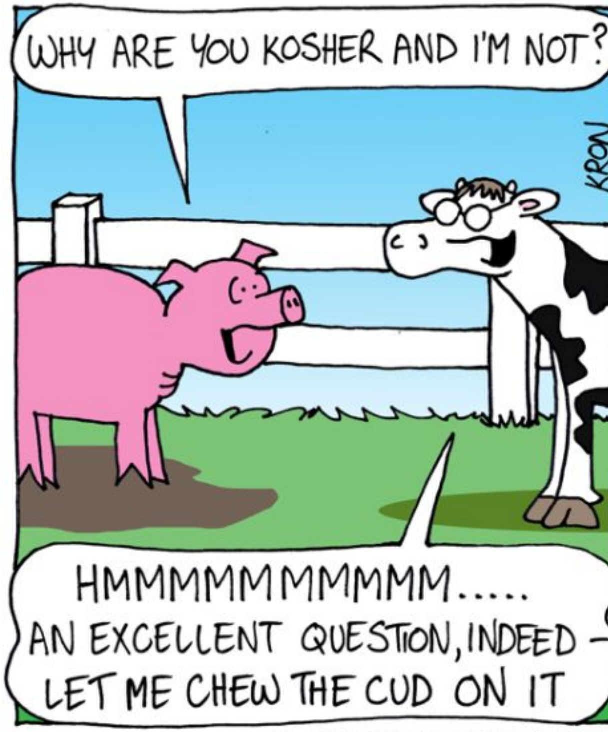
Intro to NoSQL Databases

PS #4 due next Tuesday
PS #5 will be issued tonight



The CARTOON KRONICLES

Structured vs Unstructured

Data can be broadly classified into types:

1. **Structured data:**
   Conforms to a predefined model, which organizes data into a form that is relatively uniform and, thus, easy to store, process, retrieve and manage. (e.g. rows with common attribuites (relational data))
2. **Unstructured data:**
   Opposite of structured data. BLOBs of bits. Irregular and data-dependent attributes.
   (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, text often has abstracts, intros, and references.

Dynamic vs. Static

Data can be classified by temporal properties

Dynamic Data:

Data that changes relatively frequently

e.g., How many steps Joe has walked today, live statistics of a sporting event, or financial object, how many cases reported today?

Static Data:

Opposite of dynamic data

e.g., Medical imaging data from MRI or CT scans

Historical demographic records

Data Classifications

Segmenting data according to one of the following 4 quadrants can help in designing, developing, and maintaining effective data storage and search solutions

	Dynamic	Static
Structured	Bank Transactions, Finanical Statistics	Historical Sports Statistics, College Transcripts
Unstructured	Live video streams, On-line shared documents, Message Feeds	Archived YouTube videos, Warehoused medical data

Relational databases were designed for structured data, and suffers from scalability isses when data is dynamic.

File systems or *NoSQL databases* are more often used for (static), unstructured data (*more on these later*)

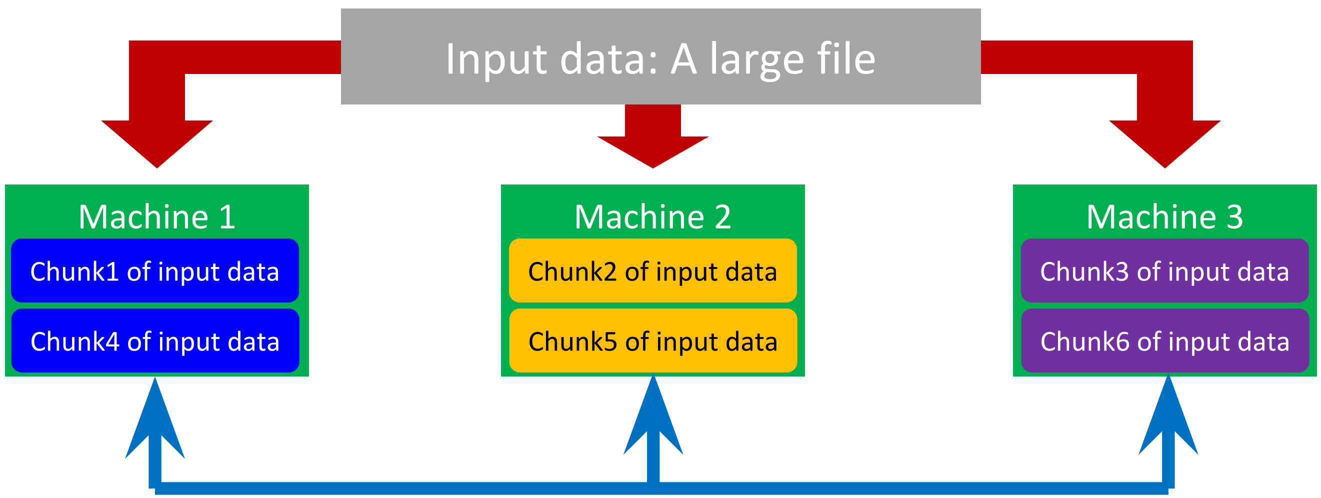
Scaling Issues of Data Access

Traditional DBMSs can be either scaled:

- Vertically (or Up)
  - Achieved by hardware upgrade (e.g., faster CPUs, more memory, or larger disks)
  - Limited by the amount of CPU, RAM and disk and network bandwitdh available to a single machine
- Horizontally (or Out)
  - Can be achieved by adding more servers (machines)
  - Requires distributing databases and probably replication
    - Distribute tables to different machines
    - Distrbute rows of tables to different machines
    - Distrbute columns of tables to different machines
  - Limited by the Read-to-Write ratio and communication overhead

Distributing Rows

Performance can be achieved by distributing the rows of tables across multiple DBMS servers. This is called *sharding*. Sharding provides concurrent/parallel access, but the final results need to be combined or merged.



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

Computational Limits

Recall Amdahl's Law…

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, some fraction, *s*, is not parallelizable (*s* is for serial) while *1-s* fraction is parallelizable.

Then the speedup by Amdahl's formula:

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

An Example

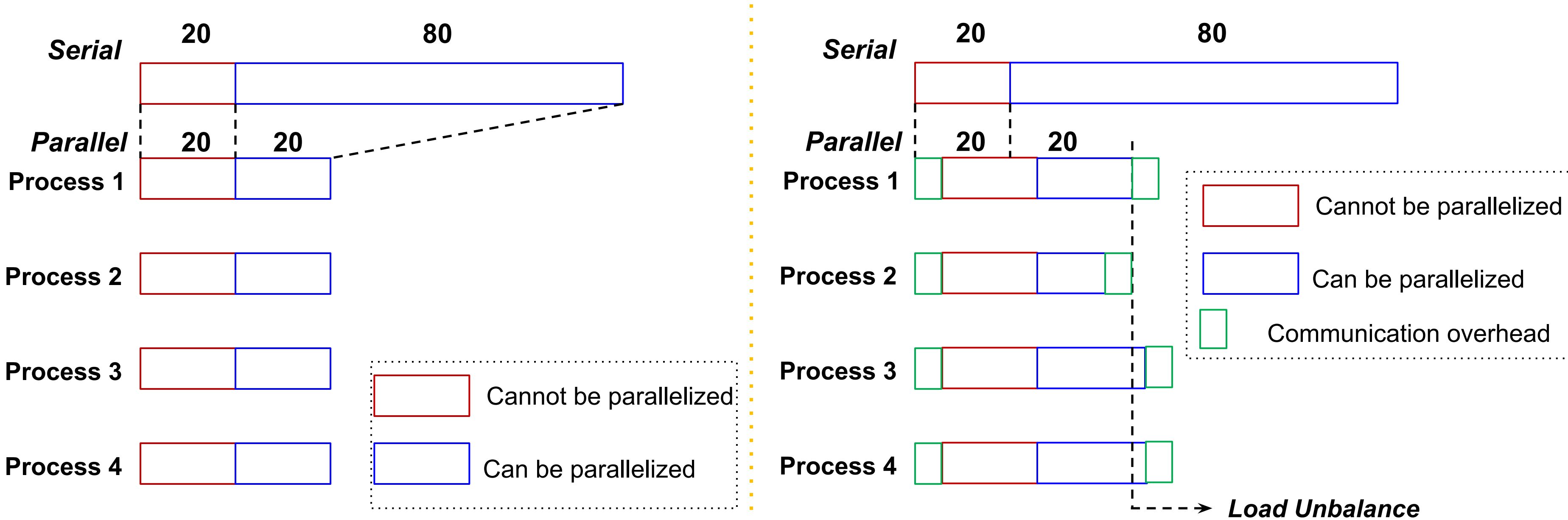
- Suppose that:
  - 60% of your query can be parallelized
  - 6 machines are used in the parallel components of the tuple selection
- The speedup you can get according to Amdahl's law is:

$$\frac{1}{s+\frac{1-s}{p}} = \frac{1}{0.4+\frac{0.6}{6}} = 2.0$$

Although you use 6 processors your speedup is only 2 times!

Communication & Imbalance

- In reality, Amdahl's argument is over simplified
- Communication overheads and workload imbalance also impact parallel programs



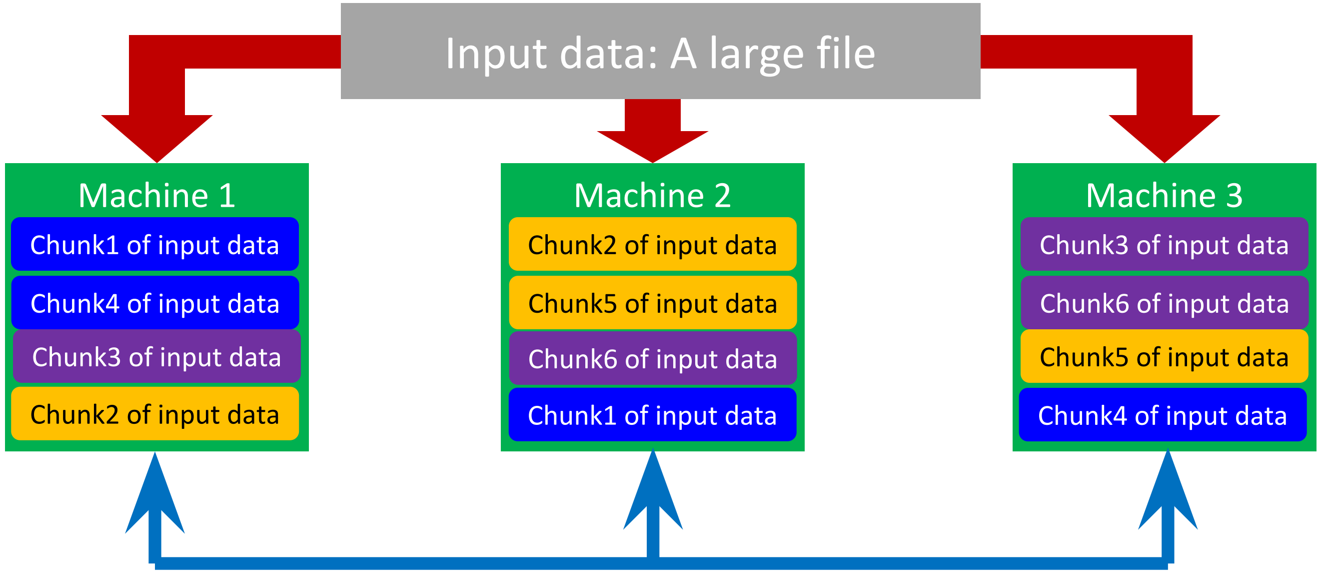
1. Parallel Speed-up: The "Ideal" Case (2.5x)

2. Parallel Speed-up: Actual Case (~2X)

Shard and Replicate

Why replicate data?

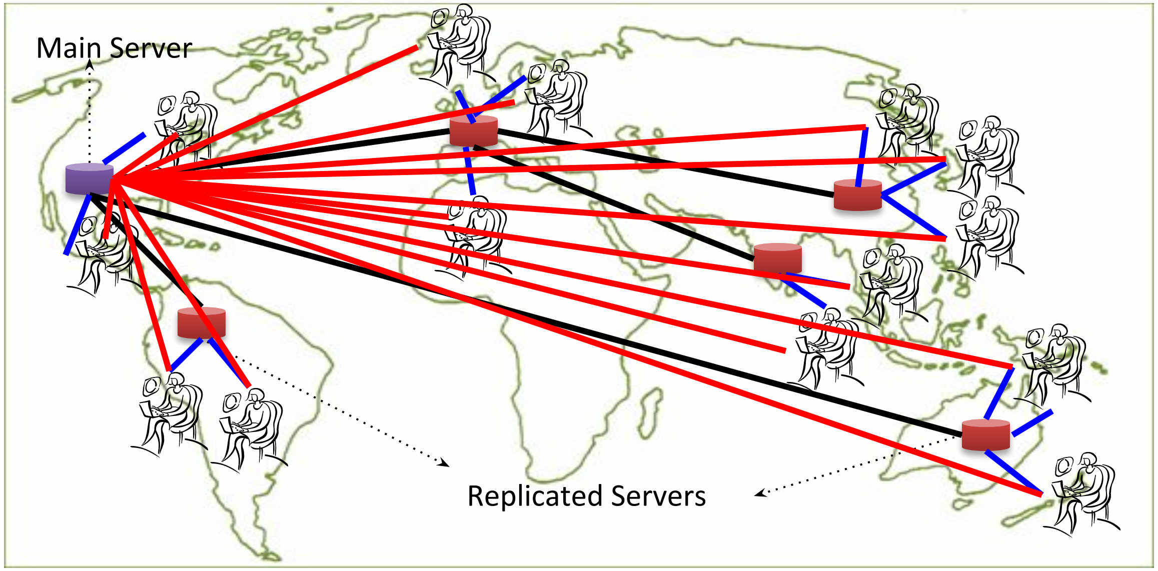
- Replicating data across servers helps by:
  - Avoiding performance bottlenecks
  - Avoiding single point of failures



Shard and Replicate

Why replicate data?

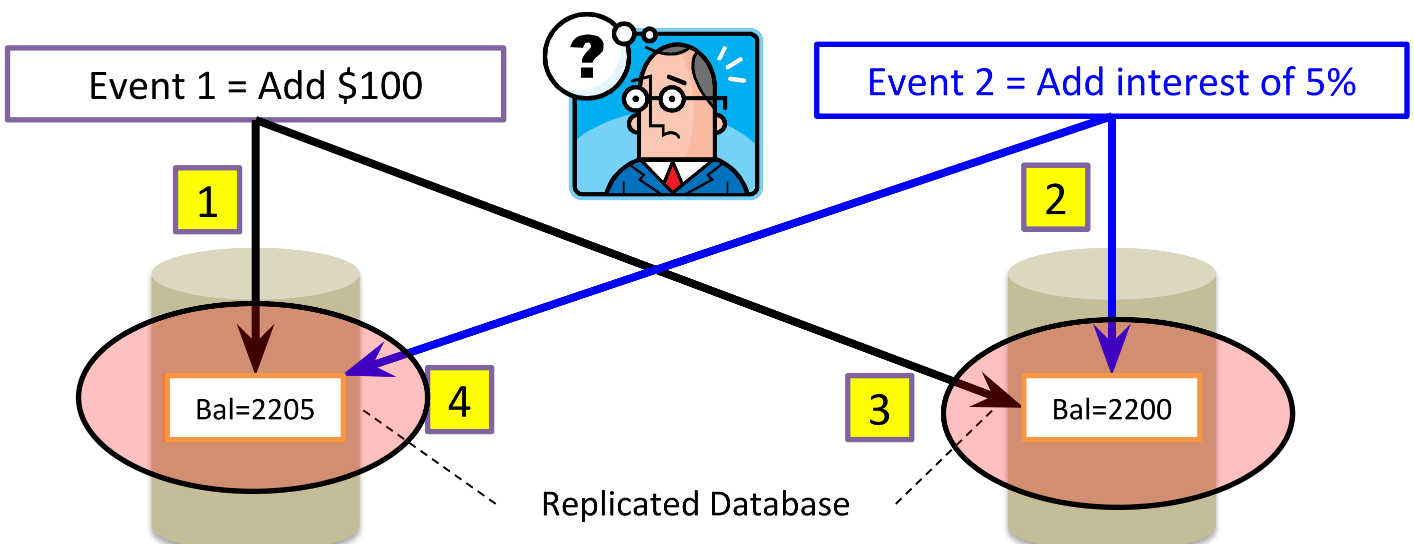
- Replicating data across servers helps by:
  - Avoiding performance bottlenecks
  - Avoiding single point of failures
  - Also enhances *scalability* and *availability*



But,

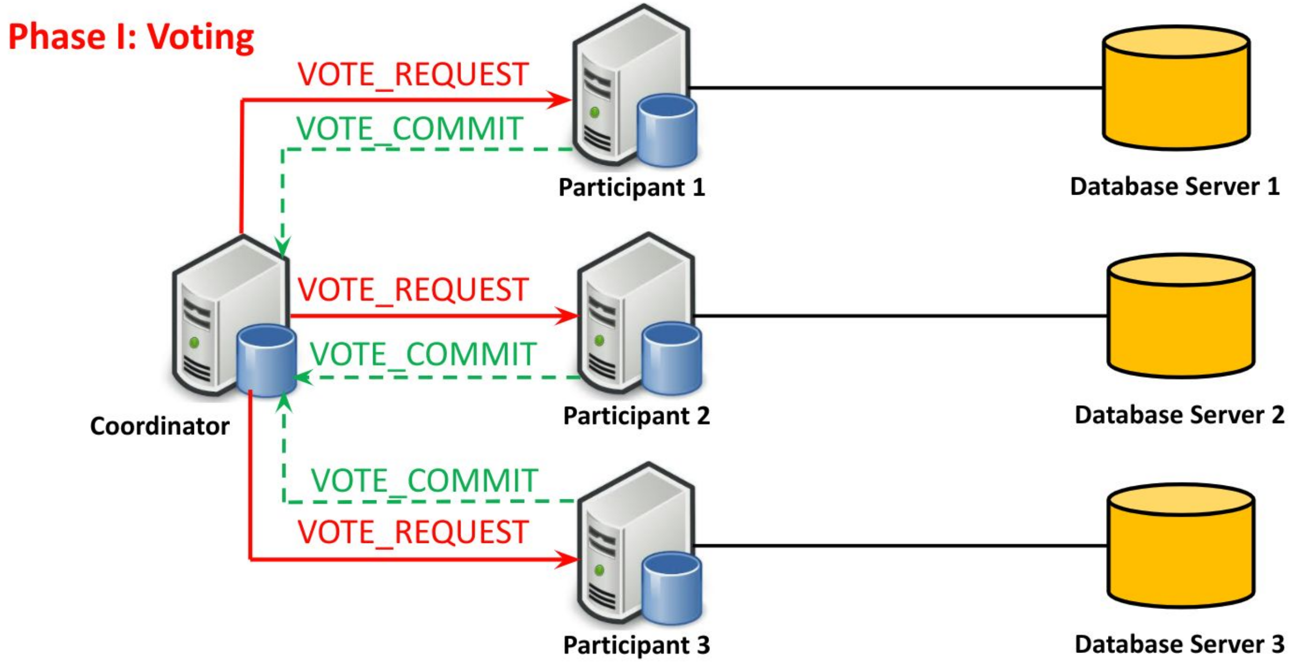
Consistency Becomes a Challenge…

- An example:
  - In an e-commerce application, the bank database has been replicated across two servers
  - Maintaining consistency of replicated data is a challenge
  - Our scheduling approach actually assumes a serial execution...



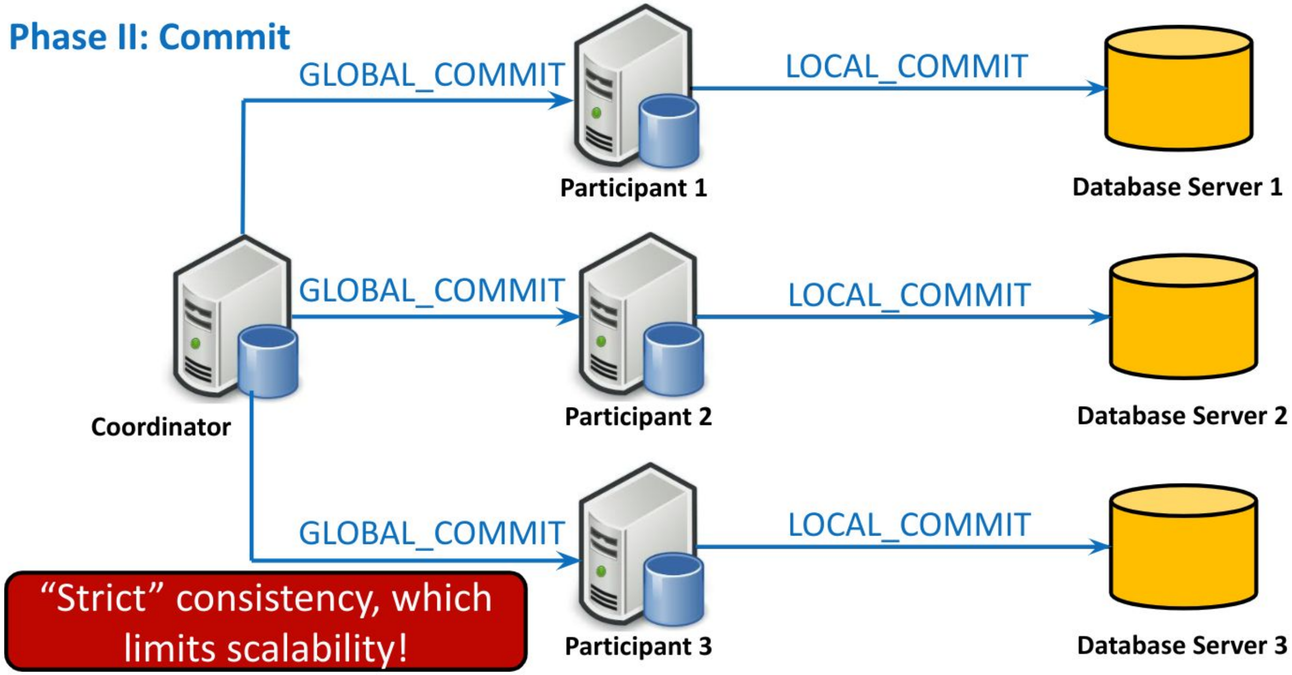
Distributed 2PL

- Two-phase locking protocol (2PL) can still be used to ensure atomicity and consistency, but it increases the serial fraction of execution, and usually involves a single "lock-authority" or coordinator.

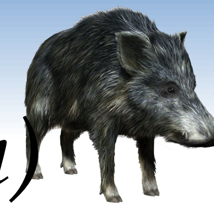


Distributed 2PL

- Two-phase locking protocol (2PL) can still be used to ensure atomicity and consistency, but it increases the serial fraction of execution, and usually involves a single "lock-authority", coordinator, and voting.



The CAP Theorem (Brewer's Theorem)

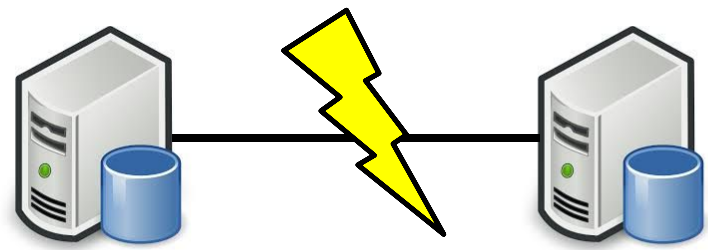


- The fundemental limitations of distributed databases can be described in the so called the CAP theorem
  - **C**onsistency: every node always sees the same data at any given instance (i.e., strict consistency)
  - **A**vailability: continues to operate, even if a node in a cluster crashes, or some hardware or software parts are down due to upgrades
  - **P**artition Tolerance: continues to operate in the presence of network partitions (breaks in connectivity)

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

CAP Examples

- Assume two nodes on opposite sides of a network partition:



- Availability + Partition Tolerance forfeit Consistency
- 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 (no-delay network is always available)

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 significant lost revenue
- When 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 (*as implied by the CAP theorem*)

The Consistency Trade-off

- Maintain a balance between the strictness of consistency versus availability/scalability
- "Good-enough" consistency is application dependent

Loose Consistency **Strict Consistency**

Easier to implement, and can be high performance

Generally hard to implement, and reduces performance

Performance is measured in throughput (how many transactions per second the system can mange) and latency (how long you have to wait)

BASE Properties

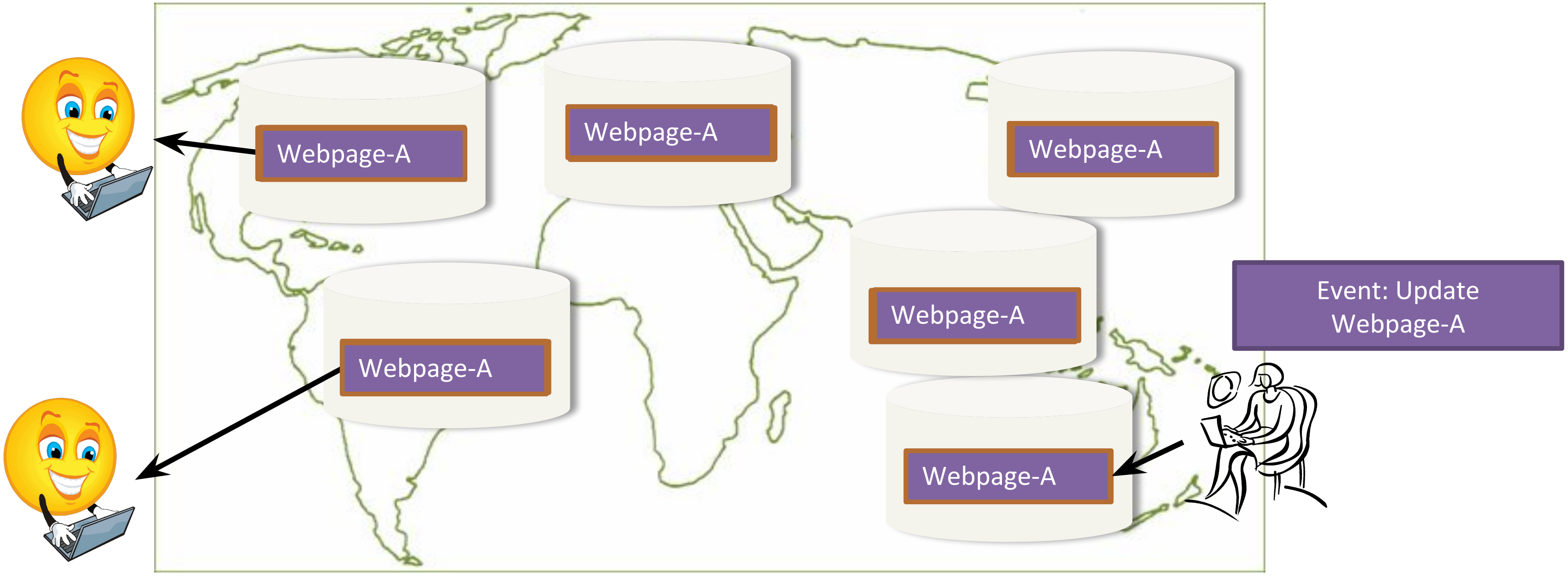
The CAP theorem proves that it is impossible to guarantee strict Consistency and Availability while being able to tolerate network Partitions.

This resulted in databases with relaxed ***ACID*** guarantees

- In particular, such databases apply the ***BASE*** properties:
  - **B**asically **A**vailable: the system favors availability
  - **S**oft-State: state of the system may change over time but might be slightly inconsistent for small intervals
  - **E**ventual Consistency: the system will *eventually* become consistent, particularly if nothing is changing

Eventual Consistency

- A database is termed as *Eventually Consistent* if:
  - All replicas will *gradually* converge to a single consistent state in the absence of any updates for some specified interval



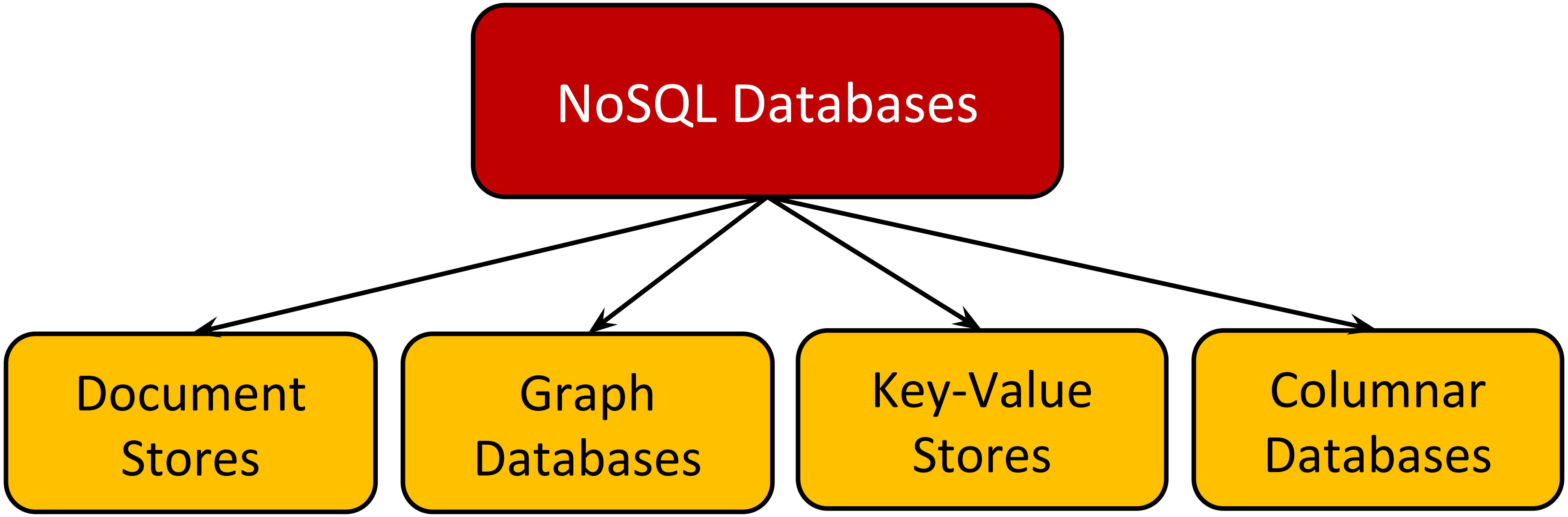
NoSQL Databases

- 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 adherence to ACID properties
  - Availability > Consistency
  - Consistency eventually, if all updates stop
  - Less strict schema requirements

Types of NoSQL Databases

Here is a taxonomy of NoSQL databases:

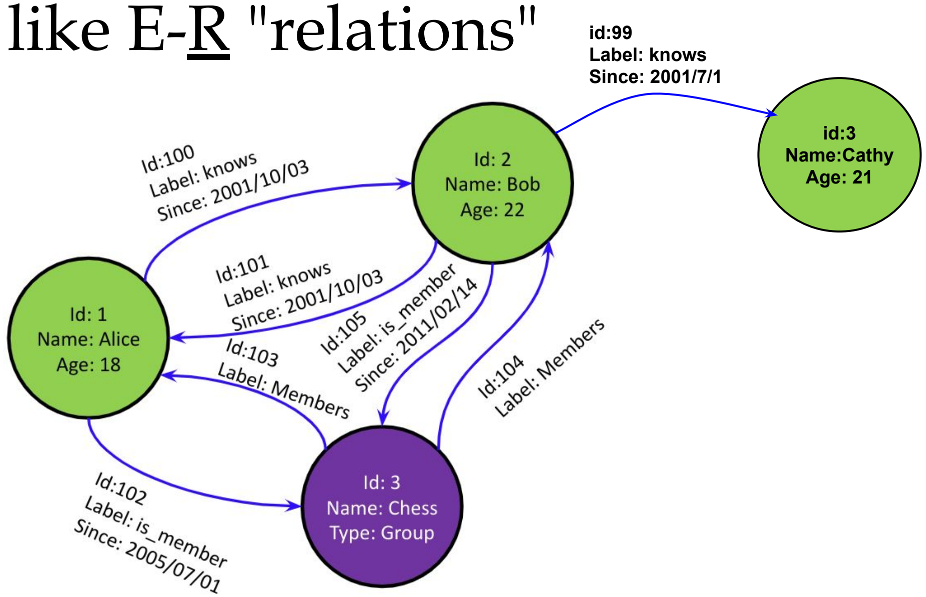


Document Stores

- 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 typical file systems
- e.g., MongoDB and CouchDB (both can be queried using MapReduce (more on this next time!))

Graph Databases

- Data are represented as vertices and edges
  - Vertices/Nodes are like E-R "entities"
  - Edges are like E-R "relations"



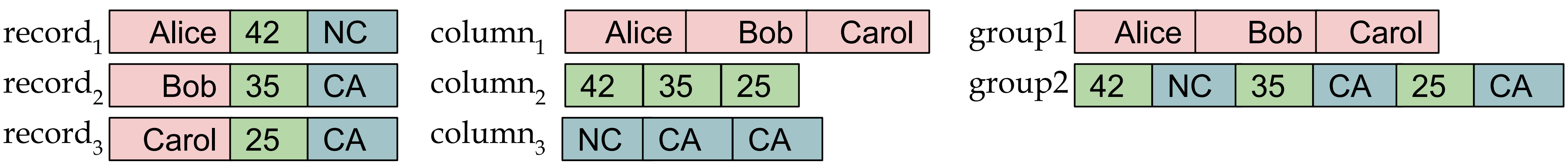
- Graph databases are powerful for graph-like queries (e.g., find the shortest path between two elements)
- E.g., Neo4j and VertexDB

Key-Value Stores

- All the world is one big dictionary, or a collection of dictionaries, of dictionaries, of …
- Keys are mapped to (possibly) more complex types (e.g., lists, dictionaries, sets)
- Keys use hash tables to map to values, hash functions can be distributed easily, even if the data can't be
- Keys map to records, keys map to attribute values, keys map to multisets (not allowed in relational DBs)
- Such stores typically support regular CRUD (create, read, update, and delete) operations
- They don't typically support joins or aggregate functions
- E.g., Amazon DynamoDB and Apache Cassandra

Columnar Databases

- Columnar databases are a hybrid of DBMSs 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 Groups

  - Values are queried by matching keys, to find column indices
- E.g., HBase and Vertica

Summary

- Data can be classified into 4 types, *structured*, *unstructured*, *dynamic* and *static*
- Databases can be scaled *up* or *out*
- Strict consistency limits scalability
- 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
- CAP theorem lead to various designs of databases with *relaxed* ACID guarantees
- NoSQL (databases follow the BASE properties: Basically Available, Soft-State, Eventual Consistency
- NoSQL databases have different types: Document Stores, Graph Databases, Key-Value Stores, Columnar Databases