DOS 9.0

↓

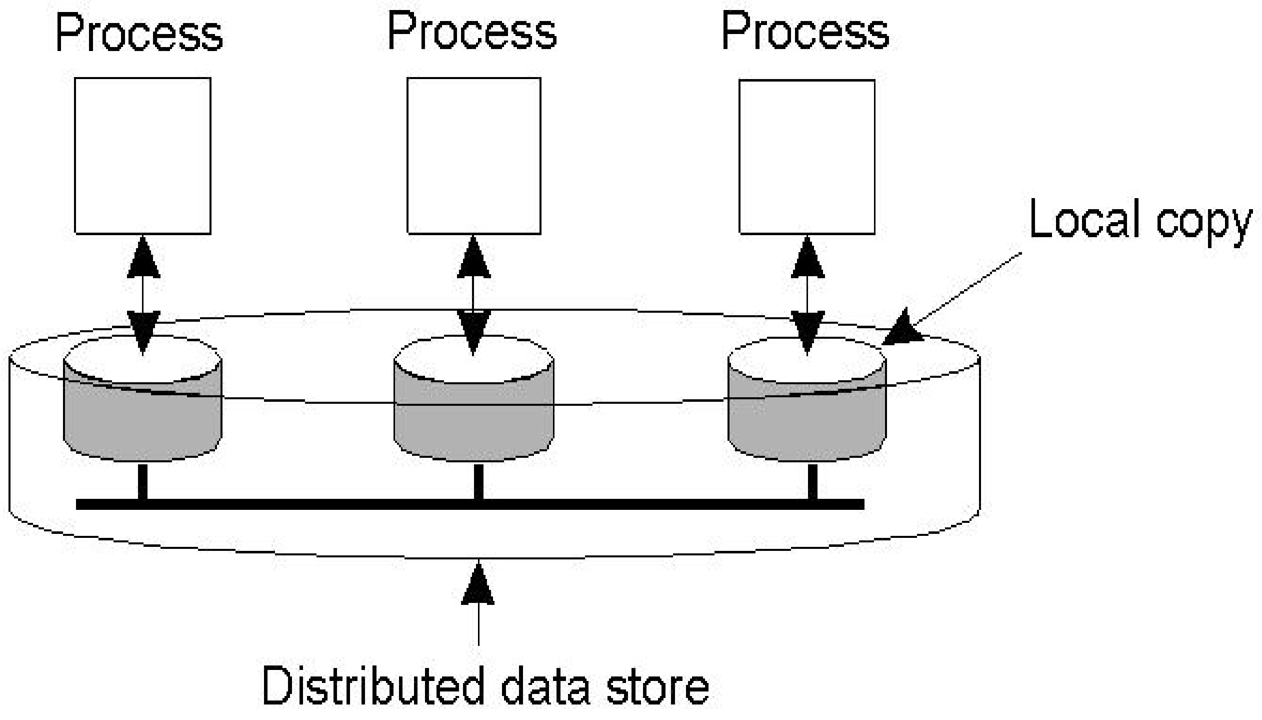
Data-Centric Consistency Models

- A data-store can be read from or written to by any process in a distributed system.

- A local copy of the data-store (replica) can support "fast reads".

- However, a write to a local replica needs to be propagated to all remote replicas.

- Various consistency models help to understand the various mechanisms used to achieve and enable this.

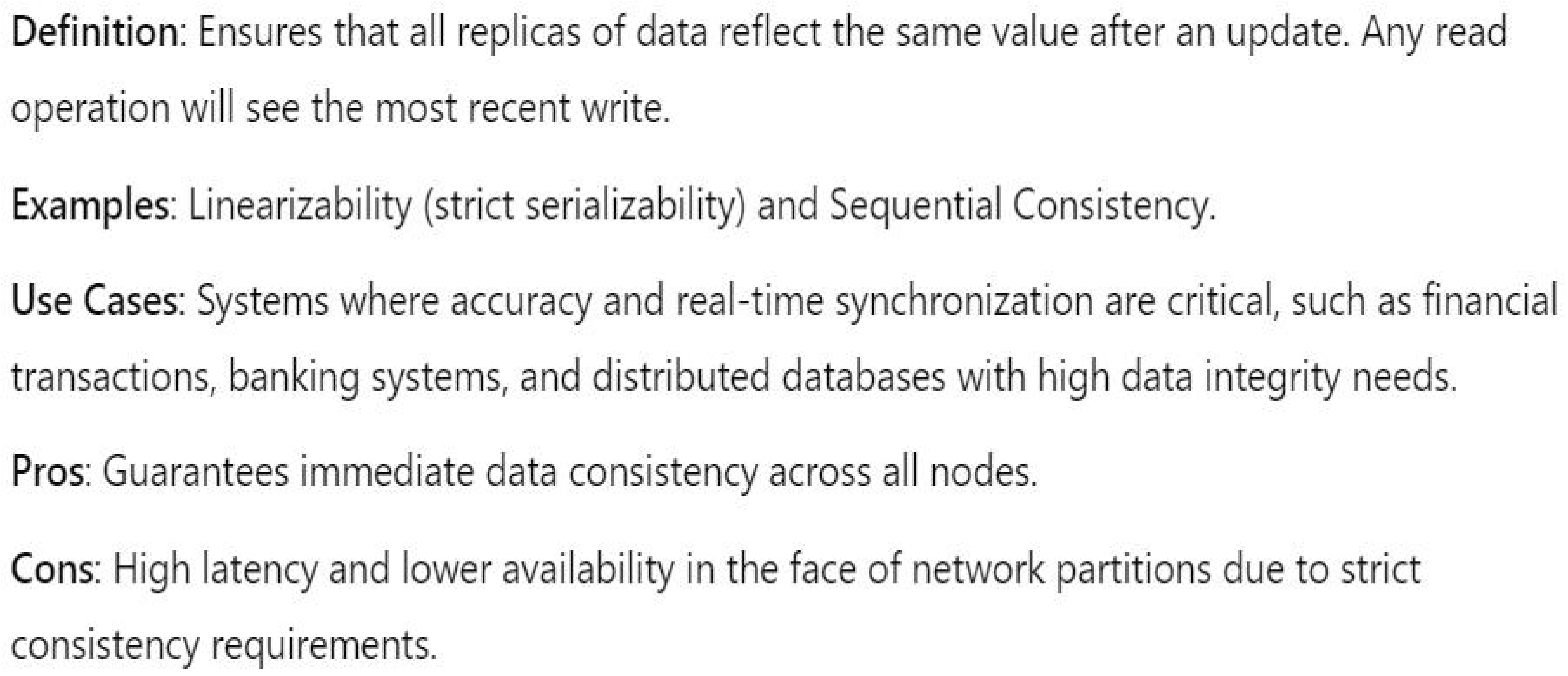


- A “consistency model” is a CONTRACT between a DS data-store and its processes.

- If the processes agree to the rules, the data-store will perform properly and as advertised.

- Types of Consistency Models are:
  - Strong Consistency
  - Causal Consistency
  - Eventual Consistency
  - Session Consistency

Strong Consistency:::



Definition: Ensures that all replicas of data reflect the same value after an update. Any read operation will see the most recent write.

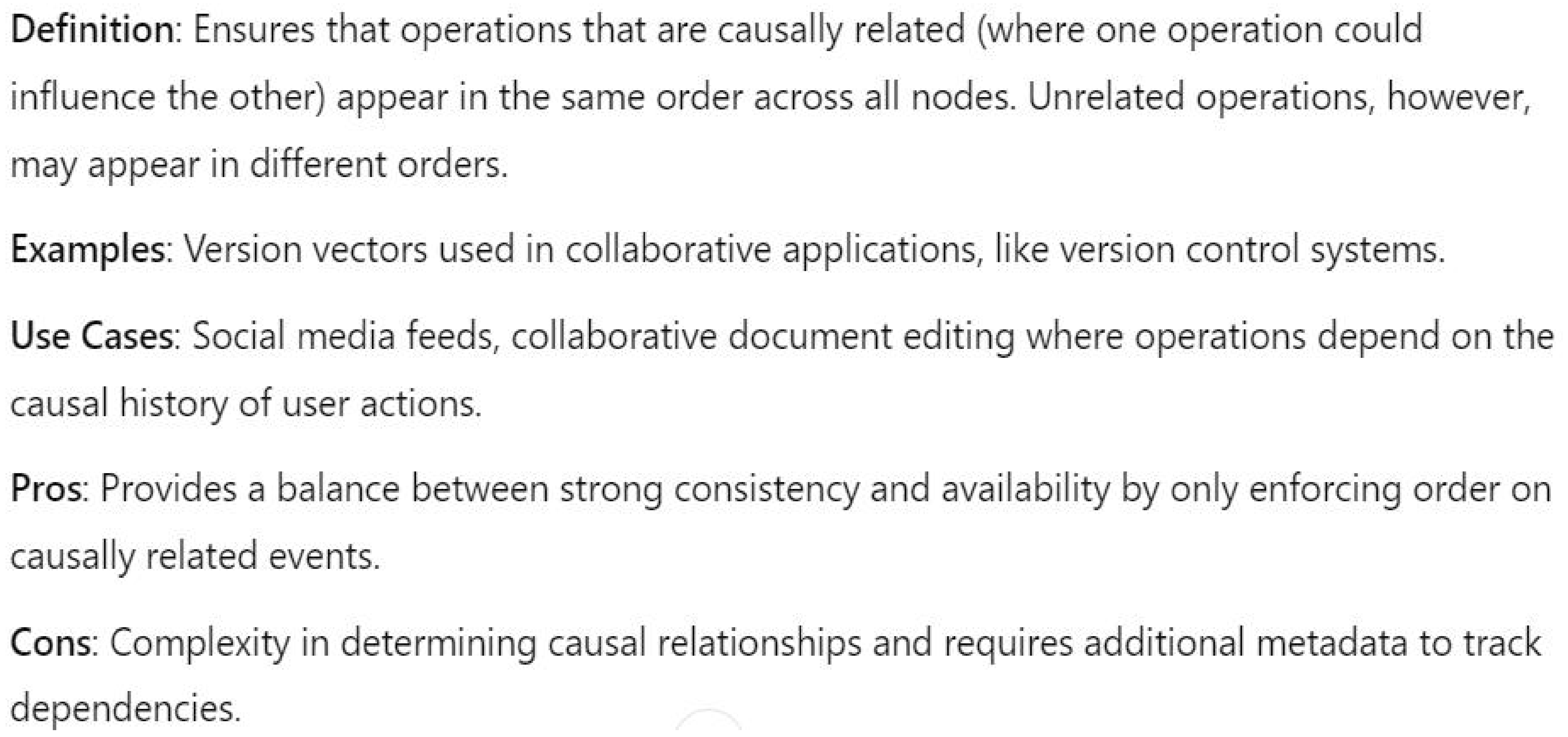
Examples: Linearizability (strict serializability) and Sequential Consistency.

Use Cases: Systems where accuracy and real-time synchronization are critical, such as financial transactions, banking systems, and distributed databases with high data integrity needs.

Pros: Guarantees immediate data consistency across all nodes.

Cons: High latency and lower availability in the face of network partitions due to strict consistency requirements.

Causal Consistency:::



Definition: Ensures that operations that are causally related (where one operation could influence the other) appear in the same order across all nodes. Unrelated operations, however, may appear in different orders.

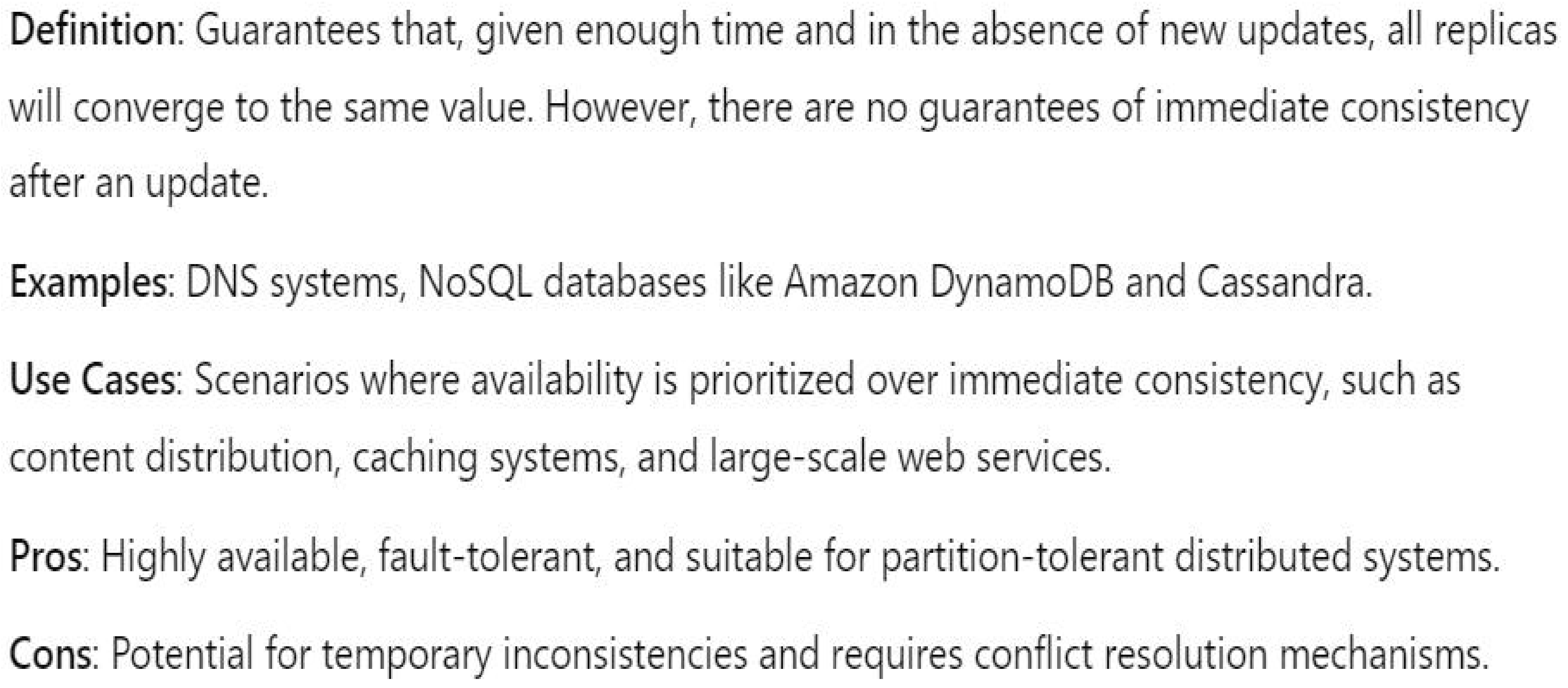
Examples: Version vectors used in collaborative applications, like version control systems.

Use Cases: Social media feeds, collaborative document editing where operations depend on the causal history of user actions.

Pros: Provides a balance between strong consistency and availability by only enforcing order on causally related events.

Cons: Complexity in determining causal relationships and requires additional metadata to track dependencies.

Eventual Consistency:::



Definition: Guarantees that, given enough time and in the absence of new updates, all replicas will converge to the same value. However, there are no guarantees of immediate consistency after an update.

Examples: DNS systems, NoSQL databases like Amazon DynamoDB and Cassandra.

Use Cases: Scenarios where availability is prioritized over immediate consistency, such as content distribution, caching systems, and large-scale web services.

Pros: Highly available, fault-tolerant, and suitable for partition-tolerant distributed systems.

Cons: Potential for temporary inconsistencies and requires conflict resolution mechanisms.

Session Consistency:::

Definition: Guarantees a consistent view of data within a session, meaning a user will see their updates and operations as consistent throughout the duration of a session.

Examples: Common in user-centric applications where users expect their session data to be coherent.

Use Cases: E-commerce carts, personalized recommendations, or applications where users maintain temporary session data.

Pros: Delivers consistency at the session level without requiring system-wide strong consistency.

Cons: Consistency is only maintained within a single session, so data may appear inconsistent across different sessions.

Non-Uniform Memory Access (NUMA)

- NUMA is a computer memory design used in multiprocessor systems where memory access times vary depending on the memory location relative to a processor.
- In NUMA systems, processors are grouped into nodes, each with its own local memory.
- Processors can access memory in other nodes, but access to remote memory is slower than to local memory.
- This architecture is especially common in multi-socket systems and is designed to improve scalability and performance in parallel processing.

NUMA vs. Uniform Memory Access (UMA):

- UMA: All processors have equal access time to all memory locations, common in small-scale systems.
- NUMA: Access time depends on memory location relative to the processor, making it suitable for large, multi-processor systems.

Key Characteristics of NUMA:

1. **Memory Access Locality**: Each processor has faster access to its local memory (located within the same node) than to memory in other nodes.
2. **Node Structure**: A NUMA system typically divides memory and processors into nodes. Each node has one or more processors and dedicated memory.
3. **Interconnects**: NUMA nodes are connected via a high-speed interconnect, enabling processors to access remote memory at the cost of higher latency.
4. **Optimized for Multithreading**: NUMA systems are optimized for workloads with high levels of parallelism, making them suitable for applications that benefit from concurrent data processing such as databases and scientific computing.

Applications of NUMA:

NUMA architecture is widely used in high-performance computing, enterprise databases, and systems requiring large, multi-threaded workloads, where memory access speed is crucial for maintaining efficiency.

NUMA-aware systems are especially useful for applications like SQL databases, in-memory analytics, and complex simulations.

DASH (Directory Architecture for Shared Memory)

- DASH cache coherence protocol is a protocol developed for distributed shared memory (DSM) systems.
- It was introduced by researchers at Stanford University to address cache coherence challenges in multiprocessor systems that use a non-uniform memory access (NUMA) architecture.
- The DASH protocol aims to maintain cache coherence across distributed caches in a scalable and efficient manner.

Components of the DASH::::

- **Directory-Based Coherence:** **DASH uses a directory-based approach to manage coherence. Each memory block has an associated directory entry, which keeps track of the caches that hold copies of that memory block.**
- **Distributed Directory:** **In DASH, the directory is not centralized; instead, each memory node maintains a directory for the memory blocks it owns. This distributed directory architecture helps in achieving scalability by reducing bottlenecks associated with a single, centralized directory.**
- **Cache States:** **DASH uses several cache states to manage coherence:**
  - **Shared:** **Multiple nodes hold a copy of the block.**
  - **Modified:** **A single node has an exclusive, modified copy of the block, which is not up-to-date in main memory.**
  - **Invalid:** **The cache does not hold a valid copy of the block.**
- **Home Node and Remote Nodes:**
  - **Each memory block has a designated home node that keeps the directory entry for that block.**
  - **Remote nodes are nodes that request access to the memory block from the home node.**
- **Message Passing for Coherence:** **DASH relies on message passing between nodes to maintain coherence. For instance, when a node wants to modify a memory block, it sends a message to the home node, which coordinates with other nodes to invalidate any copies that other caches might hold.**

The End