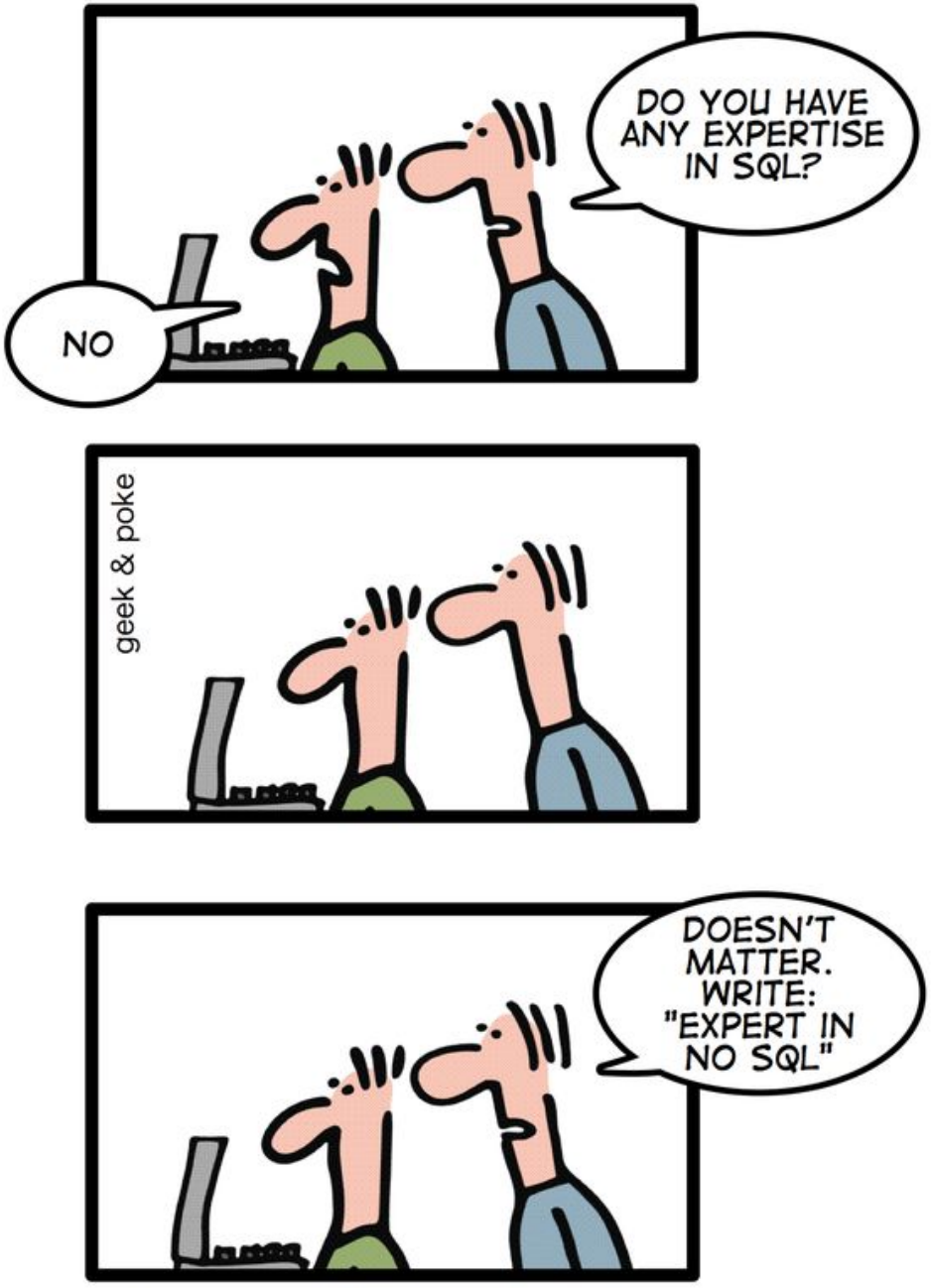
CS595—BIG DATA TECHNOLOGIES

Module 11

NoSQL

Leveraging the NoSQL Boom



Database Alternatives

- OldSQL
  - Legacy relational databases
- NoSQL
  - Give up SQL and ACID for performance
- NewSQL
  - Preserve SQL and ACID
  - Get performance from a new architecture

The Relational Model

- E.F. Codd: (1923-2003)
  - Developed the relational model while at IBM San Jose Research Laboratory
  - IBM Fellow 1976
  - Turing Award 1981
  - ACM Fellow 1994
  - British, by birth

The Relational Model

- "A Relational Model of Data for Large Shared Data Banks," E.F. Codd, Communications of the ACM, Vol. 13, No. 6, June, 1970.
- "Further Normalization of the Data Base Relational Model," E.F. Codd, *Data Base Systems*, Proceedings of 6th Courant Computer Science Symposium, May, 1971.
- "Relational Completeness of Data Base Sublanguages," E.F. Codd, *Data Base Systems*, Proceedings of 6th Courant Computer Science Symposium, May, 1971.
- Plus others…

The Relational Model

- The basic data model:
  - Relations, tuples, attributes, domains
  - Primary & foreign keys
  - Normal forms

"Employee"

ID	Last-Name	Date-of-Birth	Job-Type
15394	Jones	11/3/75	Software
21621	Smith	6/24/69	Management
17852	Brown	8/14/72	Hardware
32904	Carson	10/29/64	Software
	:		
	:		

- Query model:
  - Relational algebra – Cartesian product, selection, projection, union, set-difference
  - Relational calculus

- A primary theme:
  - Physical data independence

Relational Database Management Systems (RDBMS)

- Database Management Systems Based on the Relational Model:
  - System R – IBM research project (1974)
  - Ingres – University of California Berkeley (early 1970's)
  - Oracle – Rational Software, now Oracle Corporation (1974)
  - SQL/DS – IBM's first commercial RDBMS (1981)
  - Informix – Relational Database Systems, now IBM (1981)
  - DB2 – IBM (1984)
  - Sybase SQL Server – Sybase, now SAP (1988)

Structure Query Language (SQL)

- SQL is a language for querying relational databases.

- History:
  - Developed at IBM San Jose Research Laboratory, early 1970's, for System R
  - Credited to Donald D. Chamberlin and Raymond F. Boyce
  - Based on relational algebra and tuple calculus
  - Originally called SEQUEL

SQL Databases

- "SQL" is used both as the name of a language and a type of database
- SQL, the language - Structured Query Language - is designed for managing data in relational database management systems (RDBMS)
- Relational database management systems are often called SQL databases as they use the SQL language
- Since the mid-1980s, SQL has been a standard for querying and managing RDBMS data sets

SQL Databases

- Relational databases continue to provide the foundation for the world's transactions
- Think about all the credit card transactions being handled by the mainframes and large UNIX servers in the data centers of financial services companies
- But web-scale proved to be a great challenge to those traditional RDBMSs…
- You couldn't build a single shared resource machine big enough to handle the demands

Relational Query Languages

- Other Relational Query Languages:
  - Datalog
  - QUEL
  - Query By Example (QBE)
  - SQL variations
  - shell scripts, with relational extensions

Transactions – ACID Properties

- Atomic – All of the work in a transaction completes (commit) or none of it completes
- Consistent – A transaction transforms the database from one consistent state to another consistent state. Consistency is defined in terms of constraints.
- Isolated – The results of any changes made during a transaction are not visible until the transaction has committed.
- Durable – The results of a committed transaction survive failures

SQL Features

- Rely on relational tables
- Utilize defined data schema
- Reduce redundancy through normalization
- Support JOIN functionality
- Engineered for data integrity
- Traditionally scale up, not out
- Rely on a simple, standardized query language
- Near universal in adoption

SQL Database Advantages

- ACID transactions at the database level makes certain business app development easier
- Fine-grained security on columns and rows using views prevents views and changes by unauthorized users
- Most SQL code is portable to other SQL databases, including open source options
- Typed columns and constraints will validate data before it's added to the database and increase data quality
- Existing staff members are already familiar with entity-relational design and SQL

SQL Database Drawbacks

- The object-relational mapping layer can be complex
- Entity-relationship modeling must be completed before testing begins, which slows development
- RDBMSs don’t scale out when joins are required
- Sharding over many servers can be done but is costly and will be operationally inefficient
- Full-text search requires third-party tools
- It can be difficult to store high-variability data in tables

NoSQL Databases

- While NoSQL technologies have existed for decades, they didn't gain popularity until the early 2000s
- Organizations sought solutions to house massive quantities of big data at rest more cheaply than they could with RDBMSs
- And then to be able to handle the higher and higher velocity of incoming data.
- While an SQL (relational) database is a defined, concrete concept, a NoSQL database is not

NoSQL Databases

- There is enormous variation in technologies that fall under the NoSQL category
- So, NoSQL is a term used for a broad group of data management technologies that vary in features and functionality…
- But which address some critical big data challenges of SQL databases

NoSQL Databases

- A NoSQL database…
  - Provides a mechanism for storage and retrieval of data
  - That is modeled in means other than the tabular relations used in relational databases

- A NoSQL database…
  - Is a non-relational…
  - And largely distributed database system…
  - That enables rapid, ad-hoc organization…
  - And analysis of extremely high-volume, disparate data types

BASE Concept

- BASE is a vague term often used as contrast to ACID
  - Basically Available
    - Availability first
    - The system works basically all the time
    - Partial failures can occur, but without total system failure
  - Soft state
    - The system is in flux (unstable), non-deterministic state
    - Changes occur all the time
  - Eventual consistency
    - The system will be in some consistent state
    - At some time in future

NoSQL Features

- High performance writes and massive scalability
- Does not require a defined schema for writing data
- Primarily eventually-consistent by default
- Support wide range of modern programming languages and tools

Why NoSQL?

- NoSQL began as a movement to replace relational databases for specific needs…
- Particularly for gathering and storing massive amounts of data…
- Like all the web activity data of users of sites like Google, Twitter, LinkedIn, and Facebook
- There were no transactions going on -- just collecting and consolidating massive amounts of data on page links
- Also recall that Hadoop came out of Google's need to compute page rankings

Why NoSQL?

- Google could easily throw out most of the historical relational database model…
- Since little of it was needed for their use case
- They wanted to store terabytes and then petabytes of data, cheaply and easily…
- And then be able to run batch analytics on all that data
- Other large web-based companies also discovered that traditional transactional systems had too much baggage…
- And offered too few benefits for their particular and distinct use cases

Why NoSQL?

- Google's efforts were made public and documented in a technical paper in 2006…

  Bigtable: A Distributed Storage System for Structured Data, Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2006, pp. 205-218.

Why NoSQL?

- It is important to understand that the authors of this paper focus on solving a set of pragmatic business problems…
- And not exploring the range of theoretical considerations around creating a distributed databases
- Instead they leverage a Google system called “Chubby” for distributed application management
- Chubby functions much the same as Apache Zookeeper which we encountered in our discussions of Hadoop
- But otherwise issues of data consistency and availability are considered in an engineering context
- So, NoSQL databases, while they can be understood via distributed system theories, are first business software

Why NoSQL?

- NoSQL solutions are primarily about getting rid of structure
- Part of the power of relational database systems is the "relational" piece
- There are strict and governed relations within the database that are embodied in the schema
- You aren't allowed to write arbitrary data into a relational database…
- It has to fit properly into the schema -- and the database software will deny the attempted write if it doesn't

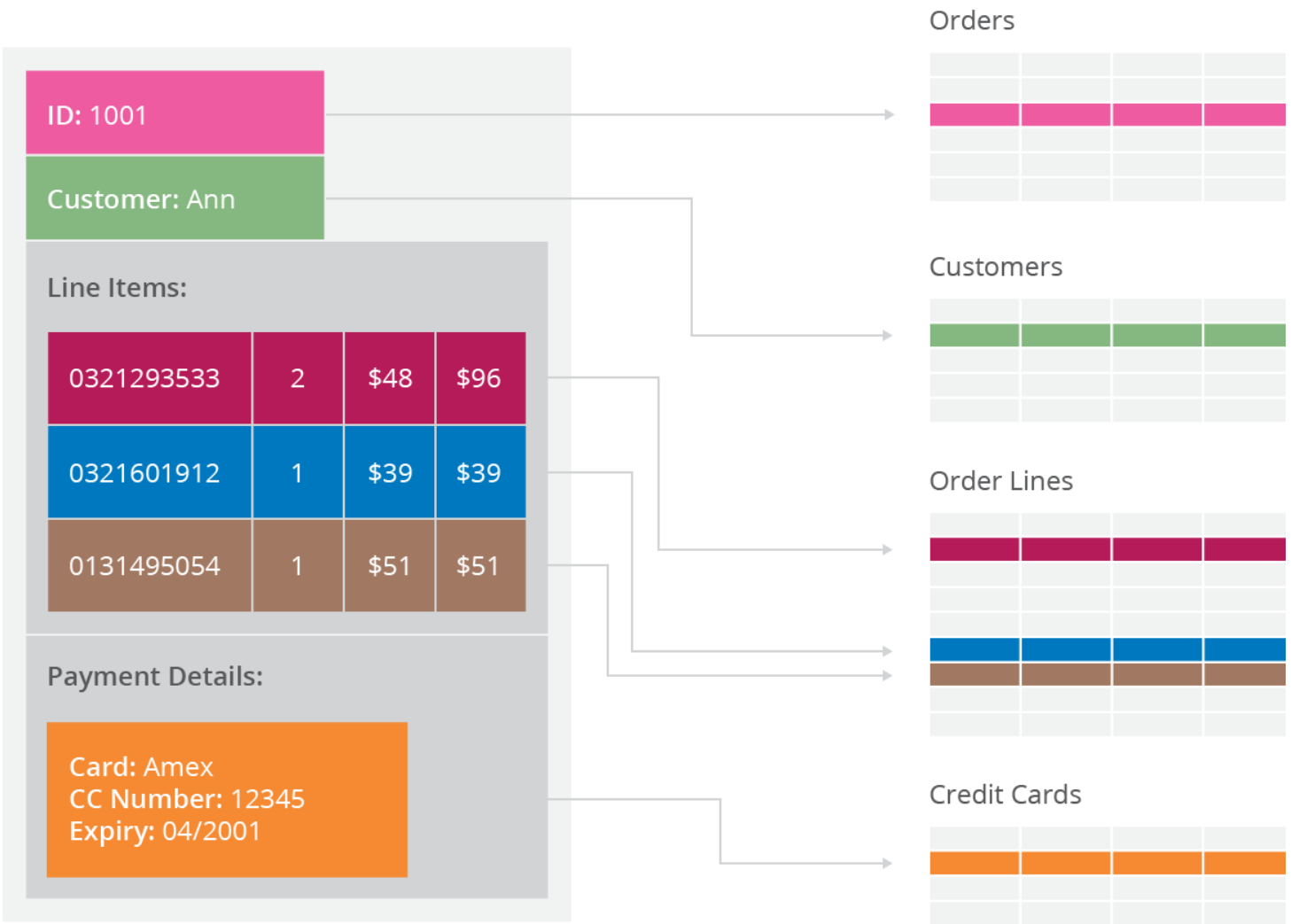
Why NoSQL?

- The primary need of the NoSQL designers was to…
- Collect all the data that was being generated by and about their users, regardless of the type of data
- They didn't want to be encumbered by schema, at least not when the data was being written ("schema on write")

Why NoSQL?

- In order to analyze the data, you do need to know what and where the data you are trying to analyze is
- NoSQL solutions in effect defer the schema question -- they are happy to try to figure out the schema of the data when they get around to analyzing it ("schema on read") rather than when writing it
- So not only do they not need to get all the interested parties to agree on what the data is and where it fits ahead of time
- It provides flexibility in allowing all those parties to view the data differently when they want to consume it

Why NoSQL?
Impedance Mismatch



Why NoSQL?
Impedance Mismatch

- Application developers have been frustrated with the impedance mismatch between relational data structures and the in-memory data structures of the application
- Using NoSQL databases allows developers to develop without having to convert in-memory structures to relational structures

Why NoSQL?
Impedance Mismatch (In Depth)

- Relational database modelling is vastly different than the types of data structures that application developers use
- Using the data structures as modelled by the developers to solve different problem domains has given rise to movement away from relational modelling and towards aggregate models,
  - Most of this is driven by *Domain Driven Design*, a book by Eric Evans
- An aggregate is a collection of data that we interact with as a unit.
- These units of data or aggregates form the boundaries for ACID operations with the database
- Key-value, Document, and Column-family databases can all be seen as forms of aggregate-oriented database

Why NoSQL?
Impedance Mismatch (In Depth)

- Aggregates make it easier for the database to manage data storage over clusters…
- Since the unit of data now could reside on any machine and when retrieved from the database gets all the related data along with it
- Aggregate-oriented databases work best when most data interaction is done with the same aggregate, for example when there is need to get an order and all its details
- It better to store order as an aggregate object but dealing with these aggregates to get item details on all the orders is not elegant

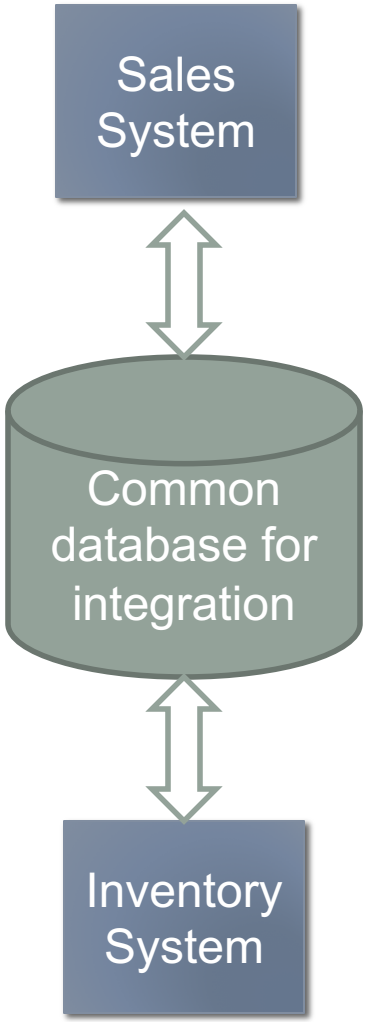
Why NoSQL?
Impedance Mismatch (In Depth)

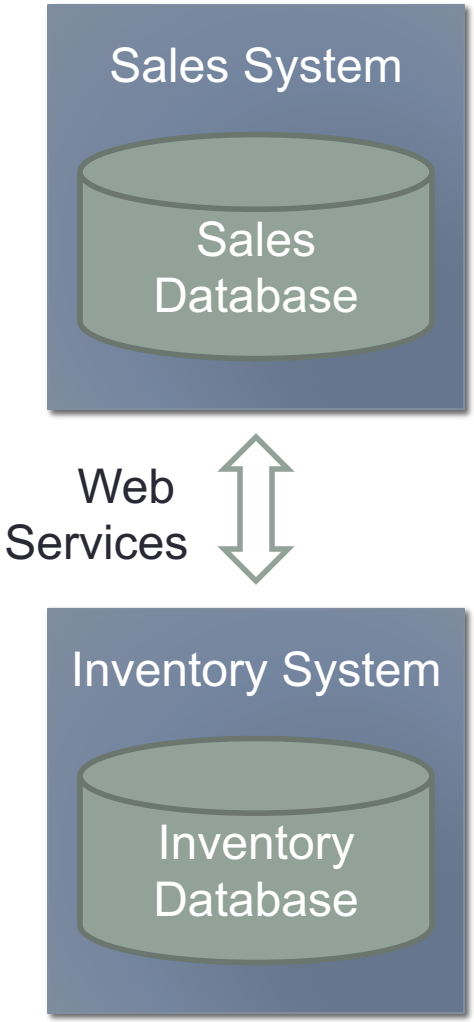
- Aggregate-oriented databases make relationships between aggregates more difficult to handle
  - Foreign keys relationships are often not supported
  - Also joins are not often supported
- Aggregate-oriented databases often support computing (materialized) views to provide data organized differently from the primary aggregates
  - This is often done with some variant of MapReduce computations

Why NoSQL? Services Mismatch

- There is also movement away from using databases as integration points…
- In favor of encapsulating databases with applications and integrating using services (Web Services, Cloud, Docker)
- The rise of the web as a platform also had a profound impact on data storage…
- With the need to support much larger volumes of data by running on clusters
- But typical relational databases were not designed to run efficiently on (large) clusters

Why NoSQL? Services Mismatch





NoSQL Summary

- NoSQL databases reject:
  - Overhead of ACID transactions
  - “Complexity” of SQL
  - Burden of up-front schema design
  - Declarative query expression
  - Yesterday’s technology
- Programmer responsible for
  - Step-by-step procedural language
  - Navigating access path

NoSQL Database Advantages

- Linear scaling takes place as new processing nodes are added to the cluster
- Lower operational costs are obtained by auto-sharding
- There's no need for an object-relational mapping layer
- It's easy to store high-variability data

NoSQL Database Drawbacks

- ACID transactions can be done only within a row or document at the database level
  - Other transactions must be done at the application level
- Document stores don't provide fine-grained security at the element level
- NoSQL systems are new to many staff members and additional training may be required
- Has its own proprietary nonstandard query language, which prohibits portability
- May not work with existing reporting and OLAP tools

SQL or NoSQL: Which is Right for You?

Need	NoSQL	SQL
Do you want a relational database?	No	Yes
Do your records have consistent properties?	No	Yes
Is your data highly variable in structure?	Yes	No
How would you like relationships captured?	Denormalized	Normalized
Do you want to easily join across data?	No	Yes
Is your data structured?	Sort of	Yes

SQL or NoSQL: Which is Right for You?

Need	NoSQL	SQL
How are your schemas?	Dynamic, flexible or none	Static and well defined
Do you require ACID transactions?	No	Yes
How important is consistency?	Varies according to solution	Strong support for consistency
How would you like to scale?	Mostly horizontally	Mostly vertically

NewSQL

More OLTP Throughput, Real-time Analytics

- SQL as the primary mechanism for application interaction
- ACID support for transactions
- A non-locking concurrency control mechanism so real-time reads will not conflict with writes, and thereby cause them to stall.
- An architecture providing much higher per-node performance than available from traditional relational databases
- A scale-out, shared-nothing architecture, capable of running on a large number of nodes without bottlenecking

Database Summary

- SQL Databases
  - Predefined Schema
  - Standard definition and interface language
  - Tight consistency
  - Well defined semantics
- NoSQL Database
  - No predefined Schema
  - Per-product definition and interface language
  - Getting an answer quickly is more important than getting a correct answer

High Level System Classification

- To abstract from the implementation details of individual NoSQL systems, high-level classification criteria can be used to group similar data stores into categories
- Here we introduce the two most prominent approaches: data models and CAP theorem classes

Data Models

- The most common distinction between NoSQL databases is the way they store and allow access to data
- Each NoSQL database can be categorized as aligning with one of four possible data models
  - Key-value store
  - Document store
  - Wide column store
  - Graph store

Data Models

- Note that SQL databases may also store data from one or more of these categories…
- But in doing so they diverge from the relational model in one way or another and depart from use of standard SQL
- Also such databases do not offer the horizontal scalability and high availability characteristics of a NoSQL database

Key Value Store

- A key-value store consists of a set of key-value pairs with unique keys
- Due to this simple structure, it only supports get and put operations
- As the nature of the stored value is transparent to the database…
- Pure key-value stores do not support operations beyond simple Create, Read, Update, Delete (CRUD)
- Key-value stores are therefore often referred to as schemaless…
- Any assumptions about the structure of stored data are implicitly encoded in the application logic (schema-on-read)…
- And not explicitly defined through a data definition language (schema-on-write).

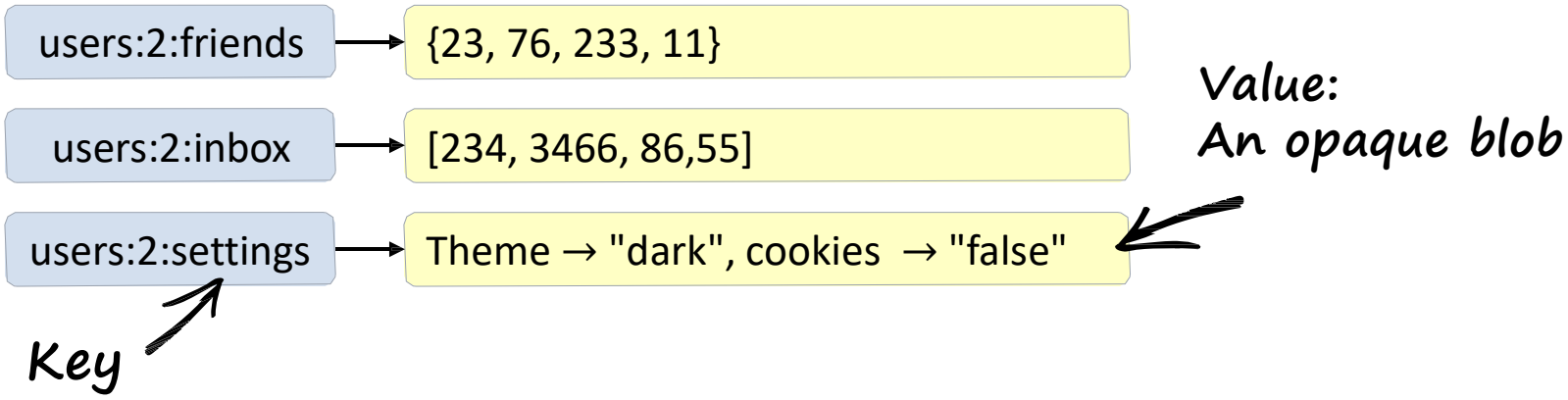
Key Value Store

- The obvious advantages of this data model lie in its simplicity
- The very simple abstraction makes it easy to partition and query the data…
- So that the database system can achieve low latency as well as high throughput
- If an application demands more complex operations, e.g. range queries, this data model is not powerful enough
- Since queries more complex than simple lookups are not supported…
- Data has to be analyzed inefficiently in application code to extract information

Key Value Store

Data model: (key) -> value

- Interface: CRUD (Create, Read, Update, Delete)

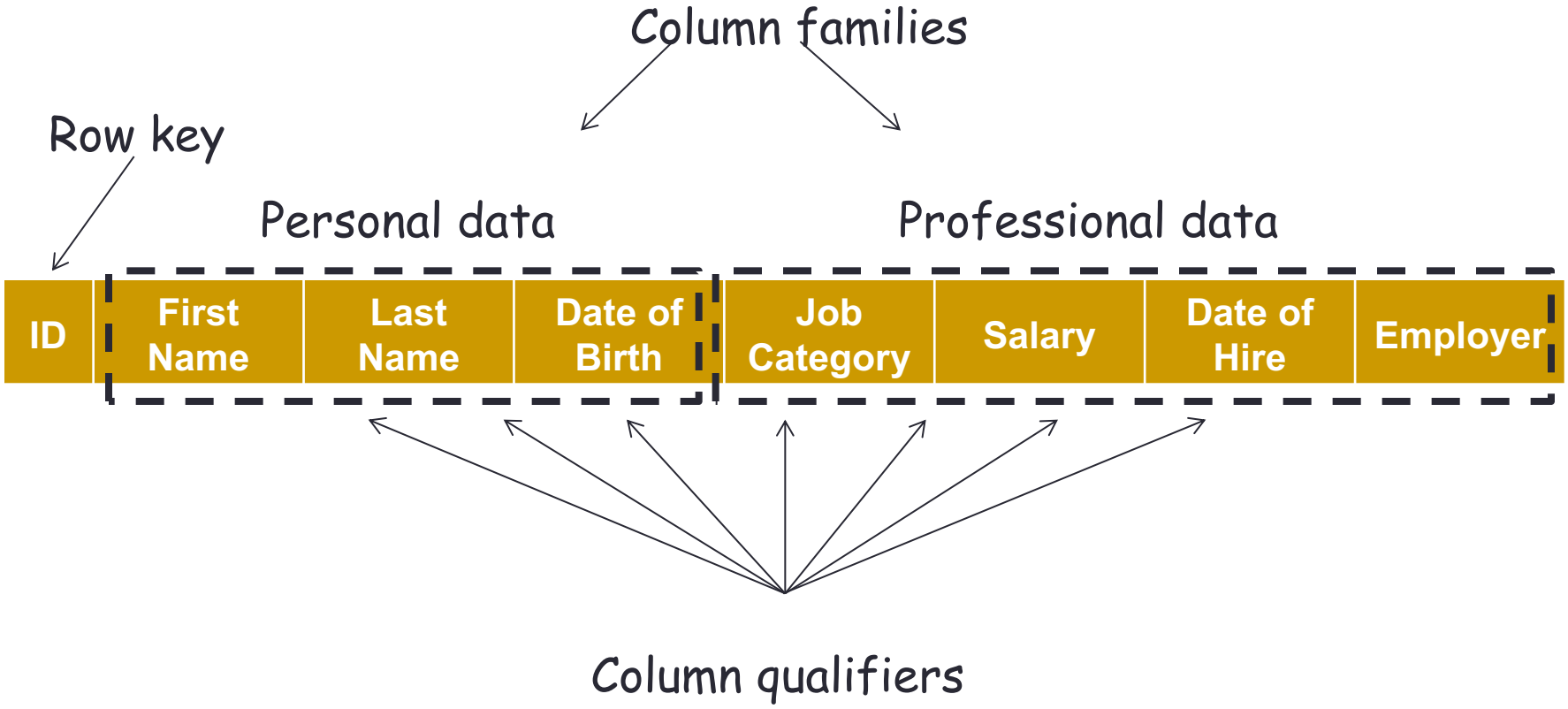


- Examples: Amazon Dynamo (AP), Riak (AP), Redis (CP)

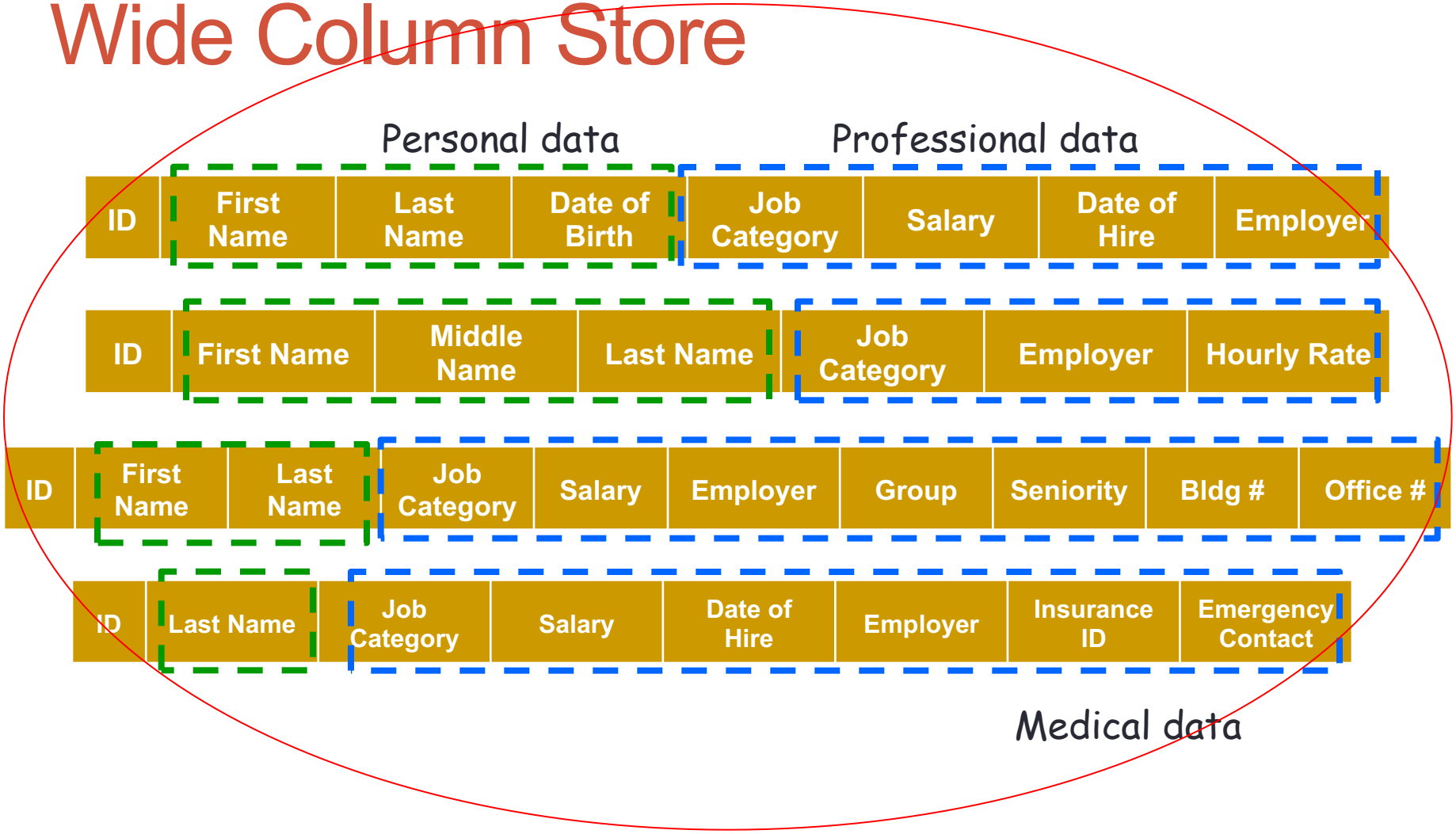
Wide Column Store

- Wide column stores extend key value stores by imposing some structure on the value linked to each row key
- This structure has four levels: column, column family, row, table
- A column is something like a name value pair and a column family is a named group of columns

Wide Column Store

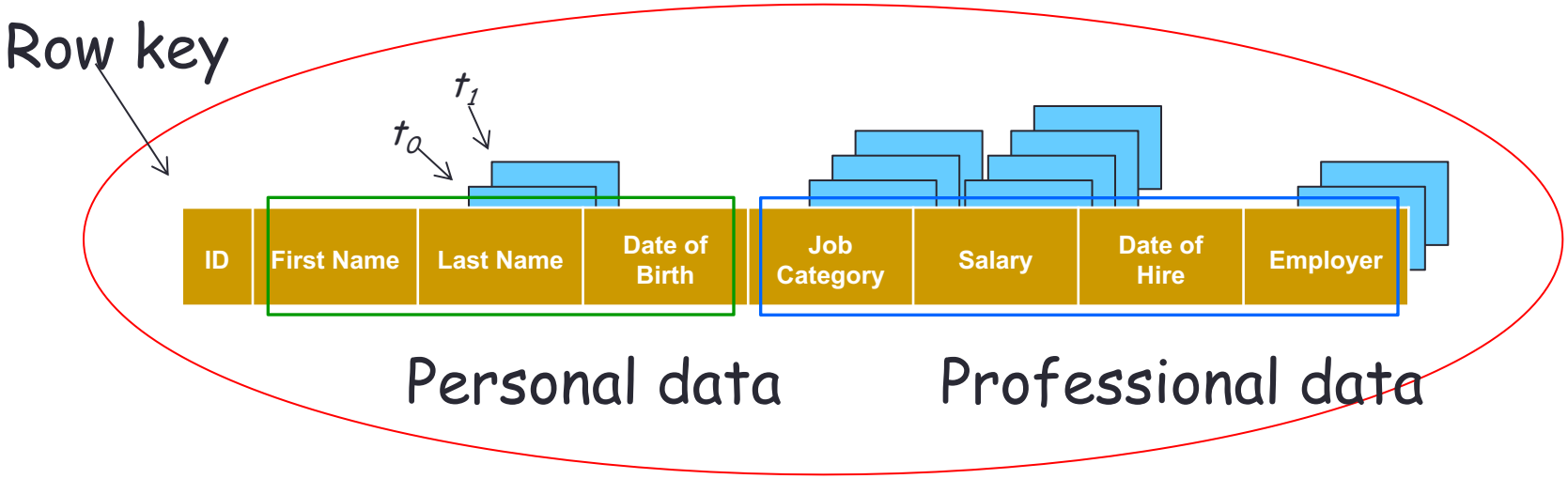


Wide Column Store



One "table"

Wide Column Store



One "row"

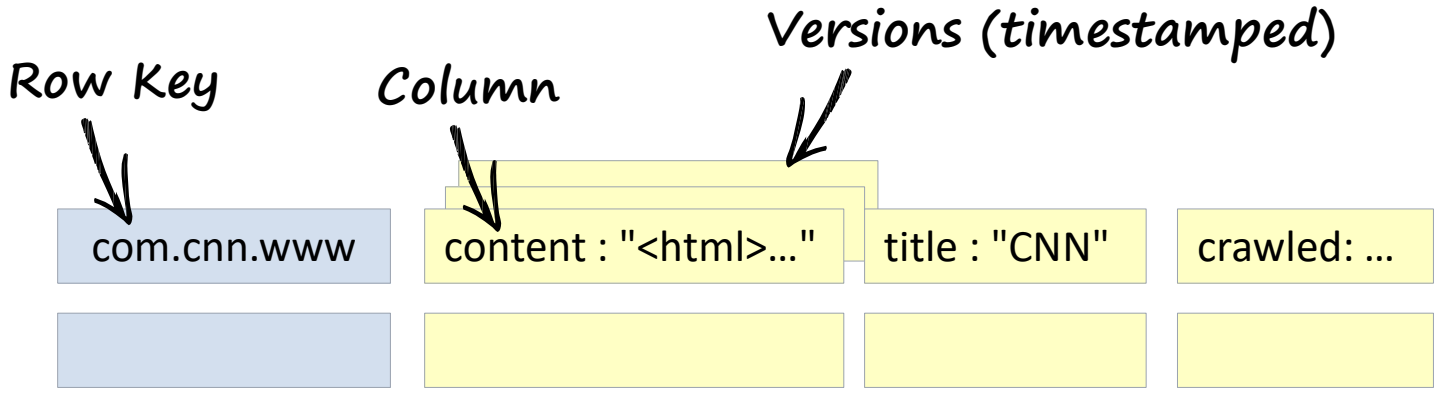
One "row" in a wide-column NoSQL database table
=
Many rows in several relations/tables in a relational database

Wide Column Store

- A sparse storage scheme makes tables with arbitrarily many columns feasible…
- Because there is no column key without a corresponding value
- Hence, null values can be stored without any space overhead
- The set of all columns of a column family is collocated on disk for efficient access together
- On disk, wide-column stores do not group all data from each row together…
- But instead group values of the same column family and from the same row
- So, a row cannot be retrieved by one single lookup as in a document store, but has to be joined together…
- Each column facility acts somewhat like a separate table in a relational database

Wide Column Store

- Data model: (rowkey, column, timestamp) -> value
- Interface: CRUD, Scan

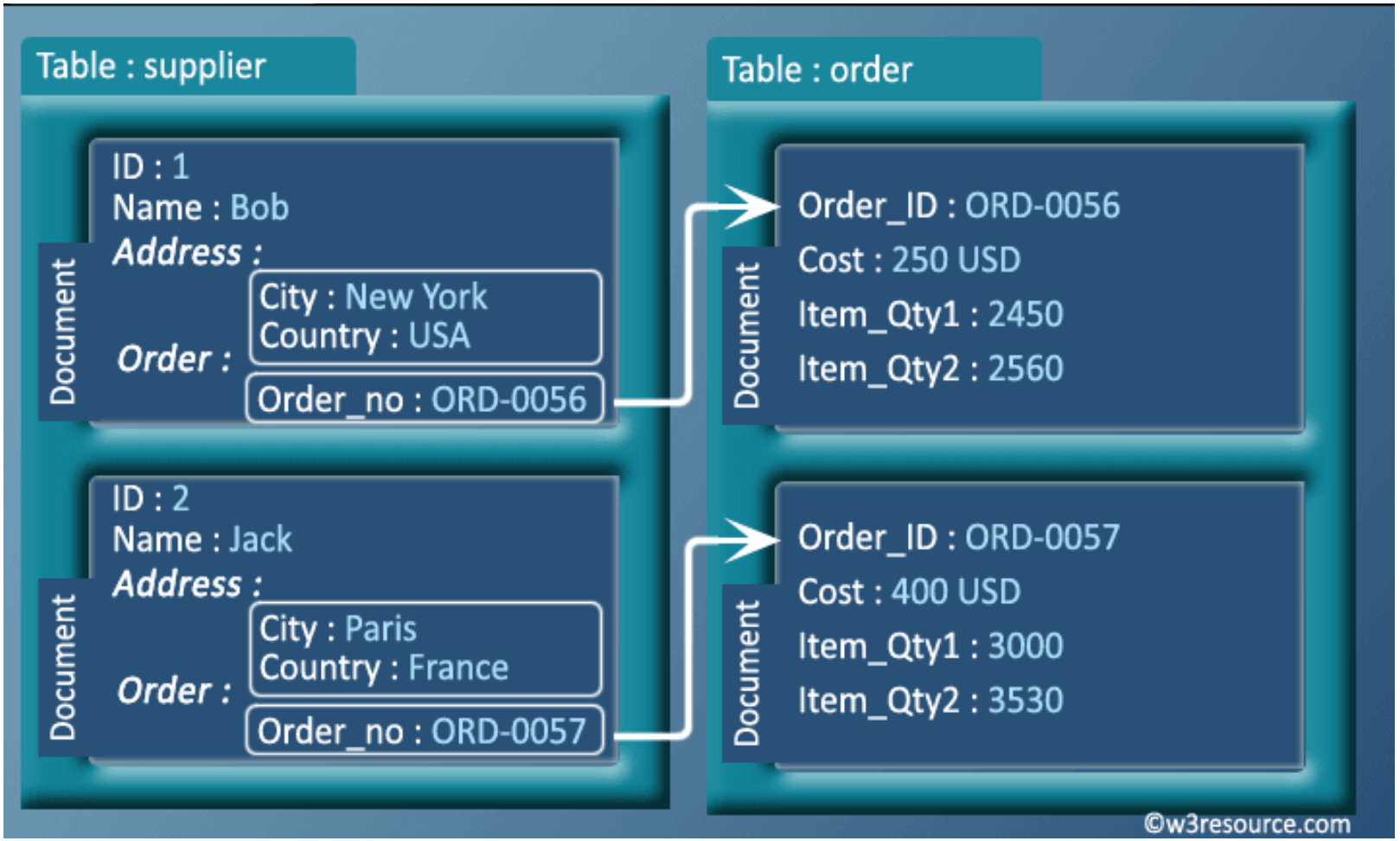


Examples: Cassandra (AP), Google BigTable (CP), HBase (CP)

Document Stores

- A document store is a key-value store that restricts values
- to semi-structured formats such as JSON documents
- This restriction, in comparison to key-value stores, brings great flexibility in accessing the data
- It is not only possible to fetch an entire document by its key, but also to retrieve only parts of a document…
- And to also execute range queries or full-text searches
- Unlike traditional relational databases, the schema for each document, even in the same collection, can vary
- Documents are grouped into “collections,” which serve a similar purpose to a relational table
- A document database provides a query mechanism to search collections for documents with particular attributes

Document Stores



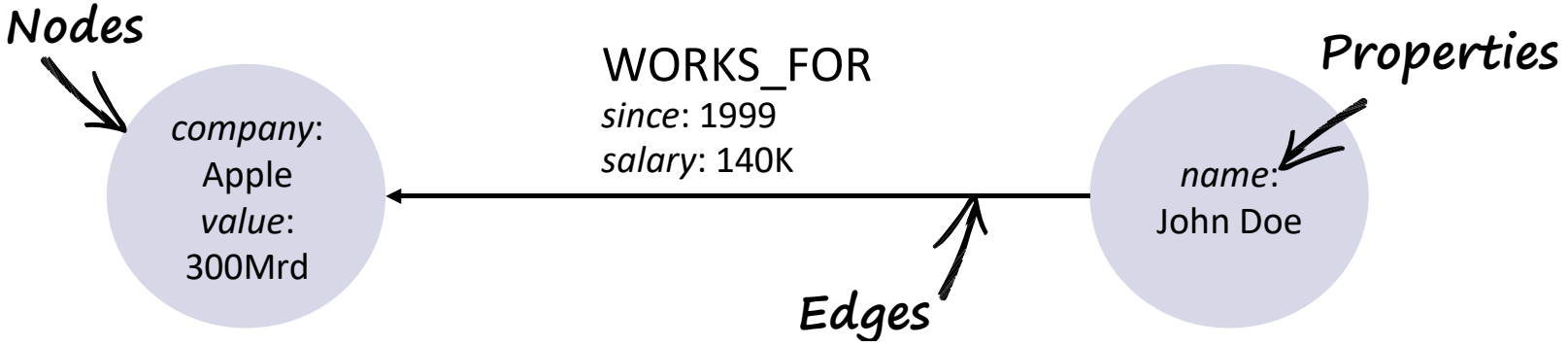
Graph Stores

- Graph databases allow you to store entities and relationships between these entities
- Entities are also known as nodes (vertices), which have properties
- Think of a node as an instance of an object in the application
- Relations are known as edges that can have properties
- Nodes are organized by relationships which allow you to find interesting patterns between the nodes
- The organization of the graph lets the data be stored once and then interpreted in different ways based on relationships.
- In graph databases, traversing the joins or relationships is very fast
- The relationship between nodes is not calculated at query time but is actually persisted as a relationship
- Traversing persisted relationships is faster than calculating them for every query.

Graph Stores

Data model: G = (V, E): Graph-Property Model

- Interface: Traversal algorithms, queries, transactions



Examples: Neo4j (CA), InfiniteGraph (CA), OrientDB (CA)

Consistency Versus Availability

- Another defining property of a NoSQL (distributed) database, is the level of functionality that it provided during a network partition
- Network Partition
  - The network stops delivering messages between subsets of servers
  - Clients may access servers in each partition but these servers cannot communicate with one another
  - So a write to servers in one partition cannot be shared with servers in other partitions

Consistency Versus Availability

- Some databases are built to favor consistency under a network partition while others favor availability
- This trade-off is inherent to every distributed database system
- The huge number of different NoSQL systems shows that there is a wide spectrum between the two paradigms
- In the following, we explain two theorems, CAP and PACELC…
- According to which database systems can be categorized by their respective positions in this spectrum

CAP Theorem

- Conjectured by Prof. Eric Brewer at PODC (Principle of Distributed Computing) 2000 keynote talk
- Described the *trade-offs involved in distributed systems around consistency and availability*

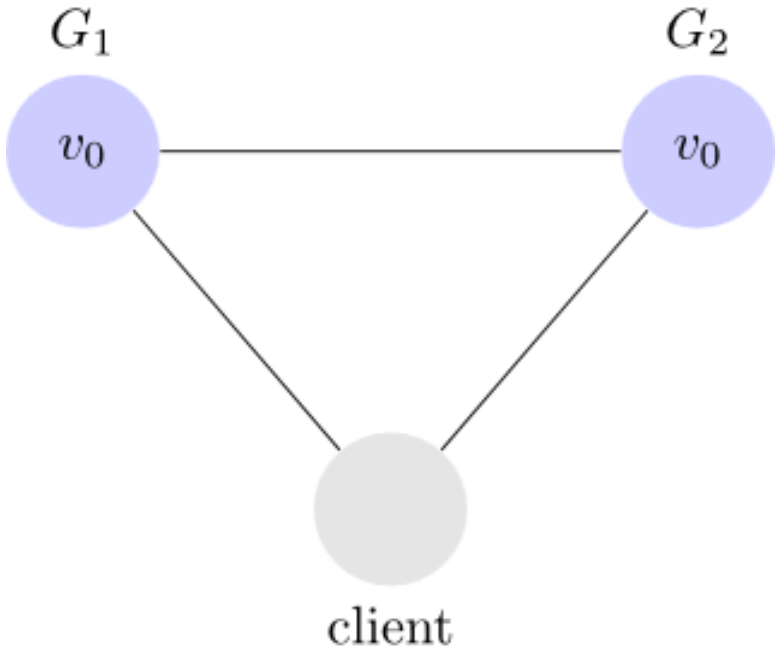
CAP Theorem

- Brewer argues that a system can be both available and consistent in normal operation, but in the presence of a system partition, this is not possible
- If a system continues to work in spite of the partition…
- There is some non-failing node that has lost contact to the other nodes so the system must decide among 2 options
  1. To continue processing client requests to preserve availability (AP)
  2. Or to reject client requests in order to uphold consistency guarantees (CP)
- The first option violates consistency, because it might lead to stale reads and conflicting writes
- While the second option obviously sacrifices availability

Distributed Database System

Model

- Let's consider a very simple distributed system
- Our system is composed of two servers, G1 and G2
- Both servers keep track of the same variable v, with an initial value v0
- G1 and G2 can communicate with each other and can also communicate with external clients



Distributed Database System

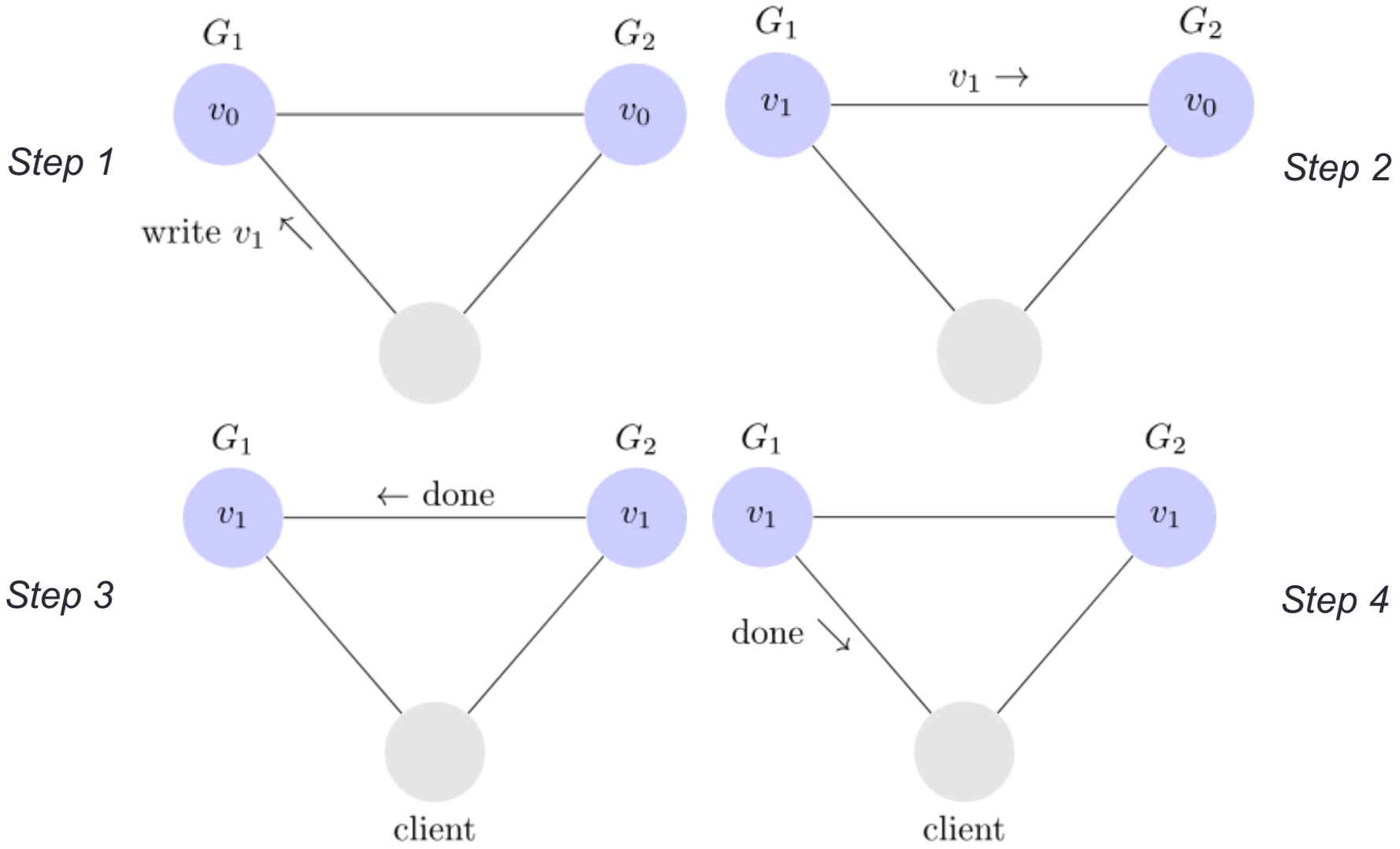
Consistency

- Any read operation that begins after a write operation completes must return that value (or the result of a later write operation)

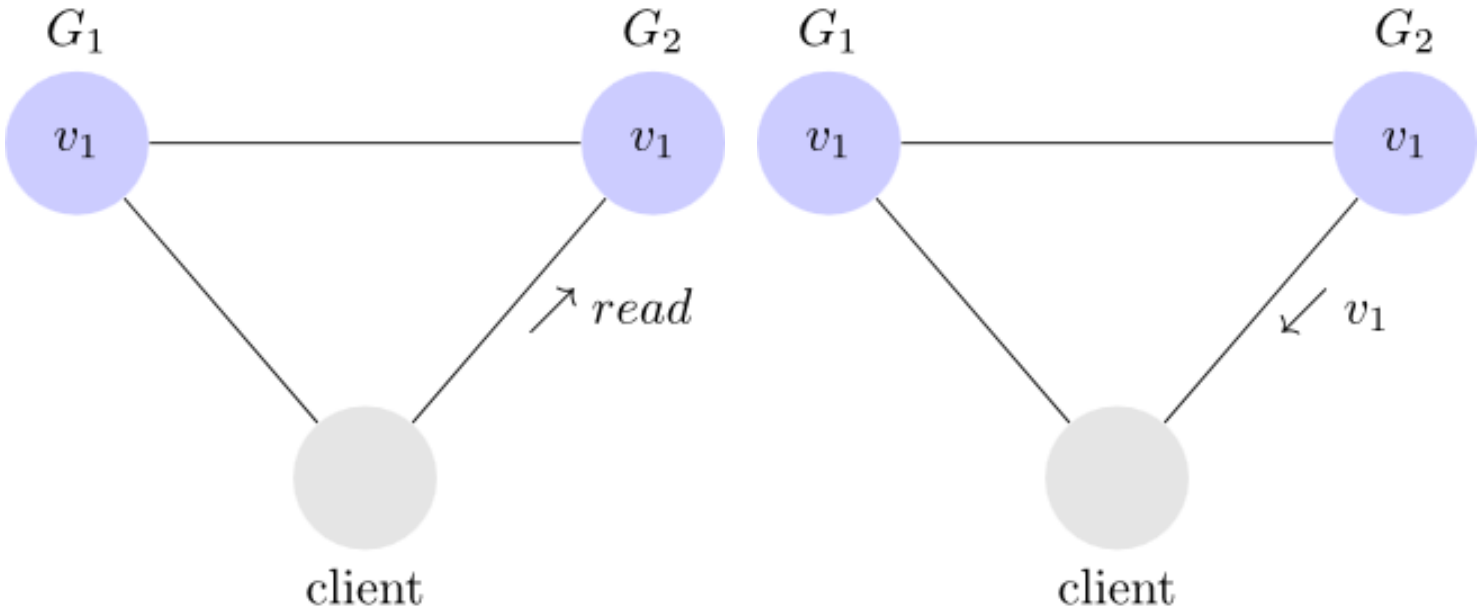
- In a consistent system, once a client writes a value to any server and gets a response, it expects to get that value (or a fresher value) back from any server it reads from.

Distributed Database System

Example of a Consistent System (Write Phase)

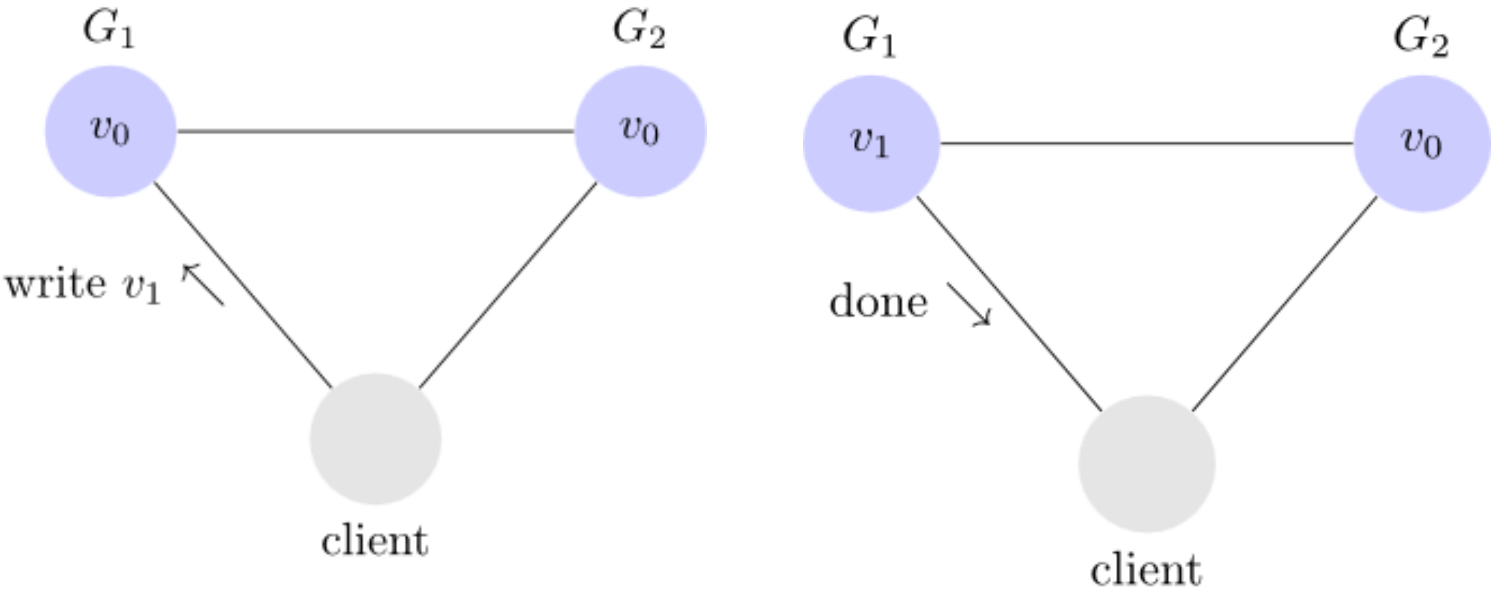


Distributed Database System
Example of a Consistent System (Read Phase)

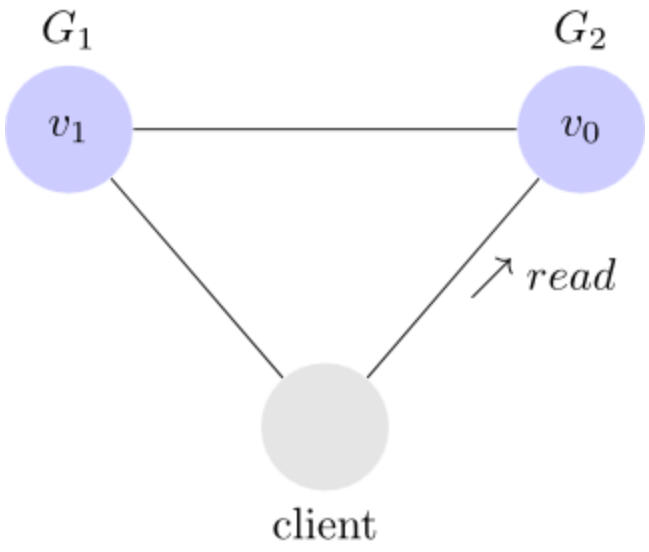


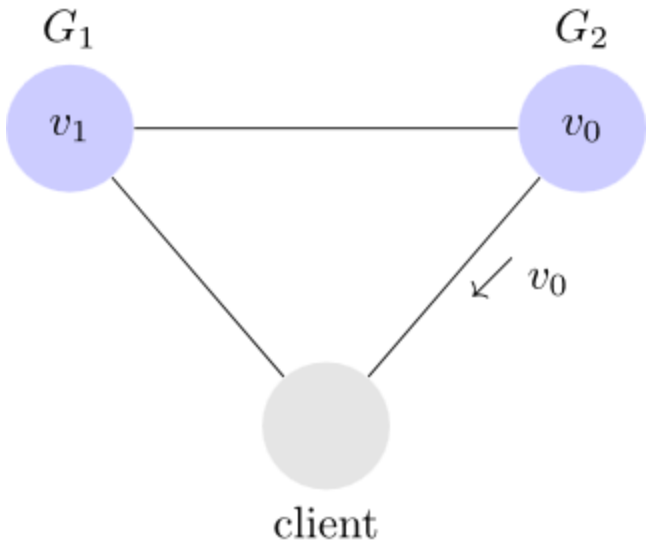
- In this system, G1 replicates its value to G2 before sending an acknowledgement to the client
- When the client reads from G2, it gets the most up to date value of v which is v1

Distributed Database System
Example of a Inconsistent System (Write Phase)



Distributed Database System
Example of a Inconsistent System (Read Phase)





- Our client writes *v*1 to G1 and G1 acknowledges, but when it reads from G2 , it gets stale data v0

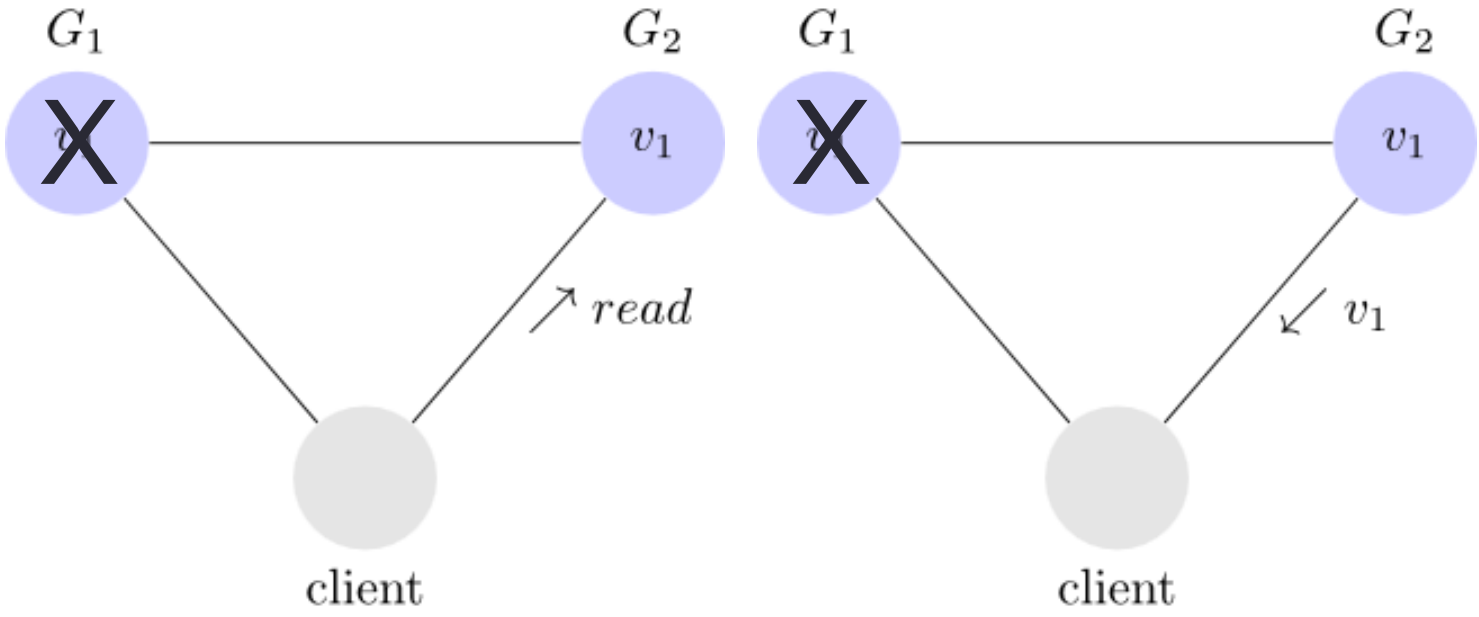
Distributed Database System

Availability

- Every request received by a non-failing node in the system must result in a response
- In an available system, if our client sends a request to a server and the server has not crashed, then the server must eventually respond to the client
- The server is not allowed to ignore the client's requests

Distributed Database System

Example of Availability



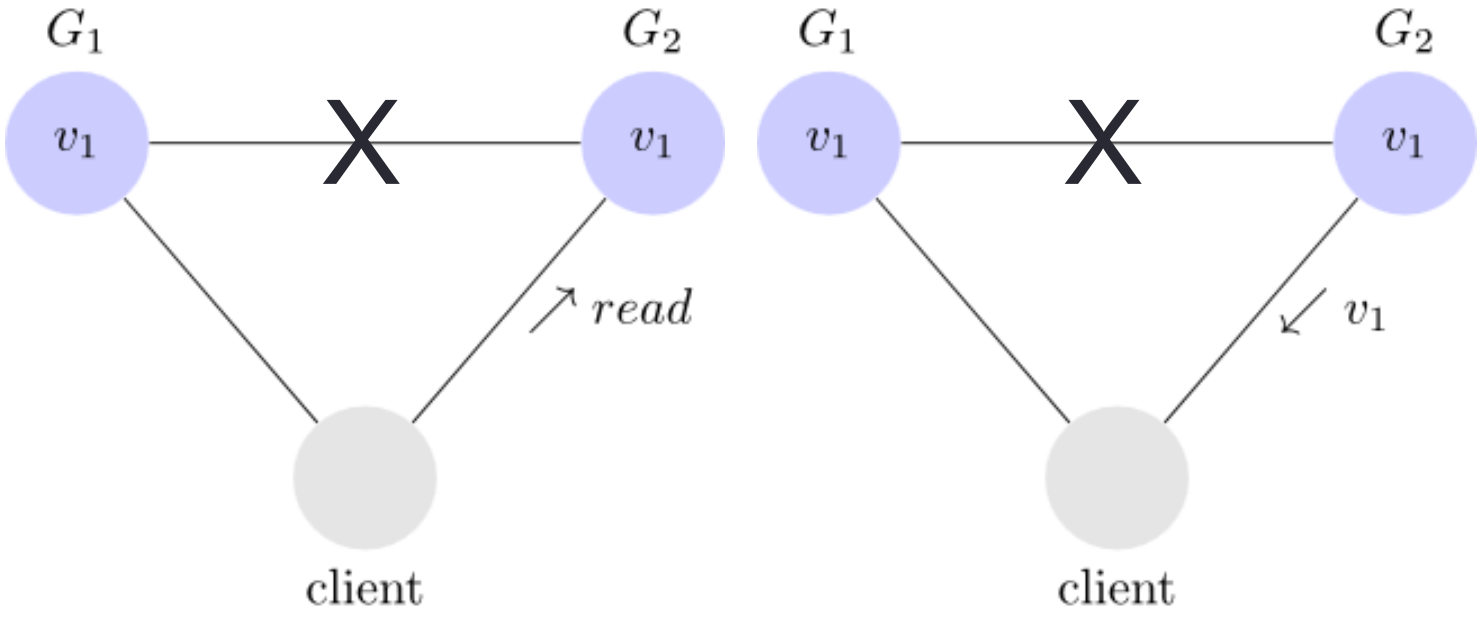
- Even though G1 has failed, this does not impact the ability of G2 to service requests

Distributed Database System
Partition Tolerance

- The network will be allowed to lose arbitrarily many messages sent from one node to another
- This means that any messages G1 and G2 send to one another can be dropped

Distributed Database System

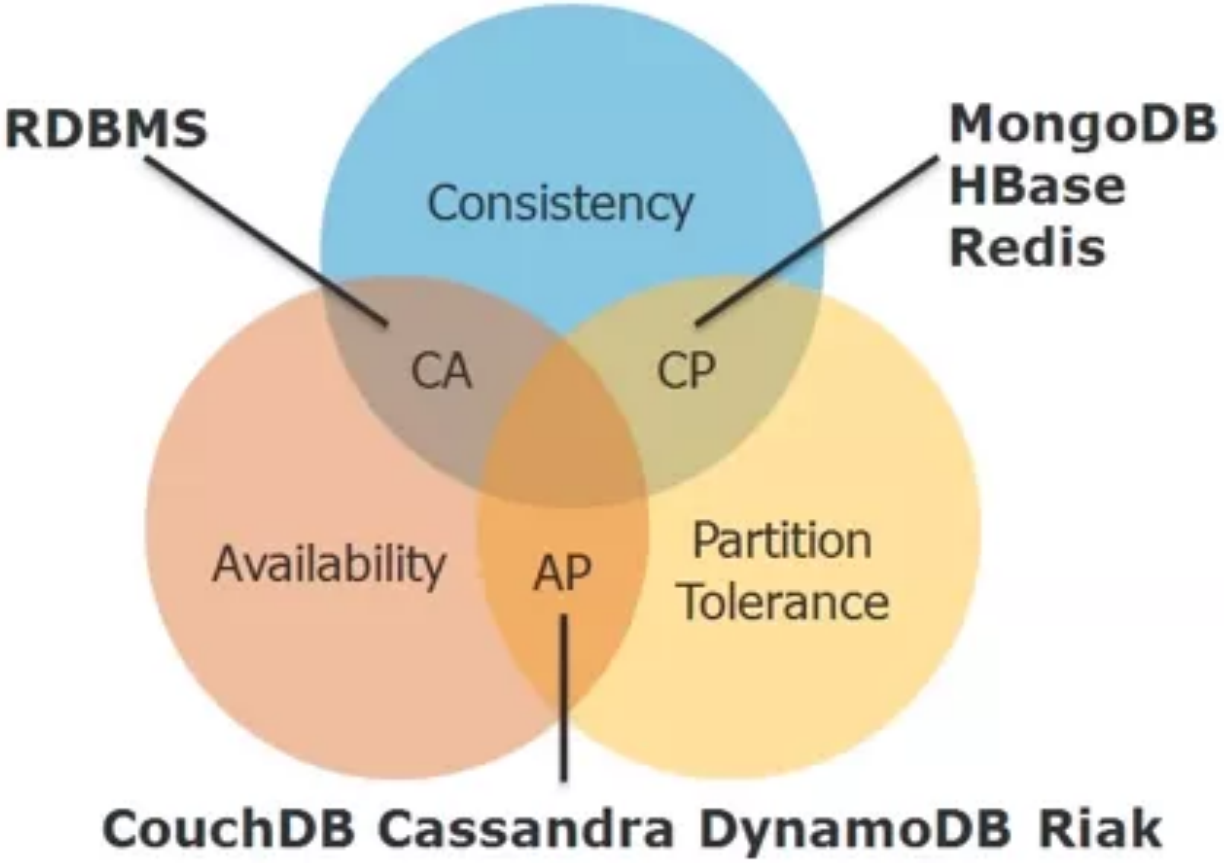
Example of Partition Tolerance



- Even though the connection between G1 and G2 has failed, this does not impact the ability of G2 to service requests

CAP Theorem

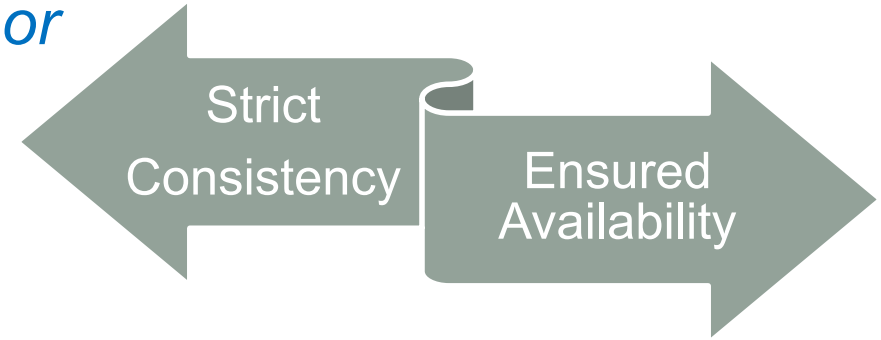
- In the face of a network partition a distributed database must choose to be either available or consistent, but it cannot choose both



CAP Theorem

- The CAP theorem demands you answer this question…

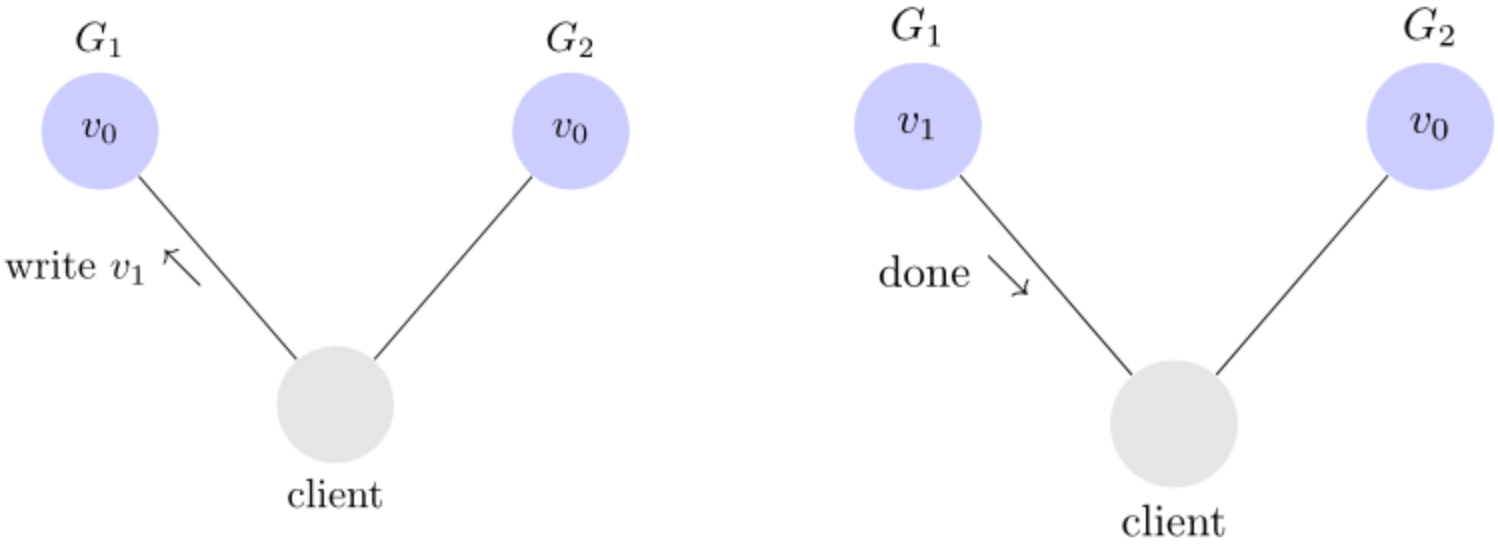
Do I give up strict consistency, or give up ensured availability?



- Do we lock out requests until consistency can be enforced across all nodes?
- Or do we service requests and accept that data on nodes may be (or become) inconsistent?

CAP Theorem

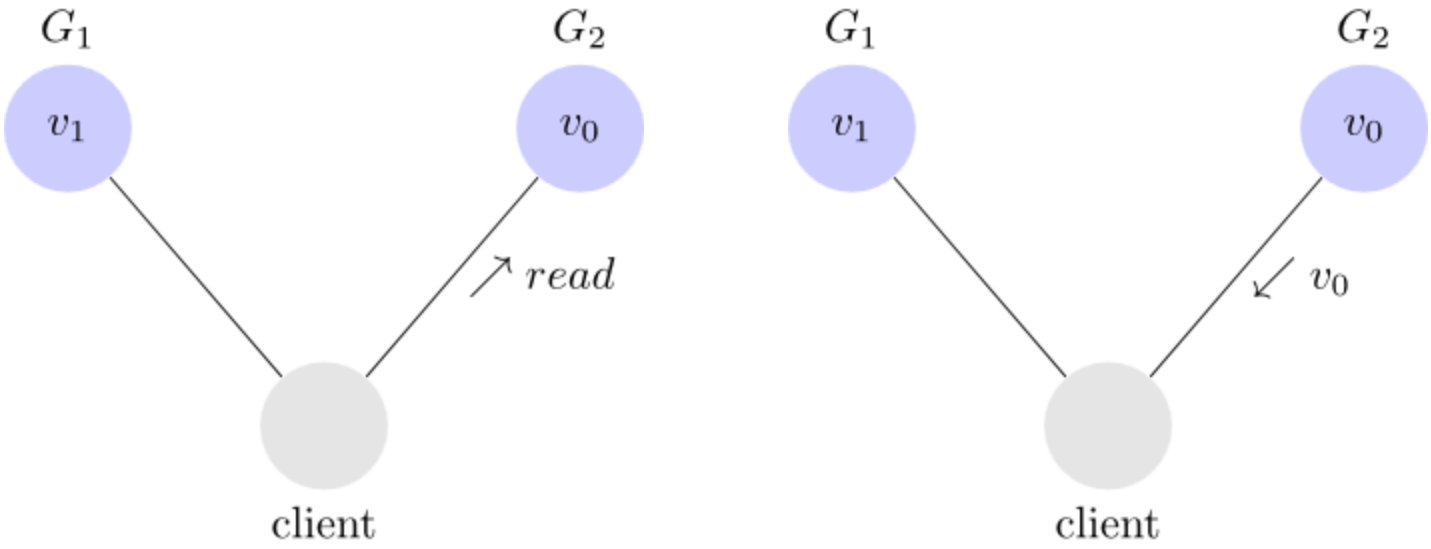
AP: Available, Partition Tolerant, Not Consistent (Write Phase)



- Since our system is available G1 must respond
- Since the network is partitioned G1 cannot replicate its data to *G2*

CAP Theorem

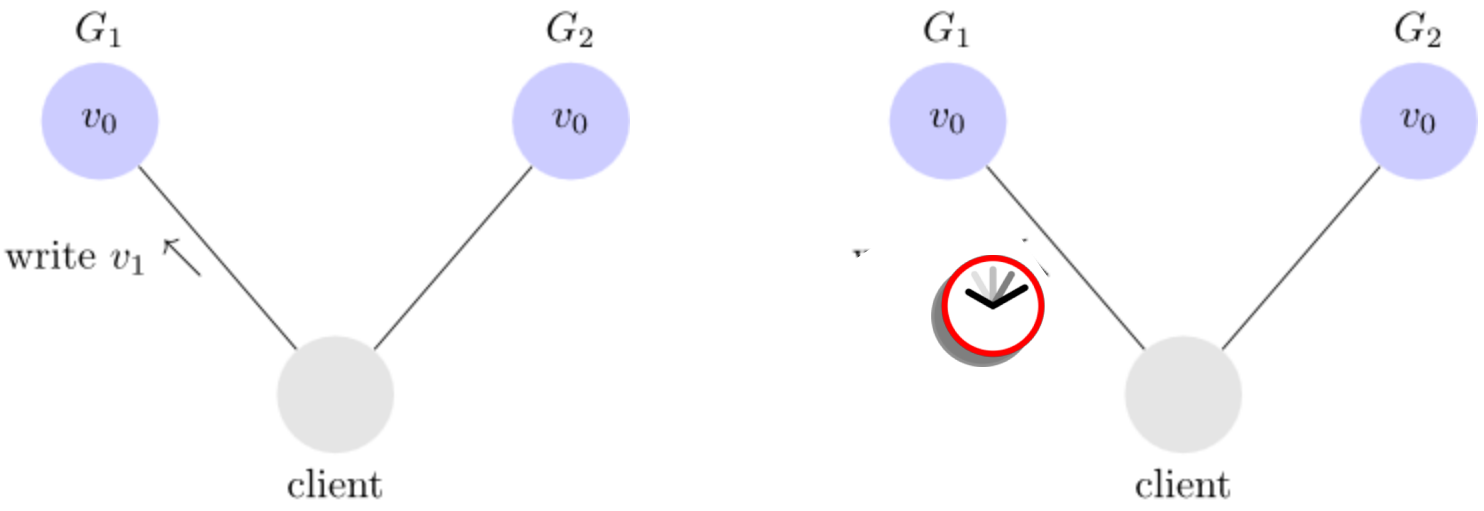
AP: Available, Partition Tolerant, Not Consistent (Read Phase)



- Since our system is available, G2 must respond
- And since the network is partitioned, G2 cannot update its value from G1
- So a client would obtain different values of V depending on whether it issues a read to G1 or G2

CAP Theorem

CP: Consistent, Partition Tolerant, Not Available



- Since the network is partitioned G1 cannot replicate its data to *G2*
- To remain consistent the write must be delayed (or fail) until G1 and G2 can communicate

CAP Theorem

CA: Consistent, Available, Not Partition Tolerant

- There are also systems that usually are available and consistent, but fail completely when there is a partition (CA)
- For example, high performance vertically scalable single node relational systems…
- Or systems which partition data across nodes but without replication

Why this is important?

- The future of databases is distributed
- The CAP theorem describes some of the tradeoffs involved in distributed systems
- A proper understanding of the CAP theorem is essential to making decisions about the future of distributed database design
- Misunderstanding can lead to erroneous or inappropriate design choices

Problem for Relational Database to Scale

- The Relational Database is built on the principle of **ACID** (Atomicity, Consistency, Isolation, Durability)
- It implies that a truly distributed relational database should have **availability, consistency and partition tolerance**.
- Which unfortunately is **impossible** …

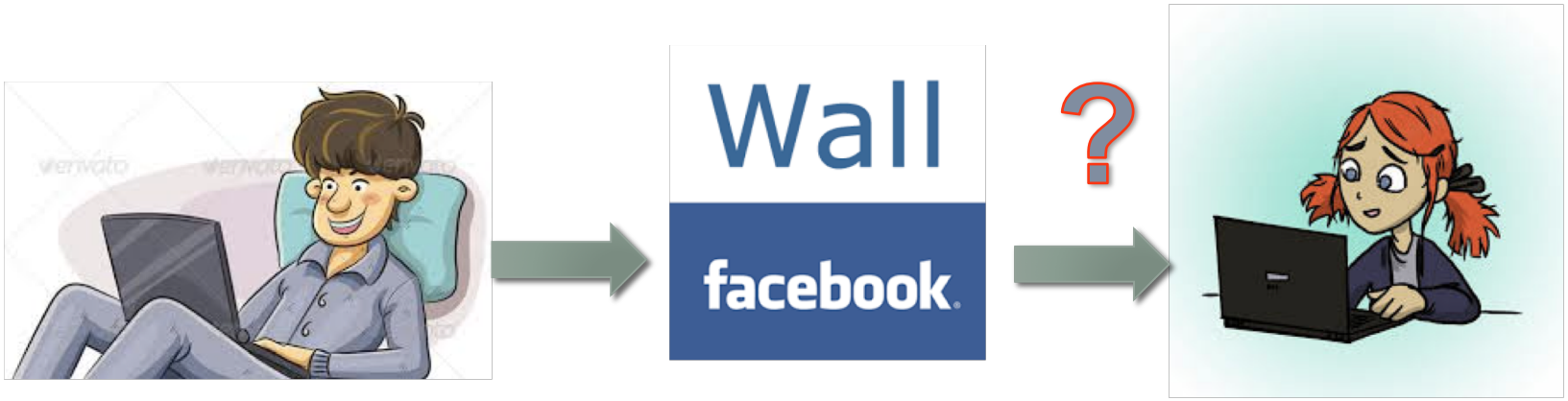
Types of Consistency

- 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 - A Facebook Example

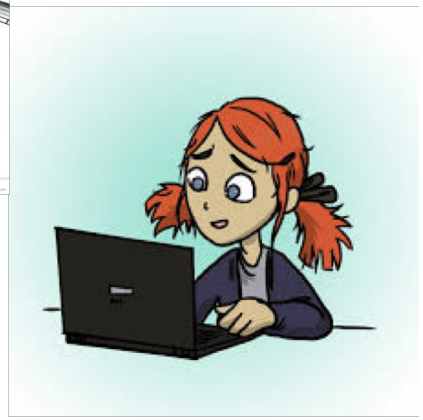
- Bob finds an interesting story and shares with Alice by posting on her Facebook wall
- Bob asks Alice to check it out
- Alice logs in her account, checks her Facebook wall but finds:

  - **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**

Tunable Availability and Consistency

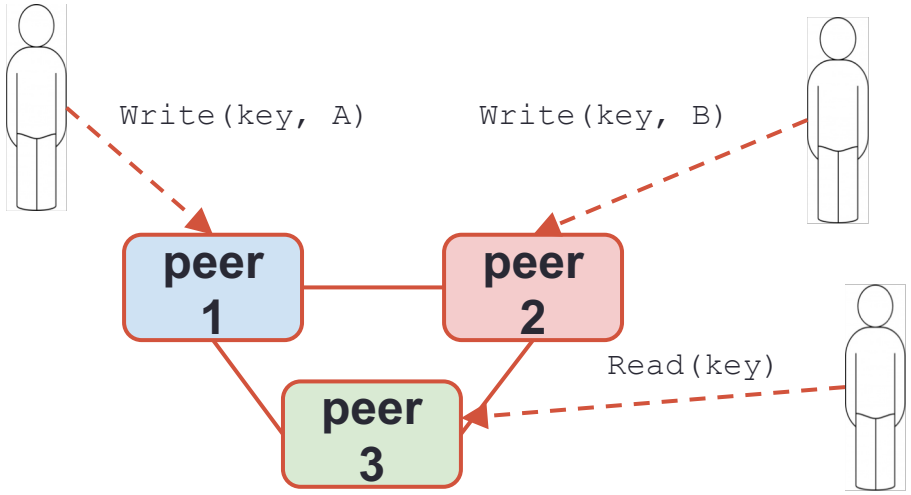
- In practice most distributed databases allow tradeoffs between consistency and availability
- As an example we will consider Riak key-value NoSQL database
- Riak's solution is based on Amazon Dynamo's novel approach of a *tunable* AP system
- Riak is highly available to serve requests, with the ability to tune its level of consistency
- Uses the concept of voting quorum of servers similar to the approach used by Zookeeper

Quorums

- Peer-to-peer replication with replication factor *N*
  - Number of replicas of each data object
- Write quorum: *W*
  - When writing, at least *W* replicas have to agree
  - Having $W > N/2$ results in write consistency
    - in case of two simultaneous writes, only one can get the majority
- Read quorum: *R*
  - Number of peers contacted for a single read
  - For a strong read consistency: $R + W > N$
    - reader surely does not read stale data

Quorums: Example

- Replication factor $N = 3$
- Write quorum: $W = 2$ $(W > N/2)$
  - Write operation waits until *2* nodes acknowledge the write
- Read quorum: $R = 2$ $(R + W > N)$
  - Two nodes contacted for read and thus surely the newest data is returned



Tunable Availability and Consistency

- Riak provides three tuning variables r, w, n where…

- r=number of nodes that should respond to a read request before its considered successful
- w=number of nodes that should respond to a write request before its considered successful
- n=number of nodes where the data is replicated aka replication factor

Tunable Availability and Consistency

- We can tweak the r, w, n values to make the system very consistent by setting r=5 and w=5
  - Each write and read operation was successful for each node
- Now we have made the cluster susceptible to network partitions…
- Since any write will not be considered successful when any node is not responding
- We can make the same cluster highly available for writes or reads by setting r=1 and w=1
  - Each write or read must be successful on a single node
- But now consistency can be compromised since some nodes may not have the latest copy of the data

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
  - 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

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

Durability

- The CAP theorem states that if you get a network partition…
- You have to trade off the availability of data versus the consistency of data
- But the CAP theorem does not speak to other failures such as the lose of a compute node or disk drive
- Durability is that property of a system which operates to preserve its data in the face of such hardware failures
- In this case the tradeoff for distributed database systems is between durability and latency

Durability

- Durability is provided through data replication as we have seen with HDFS which replicates each file block 3 times by default
- But when we write a new value to the database we might need to wait until that value is replicated across those 3 nodes
- If a write request must wait until a new value is replicated across multiple node then we trade off higher latency for durability
- But now assume replication can occur asynchronously to the completion of the write…
- We can decrease latency by allowing the write to complete before data is replicated across fewer than 3 nodes
- However durability may be compromised in this case if not all replicas can be created (perhaps due to a network partition)

CAP Theorem Limitations

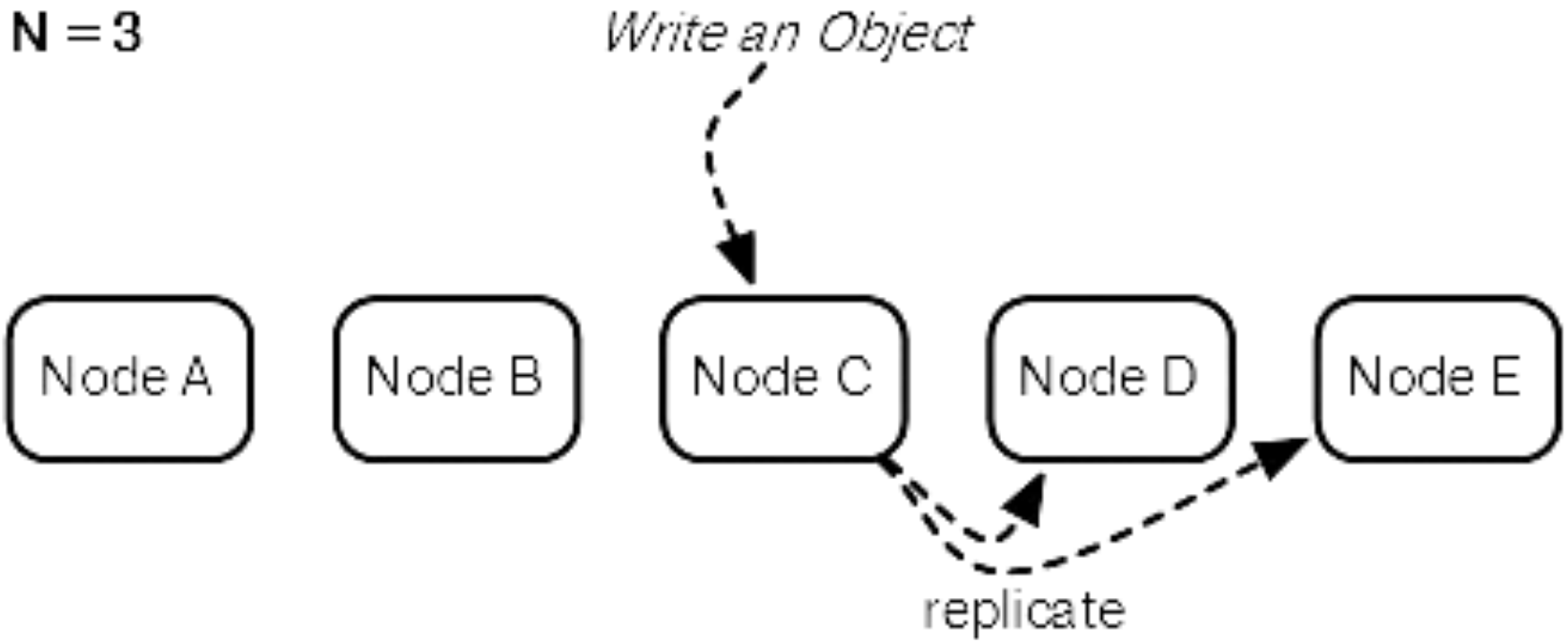
- So the CAP theorem fails to capture the tradeoffs between availability and consistency during normal operations…
- Even though this has proven to be much more influential on the design of distributed databases than the availability consistency tradeoff in failure scenarios

Latency Consistency Tradeoff

- But durability through replication leads to another related tradeoff between latency and consistency
- Between the time a replication is initiated and it completes some replicates may be out of date with others
- So some readers may be provided with stale data while others might be provided with current data
- It depends from which node the reader is provided data
- But the CAP theorem fails to capture tradeoffs between latency and consistency during normal operations…
- But this latency consistency tradeoff, in normal operation, has proven to be more influential on the design of NoSQL databases

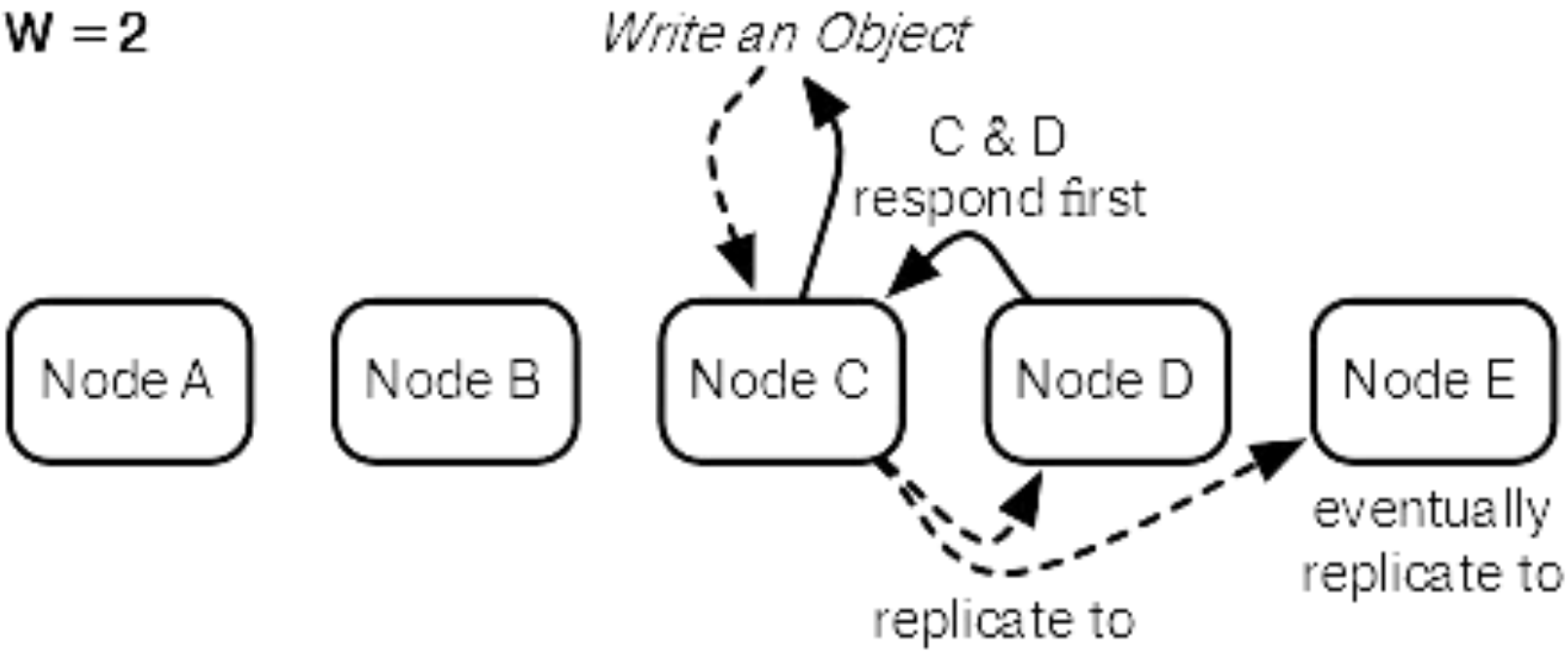
Latency Consistency Tradeoff

Replicate a new value to a total of three separate nodes



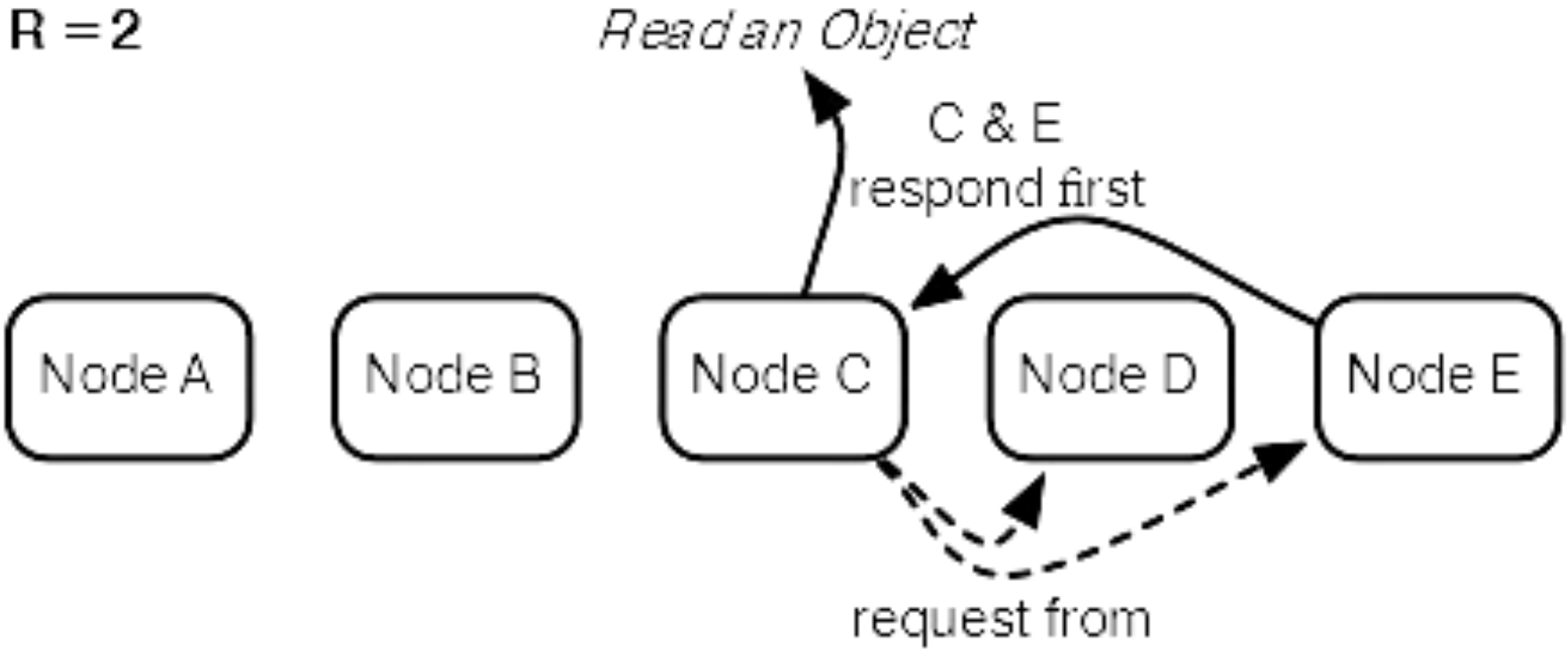
Latency Consistency Tradeoff

Replicate a new value to a total of three separate nodes
But complete a write when data is replicated only two times



Latency Consistency Tradeoff

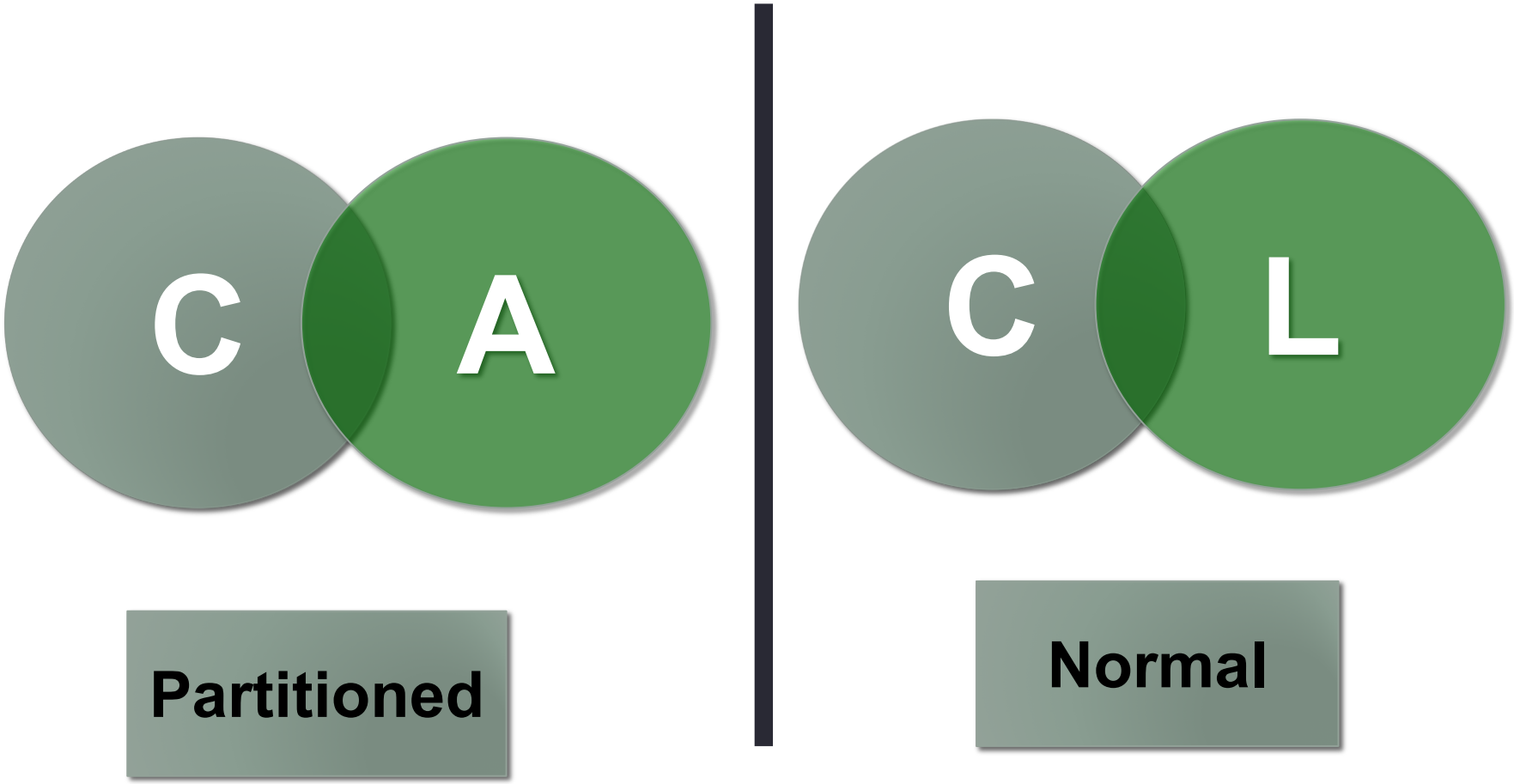
Replicate a new value to a total of three separate nodes
Complete a read when a (consistent) value has been read from only two nodes



PACELC

- The PACELC theorem unifies both latency-consistency and availability-consistency tradeoffs
- And so portrays the design space of distributed systems more accurately
- From PACELC, we learn
  - In case of a ***P***artition, there is an ***A***vailability-***C***onsistency trade-off
  - ***E***lse, in normal operation, there is a ***L***atency-***C***onsistency trade-off
- This classification offers two possible choices for the partition scenario (A/C)
- And also two possible choices for normal operation (L/C)
- Thus PACELC appears more fine-grained than the CAP classification

PACELC



Replication Strategies

- Replication allows the system to maintain some level of availability and durability in the face of system errors
- But storing the same records on different nodes in the cluster introduces the problem of data synchronization
- And so a tradeoff between consistency on the one hand and latency and availability on the other
- Gray propose a classification of different replication strategies according to *when* updates are propagated to replicas and *where* updates are accepted

Replication Strategies

- There are two possible choices of “when”
  - Eager (synchronous) replication propagates incoming changes synchronously to all replicas before a commit can be returned to the client
  - Lazy (asynchronous) replication applies changes only at the receiving replica and passes them on asynchronously
- The great advantage of eager replication is consistency among replicas…
- But it comes at the cost of higher write latency due to the need to wait for other replicas
- Lazy replication is faster, because it allows replicas to diverge; as a consequence, stale data might be served

Data Duplication Models: Overview

- Scaling out = distributing the database on a cluster of servers

- Two orthogonal techniques to data duplication:
  - Replication – the same data is copied over multiple nodes
    - Master-slave or peer-to-peer
  - Sharding – different data chunks are put on different nodes (data partitioning)

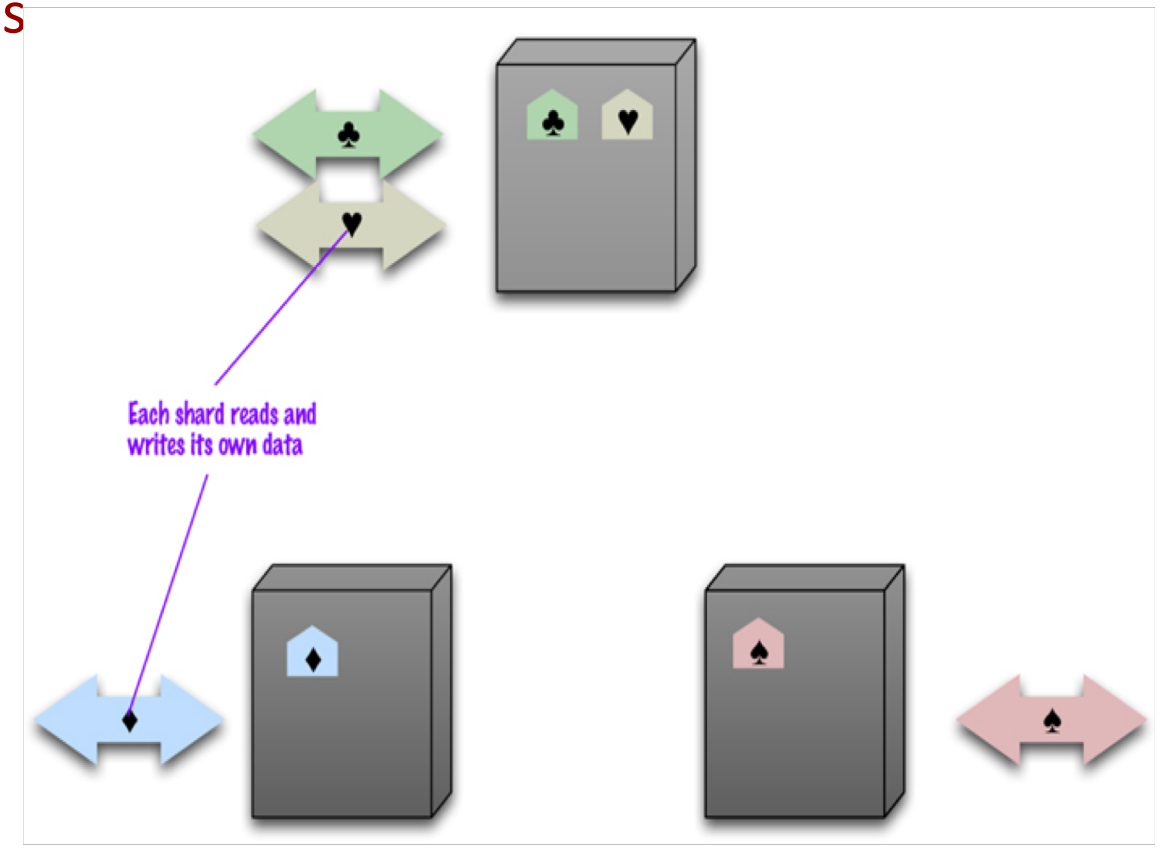
- We can use either or combine them

Data Duplication Models
Single Server

- Running the database on a single machine is always the prefered scenario
  - it spares us a lot of problems
- It can make sense to use NoSQL databases with a single-server distribution model
  - Other advantages remain: Flexible data model, simplicity
  - Graph databases: The graph is “almost” complete → it is difficult to distribute it

Data Duplication Models
Sharding

- Placing different parts of the data onto different servers
- Different people are accessing different parts of the dataset



source: Sadalage & Fowler: NoSQL Distilled, 2012

Data Duplication Models
Sharding (2)

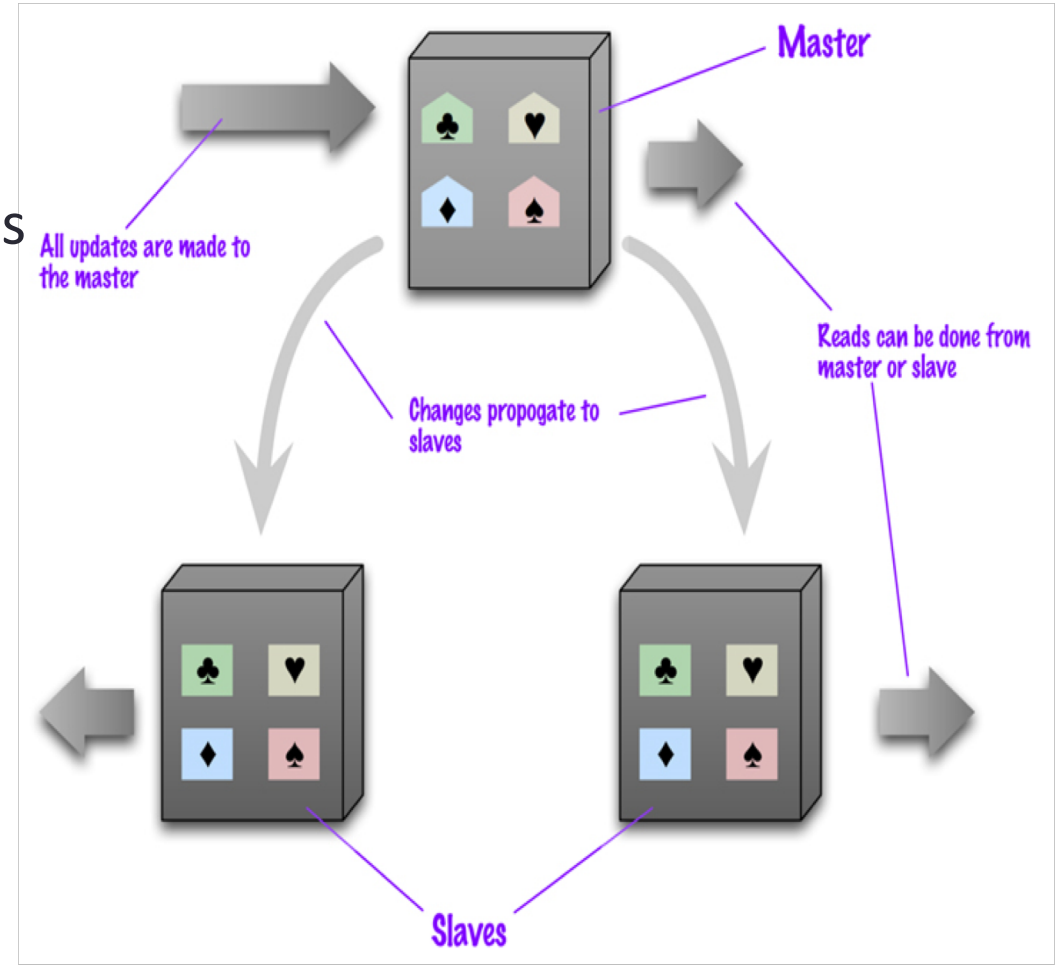
We should try to ensure that

1. Data accessed together is kept together
   - So that user gets all data from a single server
   - Aggregates data model helps to achieve this
2. Arrange the data on the nodes:
   - Based on a physical location (of the data centers)
   - Keep the load balanced (can change in time)

- Many NoSQL databases offer auto-sharding
- A node failure makes shard's data unavailable
  - Sharding is often combined with replication

Data Duplication Models
Master-slave Replication

- We replicate data across multiple nodes
- One node is designated as primary (master), others as secondary (slaves)
- Master is responsible for processing all updates to the data
- Reads from any node

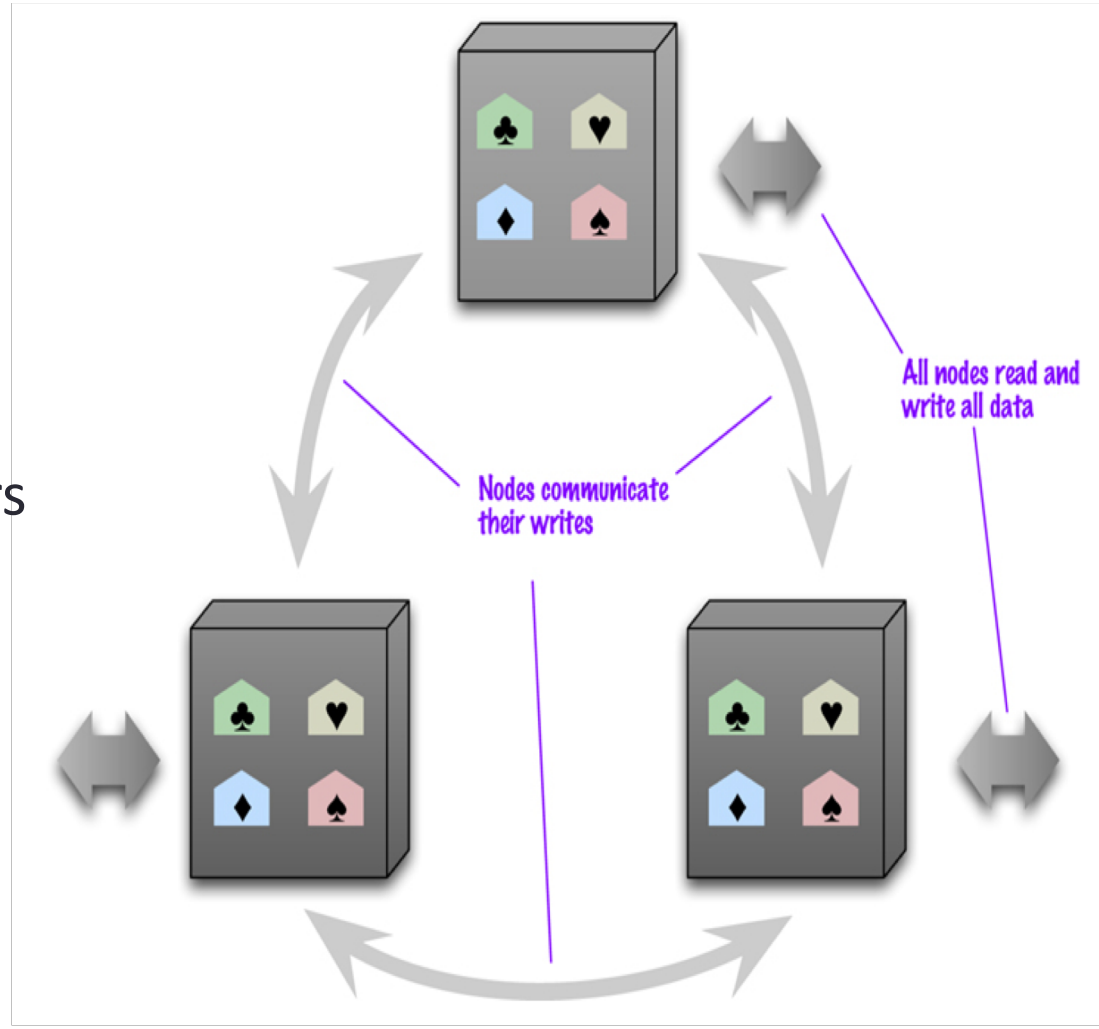


Data Duplication Models
Master-slave Replication (2)

- For scaling a read-intensive application
  - More read requests → more slave nodes
  - The master fails → the slaves can still handle read requests
  - A slave can become a new master quickly (it is a replica)
- Limited by ability of the master to process updates
- Masters are selected manually or automatically
  - User-defined vs. cluster-elected

Data Duplication Models
Peer-to-peer Replication

- No master, all the replicas are equal
- Every node can handle write and then spreads the update to the others



source: Sadalage & Fowler: NoSQL Distilled, 2012

Data Duplication Models
Peer-to-peer Replication (2)

- Problem: consistency
  - Users can write simultaneously at two different nodes
- Solution:
  - When writing, the replicas coordinate to avoid conflict
    - At the cost of network traffic
    - The write operation waits till the process is finished
  - Not all replicas need to agree on the write, just a majority (details below)

Other NoSQL Database Attributes

Sharding

- Range Sharding
- Hash Sharding
- Entity Group Sharding
- Consistent Hash
- Shared Disk

Replication

- Consensus Protocol
- Synchronous
- Asynchronous
- Primary Copy
- Update Anywhere

Storage Management

- Logging
- Updates in Place
- Caching
- In Memory Storage
- Append Only Storage

Query Processing

- Global Secondary Indexing
- Local Secondary Indexing
- Query Planning
- Materialized Views

What Database Should I Choose?