What Are NoSQL Databases

NoSQL stands for **"Not Only SQL."**

NoSQL databases are designed **to store** and **retrieve data in formats** other than the tabular relations used in relational databases. Common types include:

Document-oriented databases (e.g., MongoDB, CouchDB): Store data in document formats like JSON or BSON, making them flexible for hierarchical data.

Key-value stores (e.g., Redis, DynamoDB): Store simple key-value pairs, ideal for high-speed retrieval.

Column-family stores (e.g., Cassandra, HBase): Use column families to group related data, useful in analytics and wide-column storage.

Graph databases (e.g., Neo4j, ArangoDB): Store data as nodes and relationships, well-suited for highly connected data.

When to Use NoSQL vs. SQL

Use SQL/RDBMS if:

- You need strict data consistency in transactions. (Such as banking, finance, and inventory management systems)
- The data structure is well-defined and not likely to change.
- Complex querying and joining of tables are necessary.

Use NoSQL if:

- You need to handle large, unstructured data sets or schema-less data.
- Data models and requirements may change frequently.
- Scalability and high-speed data access are essential.

SQL vs NoSQL

Feature	Relational Databases (SQL)	NoSQL Databases
Data Model	Structured tables with rows and columns (fixed schema)	Various models (document, key-value, column-family, graph)
Schema Flexibility	Fixed schema	Schema-less or flexible schema, easy to change
Scalability	Vertical scaling (add resources to a single server)	Horizontal scaling (add servers, often distributed) Automatic Sharding)
Data Integrity	Strong ACID (Atomicity, Consistency, Isolation, Durability)	Supports BASE (Basically Available, Soft-state, Eventually consistent)
Query Language	Structured Query Language (SQL)	Various, often custom (e.g., MongoDB's BSON, Cassandra CQL)
Use Cases	Applications requiring strict consistency	Big Data, real-time analytics, content management, social apps

Scalability refers to a system's ability to handle increased load or demand by adjusting its resources.

Feature	**Horizontal Scalability**	**Vertical Scalability**
Approach	Add more machines/nodes. Clusters of commodity processors. Moving from serial to parallel processing.	Upgrade a single machine
System Type	Distributed system	Centralized system
Reliability	High (fault tolerance with multiple nodes)	Dependent on single machine's availability
Complexity	Requires distributed management	Simpler to manage, but with single point of failure
Use Cases	Big Data, NoSQL databases, web applications	Relational databases, smaller applications

Vertical Scalability
Vertical scalability (scaling up) involves upgrading the hardware on a single server, such as adding more CPU power, memory, or storage to handle increased load.

As the amount of data an organization stores increases, there may come a point when the amount of data needed to run the business exceeds the current environment and some mechanism for breaking the information into reasonable chunks is required.

Organizations and systems that reach this capacity can use automatic database *sharding* (breaking a database into chunks called *shards* and spreading the chunks across a number of distributed servers) as a means to continuing to store data while minimizing system downtime.

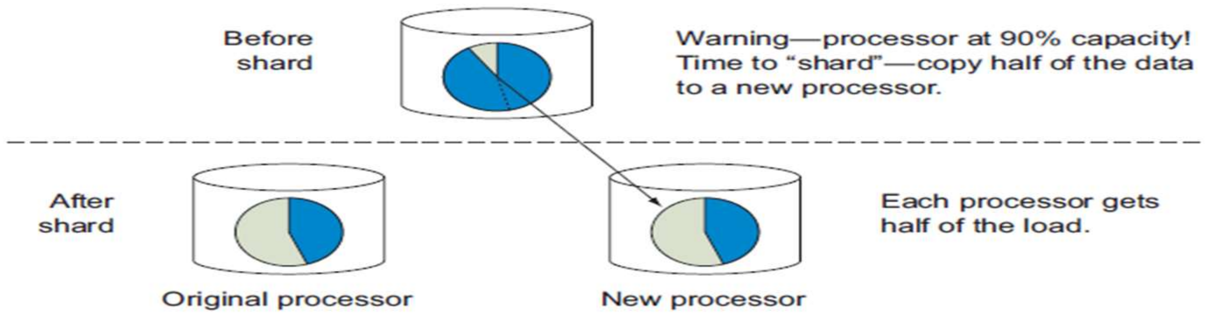

Before shard
Warning—processor at 90% capacity! Time to "shard"—copy half of the data to a new processor.
After shard
Each processor gets half of the load.
Original processor
New processor

Figure 2.9 **Sharding is performed when a single processor can't handle the throughput requirements of a system. When this happens you'll want to move the data onto two systems that each take half the work. Many NoSQL systems have automatic sharding built in so that you only need to add a new server to a pool of working nodes and the database management system automatically moves data to the new node. Most RDBMSs don't support automatic sharding.**

Distributed computing environment (DCE)

Refers to a network of independent computers that work together to achieve a common goal, allowing large-scale processing, storage, and analysis by leveraging multiple machines.

This environment supports **distributed processing** of tasks across systems, making it well-suited for handling big data, high-performance applications, and resource-intensive tasks like scientific simulations, financial modeling, and cloud services.

Comparing ACID and BASE—two methods of reliable database transactions

Let's start with a simple banking example to represent a reliable transaction. These days, many people have two bank accounts: savings and checking. If you want to move funds from one account to the other, it's likely your bank has a transfer form on their website.

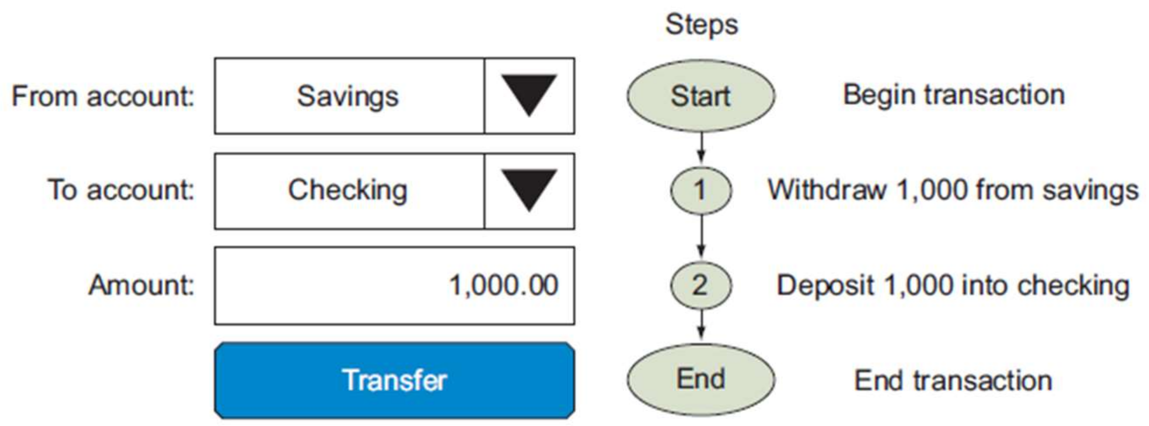


Figure 2.7 The atomic set of steps needed to transfer funds from one bank account to another. The first step subtracts the transfer amount from the source savings account. The second step adds the same transfer amount to the destination checking account. For the transaction to be considered reliable, both steps must work or both need to be undone. Between steps, no reports should be allowed to run that show the total amount as dropping by the transaction amount.

When you click the Transfer button on the web page, two discrete operations must happen in unison. The funds are subtracted from your savings account and then added to your checking account. Transaction management is the process of making sure that these two operations happen together as a single unit of work or not at all.

The software also ensures that no reports can be run on the accounts halfway through the operations. If you run a “combined balance” report during the transaction, it’d never show a total that drops by 1,000 and then increases again. If a report starts while the first part of the transaction is in process, it’ll be blocked until all parts of the transaction are complete.

RDBMS transaction control using ACID Principles

RDBMSs maintain transaction control by using atomic, consistent, independent, and durable (ACID) properties to ensure transactions are reliable

Atomicity—In the banking transaction example, we said that the exchange of funds from savings to checking must happen as an all-or-nothing transaction.

Consistency—In the banking transaction example, we talked about the fact that when moving funds between two related accounts, the total account balance must never change. This is the principle of consistency. It means that your database must never have a report that shows the withdrawal from savings has occurred but the addition to checking hasn’t.

Isolation—Isolation refers to the concept that each part of a transaction occurs without knowledge of any other transaction. For example, the transaction that adds funds doesn’t know about the transaction that subtracts funds from an account.

Durability—Durability refers to the fact that once all aspects of a transaction are complete, it’s permanent. Once the transfer button is selected, you have the right to spend the money in your checking account. If the banking system crashes that night and they have to restore the database from a backup tape, there must be some way to make sure the record of this transfer is also restored.

ACID systems focus on the consistency and integrity of data above all other considerations.

What if you have a website that relies on computers all over the world? A computer in Chicago manages your inventory, product photos are on an image database in Virgina, tax calculations are performed in Seattle, and your accounting system is in Atlanta.

What if one site goes down? Should you tell your customers to check back in 20 minutes while you solve the problem? Only if your goal is to drive them to your competitors. Is it realistic to use ACID software for every order that comes in?

Let’s look at another option. Websites that use the “shopping cart” and “checkout” constructs have a different primary consideration when it comes to transaction processing. The issue of reports that are inconsistent for a few minutes is less important than something that prevents you from taking an order, because if you block an order, you’ve lost a customer.

Non-RDBMS transaction control using BASE principles

BASE principles are specifically designed to suit the scalability and flexibility requirements of NoSQL databases

Basic availability: The system guarantees availability of data, meaning it responds to every request, though it may not reflect the most recent updates. Rather than providing a guarantee of consistency, it ensures that some form of the data will always be accessible

Soft-state: The "Soft State" aspect means that the data within the system may change or be inconsistent for a period of time, even without new input. In other words, the state of the system is "soft" or flexible rather than being "hard" or rigidly consistent at any given moment.

Eventual consistency: The system will eventually become consistent, given enough time. This principle assumes that if no new updates are made to a given data item, all copies of that item will converge to the same value over time

- Unlike RDBMSs that focus on consistency, BASE systems focus on availability
- BASE systems are noteworthy because their number-one objective is to allow new data to be stored, even at the risk of being out of sync for a short period of time.
- They relax the rules and allow reports to run even if not all portions of the database are synchronized.
- BASE systems aren't considered *pessimistic* in that they don't fret about the details if one process is behind. They're *optimistic* in that they assume that eventually all systems will catch up and become consistent.

BASE systems are ideal for web storefronts, where filling a shopping cart and placing an order is the main priority.

Q. You are designing a real-time social media application that supports millions of users posting updates, commenting, and liking posts across different regions. Given the high volume of interactions, some actions, like updating the like counts on posts, may not be reflected immediately across all nodes. However, ensuring that users can post updates and interact with others without delays is a top priority.

- Explain how BASE properties can support this social media application in managing high traffic while maintaining user engagement.

- What are the potential trade-offs of this approach?

Answer: BASE Properties in the Social Media Application

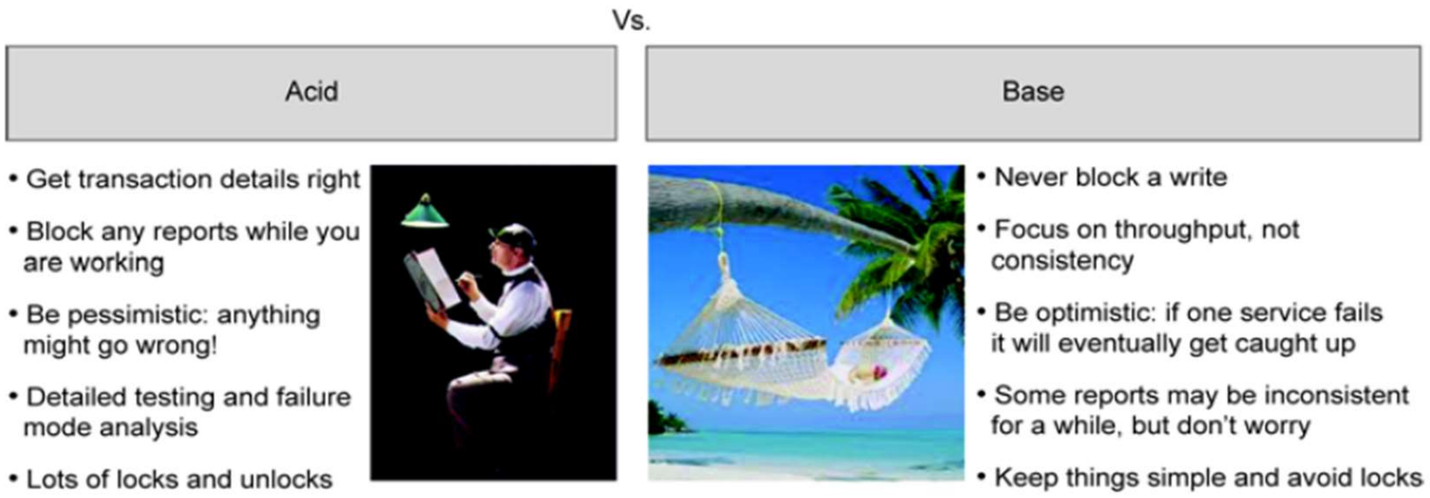
Basically Available:
The application must ensure that users can always post updates and interact with content, even if some background processes (like updating like counts or comment threads) are still being processed. By prioritizing availability, the system allows users to submit posts and likes without facing downtime or delays in accessing the application.
Soft State:
The system can accept that some user actions (like liking a post) may not immediately reflect the updated state across all nodes or servers. For instance, a user may like a post, but the like count may not update instantly for all users viewing that post. This temporary inconsistency is acceptable since the application is designed to handle updates gradually, allowing for smoother user interactions.
Eventually Consistent:
After a user interacts with a post (e.g., likes or comments), the system will eventually propagate these updates throughout the network. This means that over time, all nodes will synchronize and reflect the correct counts and interactions. For instance, while one user might see an old like count, the system will ensure that all counts are consistent after a short period, providing a seamless experience.

Trade-offs Involved

Data Accuracy:
Users may see outdated like counts or comment threads. For instance, if a user likes a post, they might see the count as unchanged for a brief period, which can be confusing.

User Experience:
While prioritizing availability is essential, it can lead to situations where users experience delays in feedback. If a user likes a post and does not see the updated count immediately, they might be uncertain whether the action was successful.

Complexity of Data Management:
Implementing BASE properties requires additional logic to manage temporary states and eventual consistency. Developers need to create robust mechanisms to handle updates and ensure that eventual consistency is achieved without burdening users.



- Get transaction details right
- Block any reports while you are working
- Be pessimistic: anything might go wrong!
- Detailed testing and failure mode analysis
- Lots of locks and unlocks

- Never block a write
- Focus on throughput, not consistency
- Be optimistic: if one service fails it will eventually get caught up
- Some reports may be inconsistent for a while, but don't worry
- Keep things simple and avoid locks

Figure 2.8 ACID versus BASE—understanding the trade-offs. This figure compares the rigid financial accounting rules of traditional RDBMS ACID transactions with the more laid-back BASE approach used in NoSQL systems. RDBMS ACID systems are ideal when all reports must always be consistent and reliable. NoSQL BASE systems are preferred when priority is given to never blocking a write transaction. Your business requirements will determine whether traditional RDBMS or NoSQL systems are right for your application.

Understanding trade-offs with Brewer’s CAP theorem

Used to make critical decisions while designing a distributed system.

Eric Brewer first introduced the CAP theorem in 2000. The CAP theorem states that any distributed database system can have at most two of the following three desirable properties:

Consistency
Availability
Partition tolerance

Understanding trade-offs with Brewer's CAP theorem

Eric Brewer first introduced the CAP theorem in 2000. The CAP theorem states that any distributed database system can have at most two of the following three desirable properties:

Consistency
Availability
Partition tolerance

Consistency—Every read receives the most recent write or an error.

Having a single, up-to-date, readable version of your data available to all clients.
This isn't the same as the consistency we talked about in ACID. Consistency here is concerned with multiple clients reading the same items from replicated partitions and getting consistent results.
Some locking mechanism is used until the data is synced across nodes.
Ex: online trading platform

Understanding trade-offs with Brewer's CAP theorem

Eric Brewer first introduced the CAP theorem in 2000. The CAP theorem states that any distributed database system can have at most two of the following three desirable properties:

Consistency
Availability
Partition tolerance

High availability Every request receives a non-error message without the guarantee of containing the most recent write.

Knowing that the distributed database will always allow database clients to update items without delay.

Internal communication failures between replicated data shouldn't prevent updates.

Understanding trade-offs with Brewer’s CAP theorem

Eric Brewer first introduced the CAP theorem in 2000. The CAP theorem states that any distributed database system can have at most two of the following three desirable properties:

Consistency
Availability
Partition tolerance

Partition tolerance The ability of the system to keep responding to client requests even if there’s a communication failure between database partitions.

This is analogous to a person still having an intelligent conversation even after a link between parts of their brain isn’t working.

Which is almost always required.

The CAP theorem helps you understand that once you partition your data, you must consider the availability-consistency spectrum in a network failure situation.

Then the CAP theorem allows you to determine which options best match your business requirements.

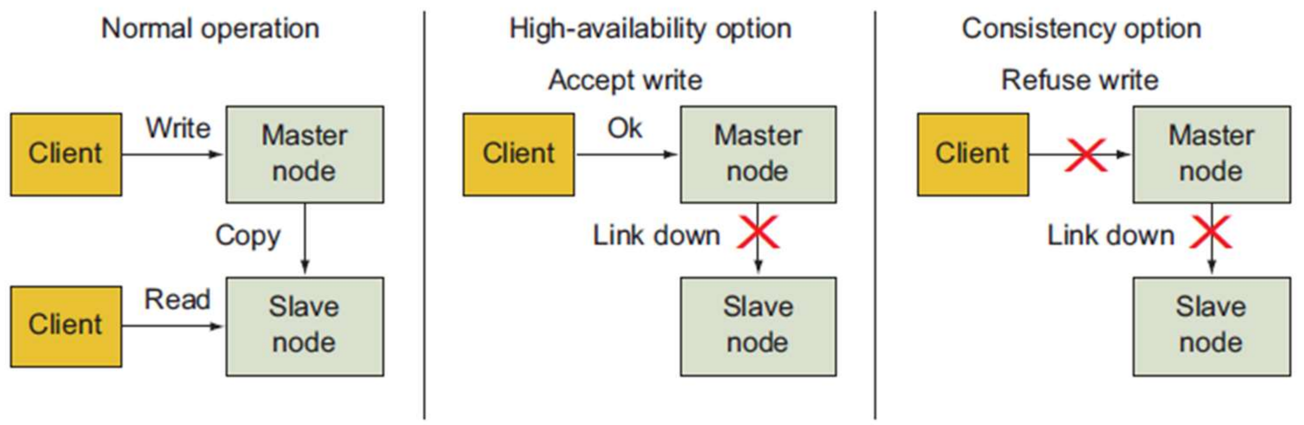


Figure 2.10 The partition decision. The CAP theorem helps you decide the relative merits of availability versus consistency when a network fails. In the left panel, under normal operation a client write will go to a master and then be replicated over the network to a slave. If the link is down, the client API can decide the relative merits of high availability or consistency. In the middle panel, you accept a write and risk inconsistent reads from the slave. In the right panel, you choose consistency and block the client write until the link between the data centers is restored.

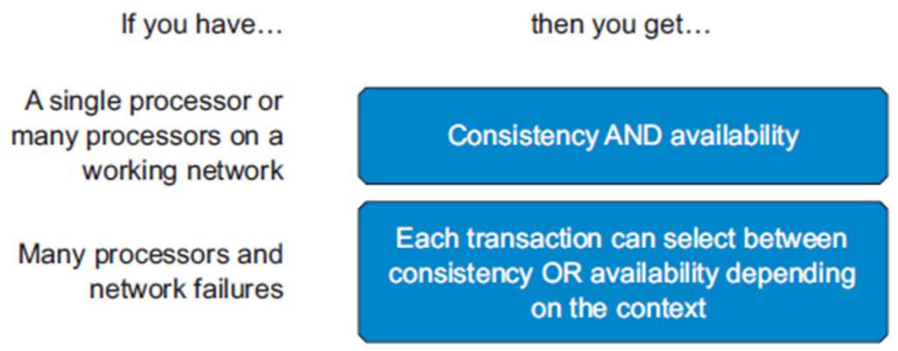

If you have…
then you get…
A single processor or many processors on a working network
Consistency AND availability
Many processors and network failures
Each transaction can select between consistency OR availability depending on the context

Figure 2.11 **The CAP theorem shows that you can have both consistency and availability if you're only using a single processor. If you're using many processors, you can chose between consistency and availability depending on the transaction type, user, estimated downtime, or other factors.**

Tools like the CAP theorem can help guide database selection discussions within an organization and prioritize what properties (consistency, availability, and scalability) are most important.

If high consistency and update availability are simultaneously required, then a faster single processor might be your best choice.

If you need the scale-out benefits that distributed systems offer, then you can make decisions about your need for update availability versus read consistency for each transaction type.

Q. You are tasked with developing a healthcare management system that enables patients to schedule appointments, access medical records, and communicate with healthcare providers. The system is distributed across multiple regions to ensure high availability and low latency. However, during peak usage times or unexpected outages, network partitions between data centers can occur. Considering the CAP theorem (Consistency, Availability, and Partition Tolerance), how would you make design decisions regarding the system's behavior during network partition scenarios?

a) Which two properties (from Consistency, Availability, and Partition Tolerance) would you prioritize during network partition scenarios, and why?

b) What are the trade-offs of prioritizing these two properties in the context of a healthcare management system?

Answer: Prioritized Properties during Network Partition Scenarios

Prioritized Properties: **Availability** and **Partition Tolerance**

Availability: In a healthcare system, it is crucial that patients can access their medical records, schedule appointments, and communicate with healthcare providers without experiencing downtime. Ensuring that the system remains available, even during network partitions, helps maintain continuous access to critical health information and services.

Partition Tolerance: Given the distributed nature of the system across multiple data centers, the ability to continue operating despite network partitions is essential. The system should be designed to function effectively, even when some parts of the network are unable to communicate.

b) Trade-offs of Prioritizing Availability and Partition Tolerance

Eventual Consistency:

By prioritizing availability, the system may allow temporary inconsistencies in data. For instance, if a patient updates their medical records, that update might not immediately reflect in all nodes. This could lead to healthcare providers accessing outdated information, potentially affecting patient care.

Risk of Conflicting Information:

With an AP system, there is a risk that different data centers might hold conflicting versions of data during a partition. For example, if two healthcare providers are accessing patient records simultaneously, one might see an updated medication list while the other does not, leading to potential confusion or miscommunication.

Consistency is deprioritized in favor of **Availability** and **Partition Tolerance** (AP) during network partitions because:

Immediate Access to Healthcare Services: In healthcare, accessibility is often critical. Patients need to access their records, schedule appointments, or communicate with providers, sometimes urgently. If strict **Consistency** were prioritized, a network partition would require the system to prevent certain actions (like updating records) until all data centers synchronize, causing delays. This delay could impact patient care, which is often time-sensitive.

Partition Tolerance is Non-Negotiable in Distributed Systems. Since the system spans multiple regions to ensure low latency and broad accessibility, it has to be **Partition Tolerant** by design to handle inevitable network issues

Eventual Consistency in Non-Critical Scenarios: Many healthcare operations can accept a short delay in data consistency during a partition if it means maintaining service availability.

You are tasked with developing an online banking system that enables customers to view account balances, make transfers, and manage transactions. The system is distributed across multiple regions to ensure high availability and low latency. However, during peak usage times or unexpected outages, network partitions between data centers can occur. Considering the CAP theorem (Consistency, Availability, and Partition Tolerance), how would you make design decisions regarding the system's behavior during network partition scenarios?

Which two properties (from Consistency, Availability, and Partition Tolerance) would you prioritize during network partition scenarios, and why?

What are the trade-offs of prioritizing these two properties in the context of a banking system?

You are tasked with developing an online banking system that enables customers to view account balances, make transfers, and manage transactions. The system is distributed across multiple regions to ensure high availability and low latency. However, during peak usage times or unexpected outages, network partitions between data centers can occur. Considering the CAP theorem (Consistency, Availability, and Partition Tolerance), how would you make design decisions regarding the system's behavior during network partition scenarios?

Which two properties (from Consistency, Availability, and Partition Tolerance) would you prioritize during network partition scenarios, and why?

What are the trade-offs of prioritizing these two properties in the context of a banking system?

Q. You are designing a global online multiplayer gaming platform where players from around the world can connect and play real-time games. The platform has servers located in multiple regions to provide low-latency gaming experiences and support millions of concurrent users.
During peak gaming hours or unexpected server issues, network partitions may occur between data centers, causing some regions to temporarily lose communication with others. However, the game needs to remain accessible for users in all regions, and players expect continuous, uninterrupted gameplay, even if they are temporarily isolated from other players or certain game data.

a) Which two properties (from Consistency, Availability, and Partition Tolerance) would you prioritize during network partition scenarios?

b) What are the trade-offs of prioritizing these two properties in the context of a global online gaming platform?

Answer is availability and partition tolerance

In the context of the **global online multiplayer gaming platform**, **Consistency** refers to the property where all players, across different regions or data centers, see the **same game state** at any given point in time. This means that when an action is performed by one player, such as making a move or scoring a point, that action should be reflected consistently across all players’ views of the game