value Stores

- 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

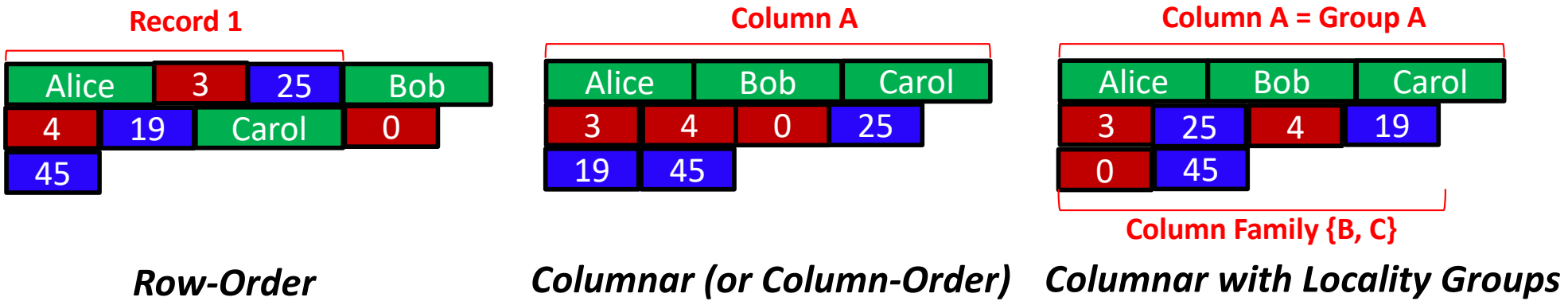
Types of NoSQL Databases

- 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

- Data can be classified into 4 types, *structured*, *unstructured*, *dynamic* and *static*
- Different data types usually entail different database designs
- Databases can be scaled *up* or *out*
- The *2PC protocol* can be used to ensure strict consistency
- Strict consistency limits scalability

Summary (*Cont'd*)

- 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 lead 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

- NoSQL databases have different types:
  - Document Stores
  - Graph Databases
  - Key-Value Stores
  - Columnar Databases