Lecture 9

Big Data Storage Concepts and Strategies

Dr. Lydia Wahid

Agenda

- Sharding
- Replication:
  - Master-Slave
  - Peer-to-Peer
- Sharding and Replication:
  - Sharding and master-slave replication
  - Sharding and peer-to-peer replication
- CAP theorem and BASE

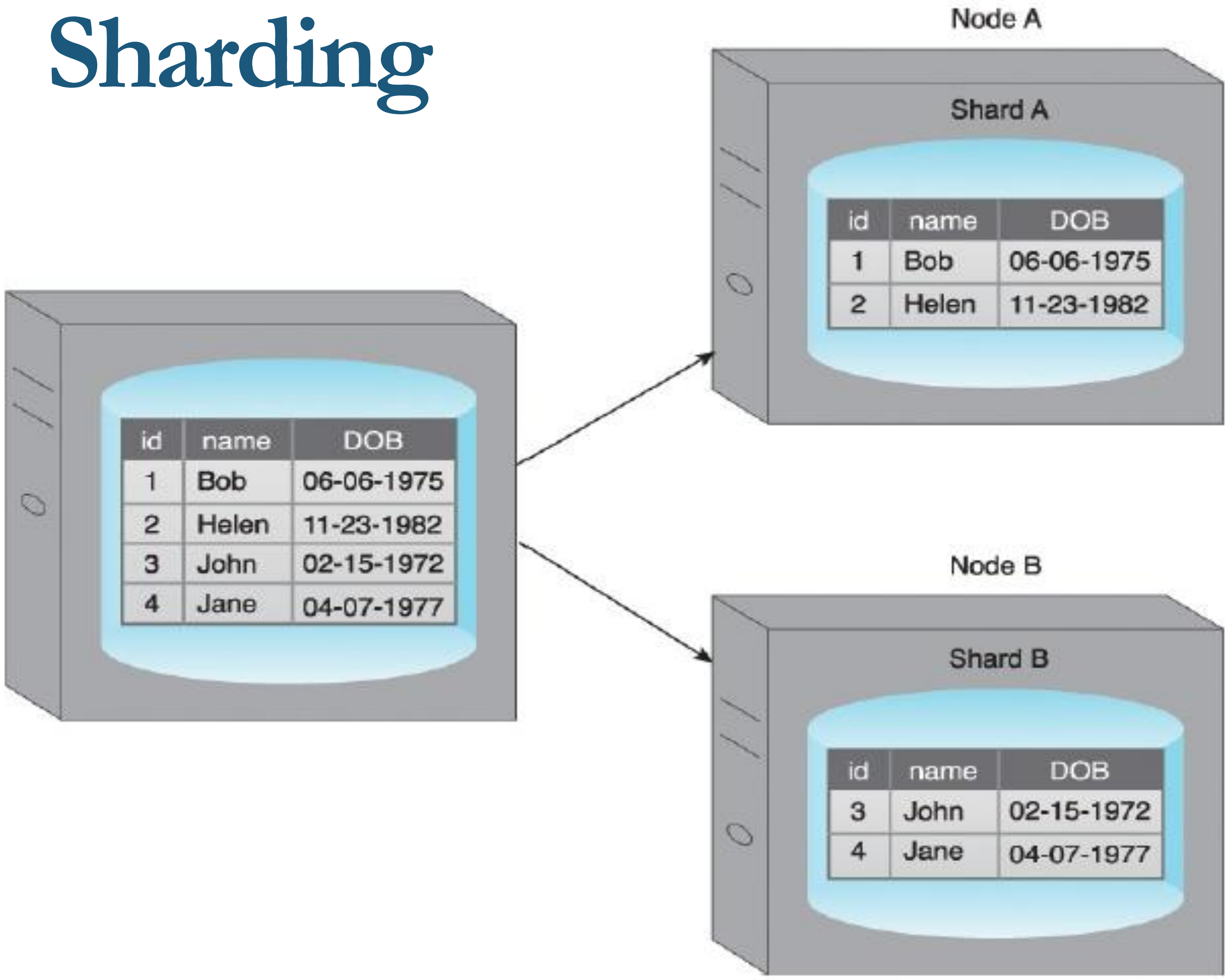
Sharding

Sharding

- Sharding is the process of **horizontally** partitioning a large dataset into a collection of smaller, more manageable datasets called **shards**.
- The shards are **distributed** across **multiple** nodes, where a node is a server or a machine.
- Each shard is stored on a separate node and each node is responsible for only the data stored on it.
- Each shard shares the **same schema**, and all shards collectively represent the complete dataset.

Sharding

- ➢An example of sharding:



id	name	DOB
1	Bob	06-06-1975
2	Helen	11-23-1982
3	John	02-15-1972
4	Jane	04-07-1977

Node A

Shard A

id	name	DOB
1	Bob	06-06-1975
2	Helen	11-23-1982

Node B

Shard B

id	name	DOB
3	John	02-15-1972
4	Jane	04-07-1977

Sharding

- Sharding allows the distribution of processing loads across multiple nodes to achieve **horizontal scalability**.

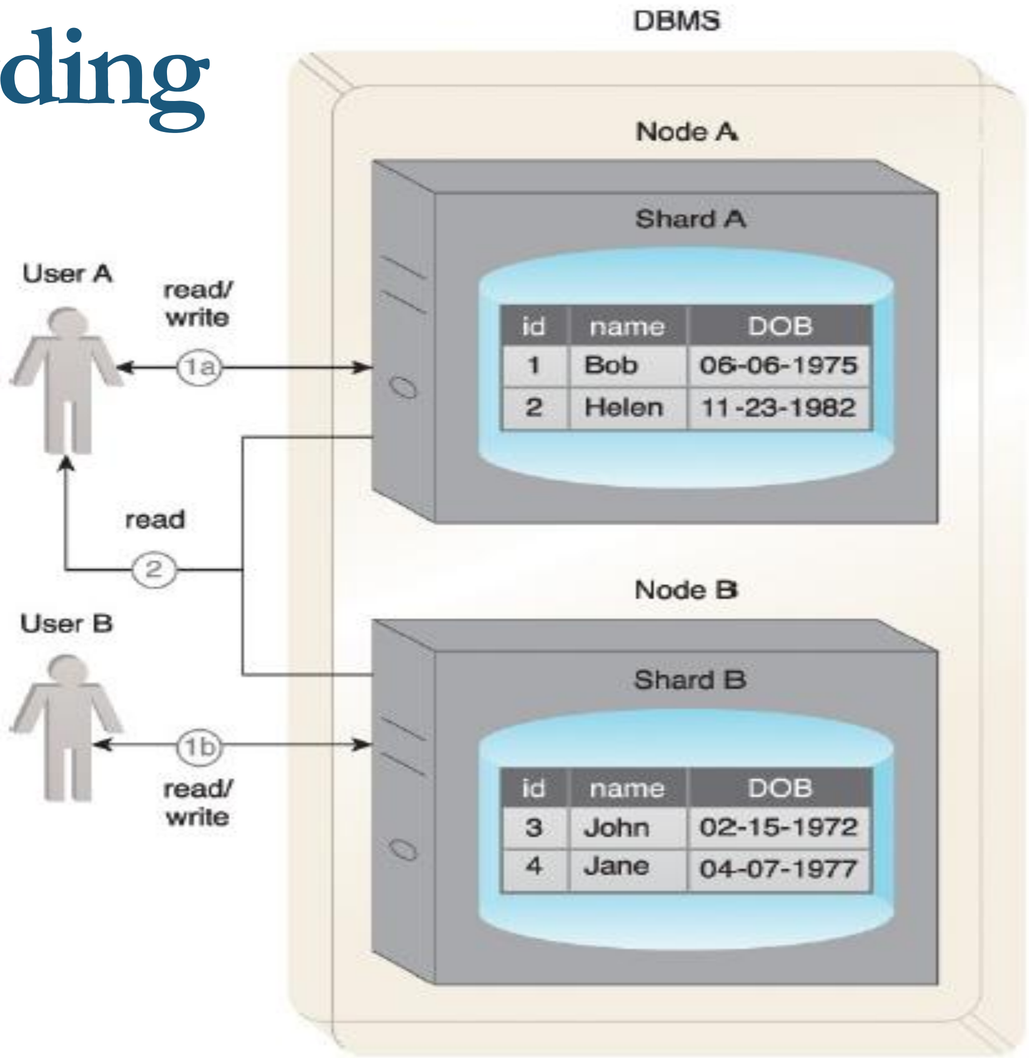
- Horizontal scaling is a method for **increasing a system's capacity** by **adding similar or higher capacity** resources alongside existing resources.

- Since each node is responsible for only a part of the whole dataset, **read/write times are greatly improved**.

Sharding

- Each shard can **independently** service reads and writes for the specific subset of data that it is responsible for.

- Depending on the query, data may need to be fetched from **both shards**.



id	name	DOB
1	Bob	06-06-1975
2	Helen	11-23-1982

id	name	DOB
3	John	02-15-1972
4	Jane	04-07-1977

Sharding

- A benefit of sharding is that it provides **partial tolerance** toward failures.
- In case of a node failure, only data stored on that node is affected.
- With regards to **data partitioning**, **query patterns** need to be taken into account so that shards themselves do not become **performance bottlenecks**.
- For example, queries requiring data from multiple shards will impose **performance penalties**.
- **Data locality** keeps commonly accessed data co-located on a *single* shard and helps counter such performance issues.

Replication

Replication

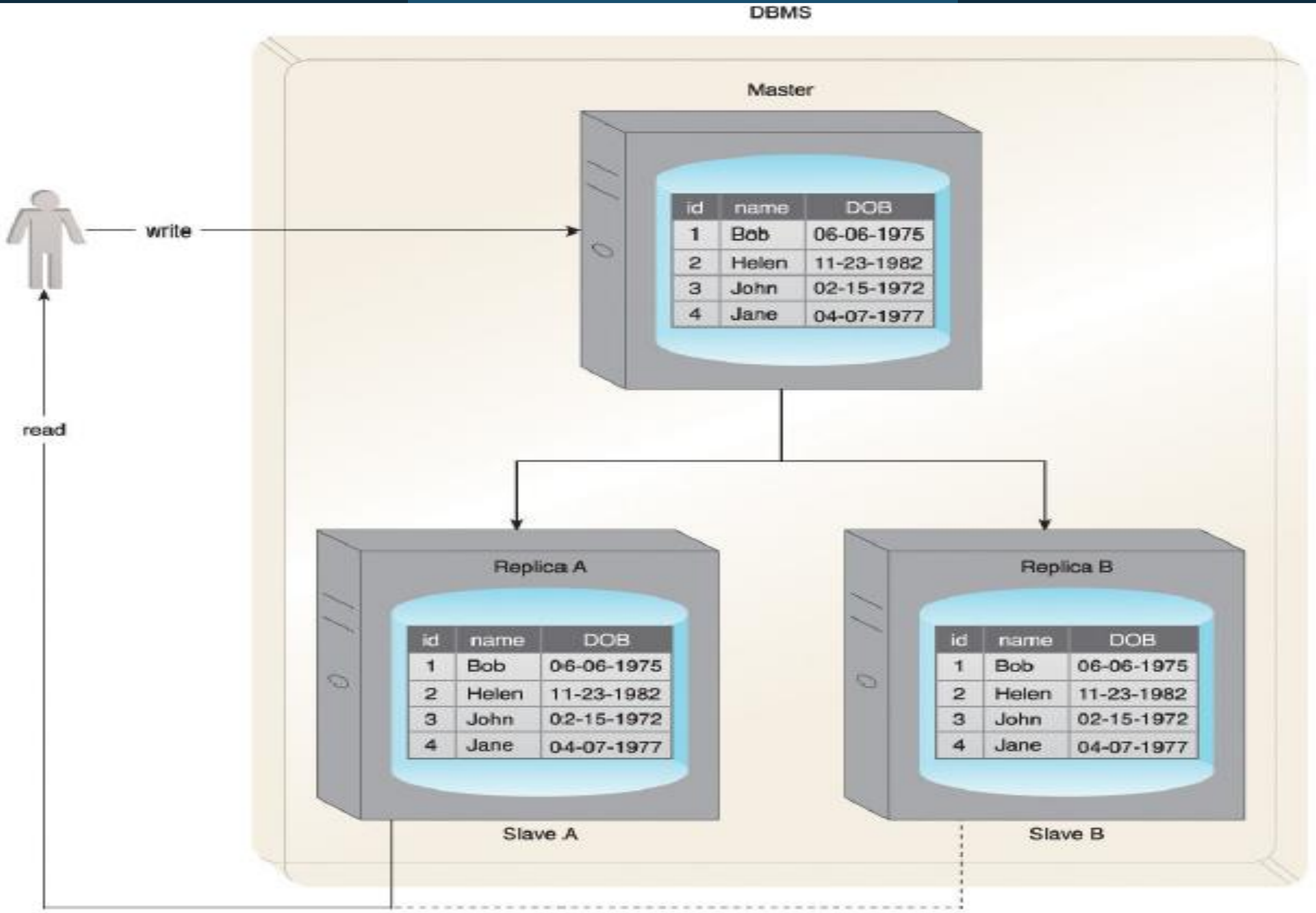
- Replication stores **multiple copies** of a dataset, known as **replicas**, on multiple nodes.
- Replication provides **scalability** and **availability** due to the fact that the same data is replicated on various nodes.
- **Fault tolerance** is also achieved since **data redundancy** ensures that data is not lost when an individual node fails.
- There are two different methods that are used to implement replication:
  - Master-Slave
  - Peer-to-Peer

Replication: **Master-Slave**

- During master-slave replication, nodes are arranged in a master-slave configuration, and **all data** is written to a **master node**.

- Once saved, the data is **replicated** over to **multiple slave nodes**.

- All external **write** requests, including insert, update and delete, occur on the **master node**, whereas **read** requests can be fulfilled by any **slave** node.



Master

id	name	DOB
1	Bob	06-06-1975
2	Helen	11-23-1982
3	John	02-15-1972
4	Jane	04-07-1977

Replica A

id	name	DOB
1	Bob	06-06-1975
2	Helen	11-23-1982
3	John	02-15-1972
4	Jane	04-07-1977

Replica B

id	name	DOB
1	Bob	06-06-1975
2	Helen	11-23-1982
3	John	02-15-1972
4	Jane	04-07-1977

Master-Slave replication example

Replication: Master-Slave

- Master-slave replication is ideal for **read intensive loads** rather than write intensive loads since growing read demands can be managed by horizontal scaling to add more slave nodes.

- **Writes are consistent**, as all writes are coordinated by the master node.

- The implication is that **write performance** will *suffer* as the amount of **writes increases**.

Replication: Master-Slave

- If the **master node fails**, **reads are still possible** via any of the slave nodes.

- A slave node can be configured as a **backup node** for the master node.

- In the event that the master node fails, writes are **not supported** until a **master node is reestablished**.

- The master node is either **revived** from a backup of the master node, or a **new master node** is chosen from the slave nodes.

Replication: Master-Slave

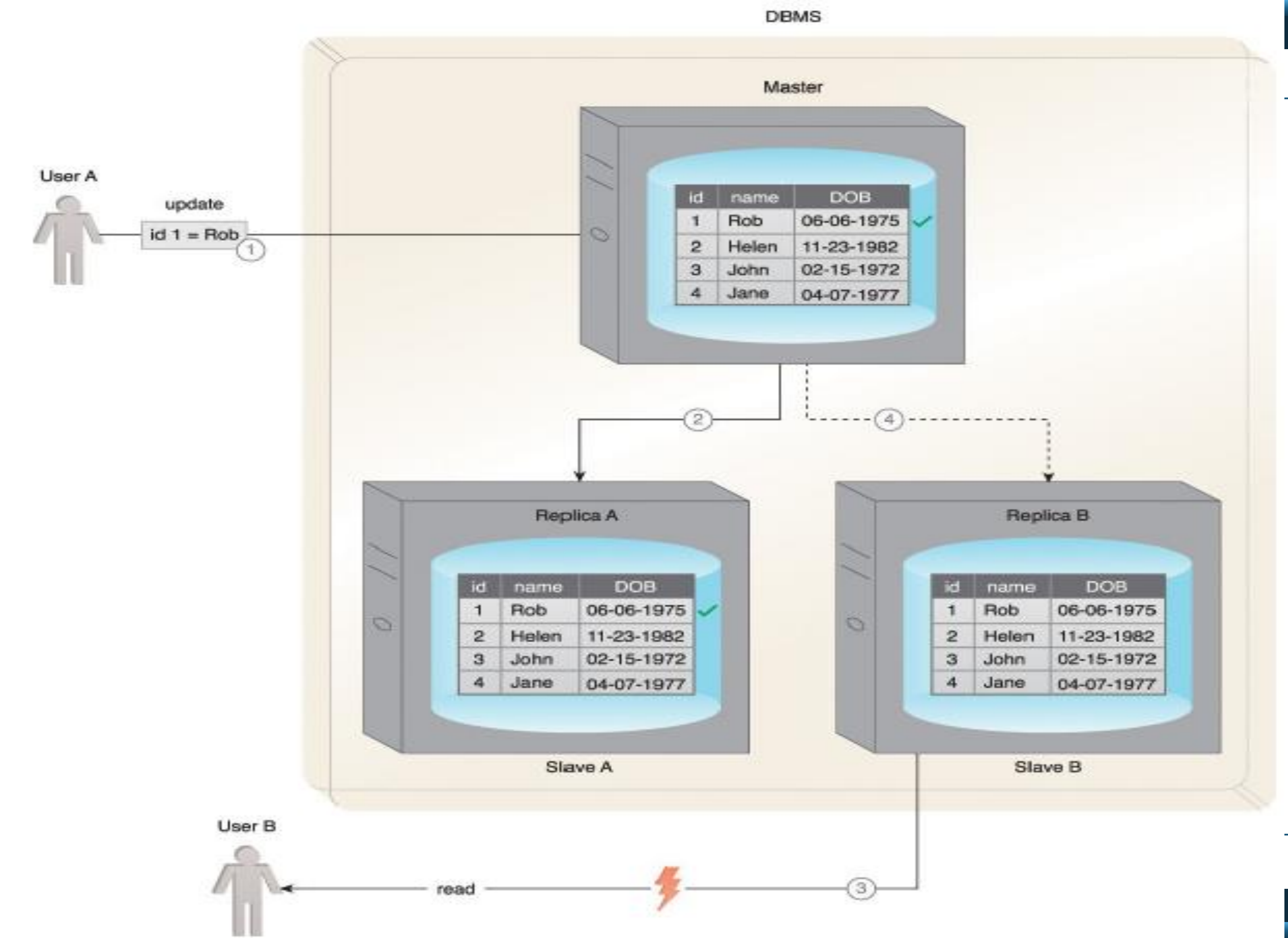
- One concern with master-slave replication is **read inconsistency**, which can be an issue if a slave node is read prior to an update to the master being copied to it.

- To ensure read consistency, a **voting system** can be implemented where a read is declared consistent if the **majority** of the slaves contain the **same** version of the record.

- Implementation of such a voting system requires a *reliable* and *fast* communication mechanism between the slaves.

Replication: **Master-Slave**

- An example scenario of a read inconsistency is illustrated as follows:
  1. **User A updates** data.
  2. The data is **copied** over to **Slave A** by the Master.
  3. **Before** the data is **copied** over to **Slave B**, **User B** tries to **read** the data from **Slave B**, which results in an inconsistent **read**.
  4. The data will eventually become consistent when **Slave B** is **updated** by the Master.



Master

id	name	DOB	
1	Rob	06-06-1975	✓
2	Helen	11-23-1982	
3	John	02-15-1972	
4	Jane	04-07-1977	

Replica A

id	name	DOB	
1	Rob	06-06-1975	✓
2	Helen	11-23-1982	
3	John	02-15-1972	
4	Jane	04-07-1977	

Replica B

id	name	DOB
1	Rob	06-06-1975
2	Helen	11-23-1982
3	John	02-15-1972
4	Jane	04-07-1977

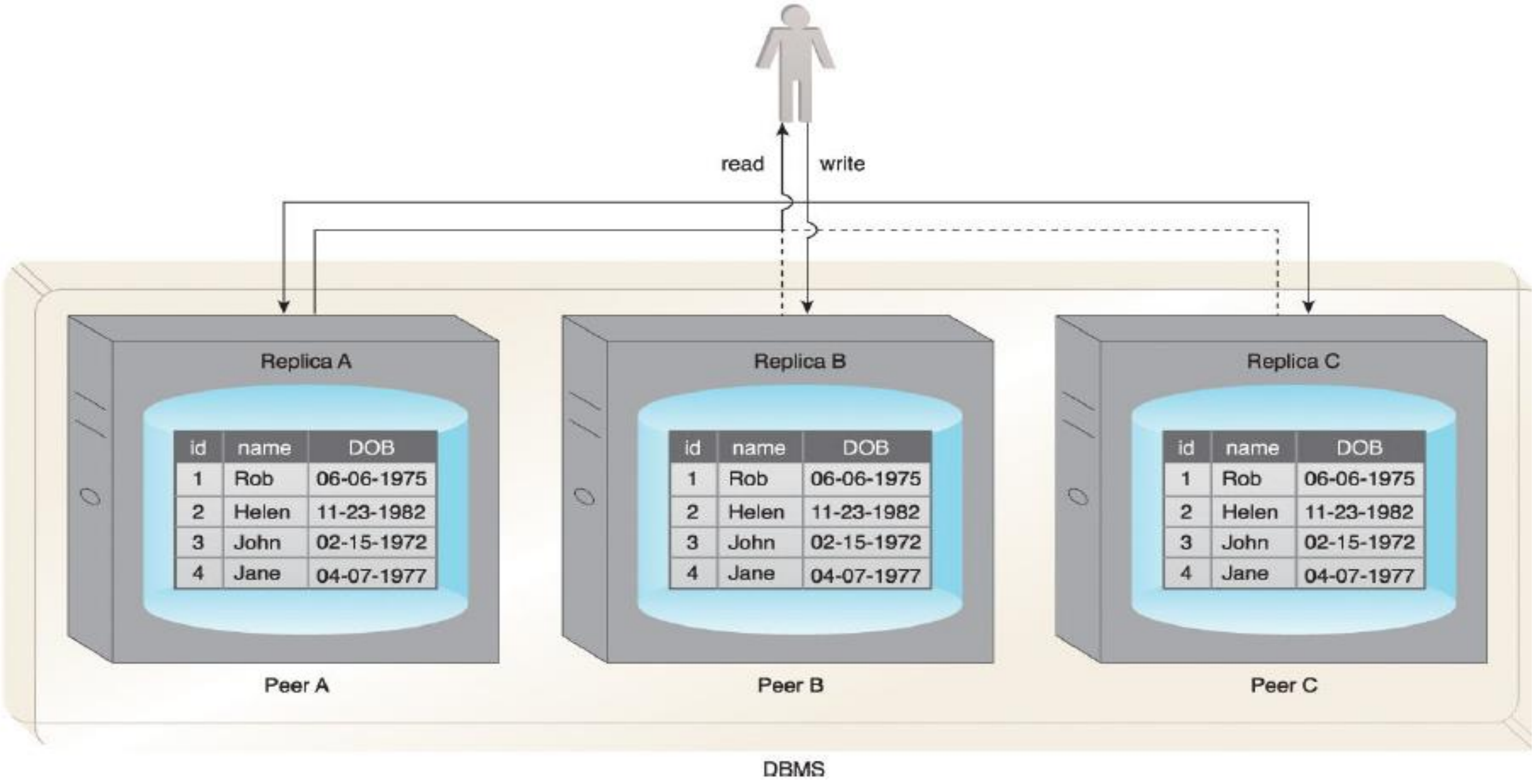
Replication: **Peer-to-Peer**

- With peer-to-peer replication, **all nodes operate at the same level**.

- In other words, there is not a master-slave relationship between the nodes.

- Each node, known as a **peer**, is **equally capable** of handling **reads** and **writes**.

- Each write is **copied to all peers simultaneously.**



id	name	DOB
1	Rob	06-06-1975
2	Helen	11-23-1982
3	John	02-15-1972
4	Jane	04-07-1977

id	name	DOB
1	Rob	06-06-1975
2	Helen	11-23-1982
3	John	02-15-1972
4	Jane	04-07-1977

id	name	DOB
1	Rob	06-06-1975
2	Helen	11-23-1982
3	John	02-15-1972
4	Jane	04-07-1977

Replication: **Peer-to-Peer**

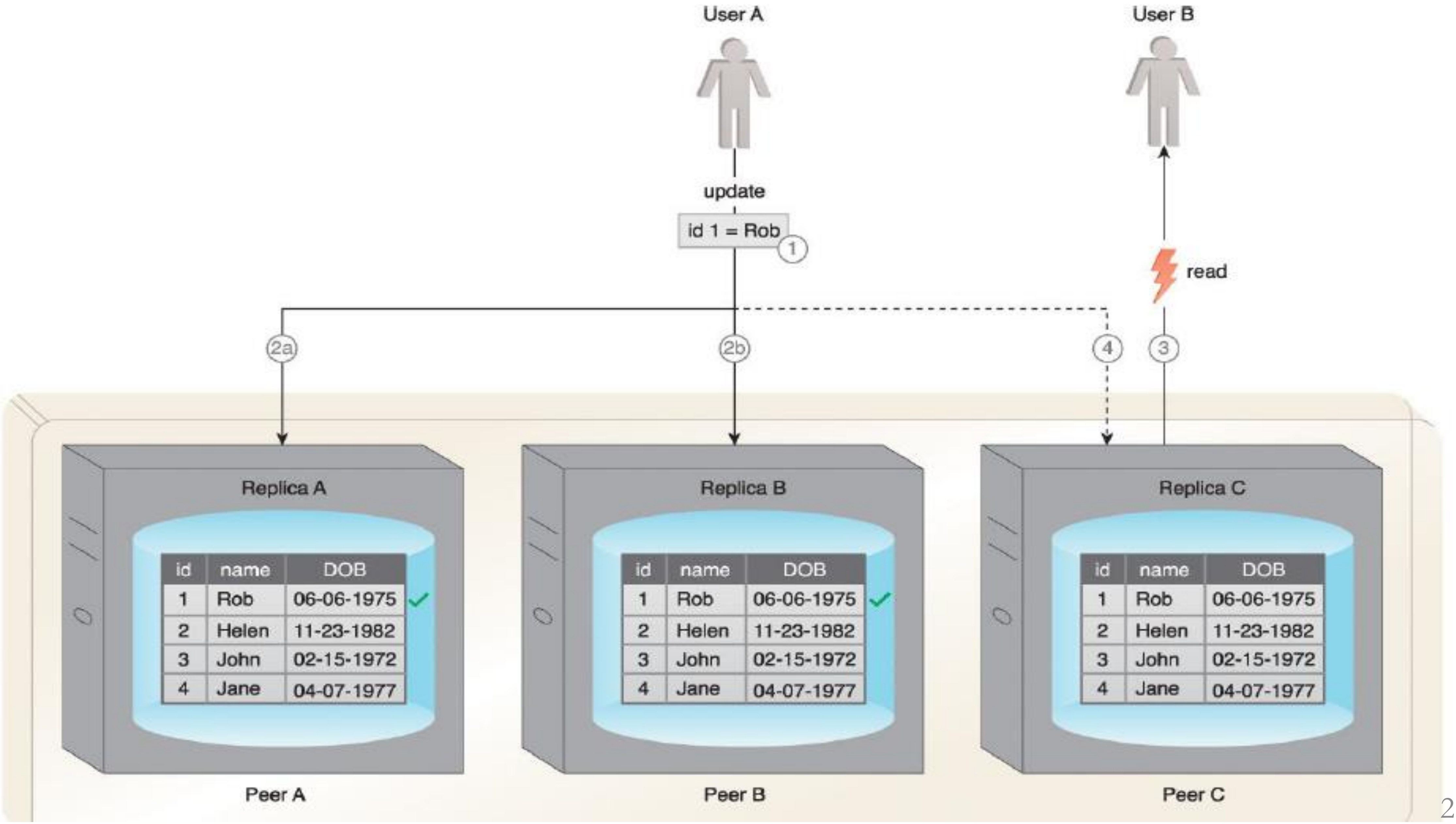
- Peer-to-peer replication is prone to **write inconsistencies** that occur as a result of a simultaneous update of the same data across multiple peers.
- This can be addressed by implementing either a **pessimistic** or **optimistic** concurrency strategy:
  - **Pessimistic concurrency** is a proactive strategy that prevents inconsistency. It uses *locking* to ensure that only one update to a record can occur at a time. However, this is unfavorable to availability since the database record being updated remains *unavailable* until all locks are released.
  - **Optimistic concurrency** is a reactive strategy that does not use locking. Instead, it allows inconsistency to occur with knowledge that eventually consistency will be achieved after all updates have propagated.

Replication: Peer-to-Peer

- With optimistic concurrency, peers may remain **inconsistent** for some period of time before attaining consistency.
- However, the database remains **available** as no locking is involved.
- Like master-slave replication, reads can be **inconsistent** during the time when some of the peers have completed their updates while others perform their updates.
- However, **reads eventually** become **consistent** when the updates have been executed on all peers.

Replication: **Peer-to-Peer**

- An example scenario where an inconsistent read occurs:
  1. **User A updates** data.
  2. a. The data is **copied** over to **Peer A**.
     b. The data is **copied** over to **Peer B**.
  3. Before the data is **copied** over to **Peer C**, **User B** tries to **read** the data from **Peer C**, resulting in an inconsistent read.
  4. The data will eventually be updated on **Peer C**, and the database will once again become consistent.



Replica A

id	name	DOB	
1	Rob	06-06-1975	✓
2	Helen	11-23-1982	
3	John	02-15-1972	
4	Jane	04-07-1977	

Replica B

id	name	DOB	
1	Rob	06-06-1975	✓
2	Helen	11-23-1982	
3	John	02-15-1972	
4	Jane	04-07-1977	

Replica C

id	name	DOB
1	Rob	06-06-1975
2	Helen	11-23-1982
3	John	02-15-1972
4	Jane	04-07-1977

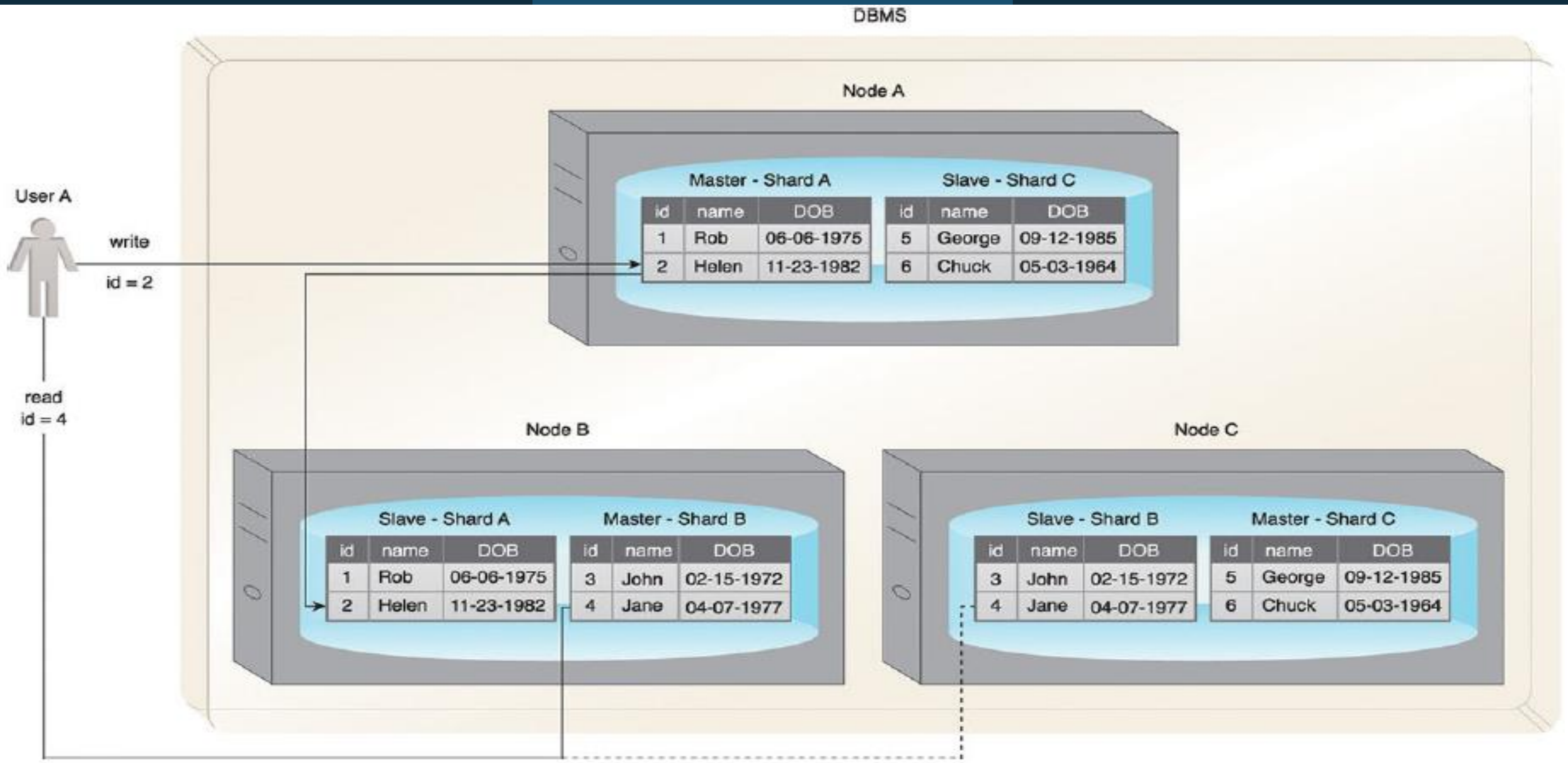
Sharding and Replication

Sharding and Replication

- To improve on the limited **fault tolerance** offered by sharding, while additionally benefiting from the increased **scalability** and **availability** of replication, both sharding and replication can be combined.

Sharding and master-slave replication

- When sharding is combined with master-slave replication, **multiple shards** become **slaves** of a single **master**, and the **master** itself is a **shard**.
- Although this results in **multiple masters**, a single slave-shard can only be managed by a single master-shard.
- **Write consistency** is maintained by the **master-shard**.
- However, if the master-shard becomes **non-operational** or a network outage occurs, fault tolerance with regards to write operations is impacted.
- Replicas of shards are kept on multiple slave nodes to provide fault tolerance for read operations.



Master - Shard A

id	name	DOB
1	Rob	06-06-1975
2	Helen	11-23-1982

Slave - Shard C

id	name	DOB
5	George	09-12-1985
6	Chuck	05-03-1964

Slave - Shard A

id	name	DOB
1	Rob	06-06-1975
2	Helen	11-23-1982

Master - Shard B

id	name	DOB
3	John	02-15-1972
4	Jane	04-07-1977

Slave - Shard B

id	name	DOB
3	John	02-15-1972
4	Jane	04-07-1977

Master - Shard C

id	name	DOB
5	George	09-12-1985
6	Chuck	05-03-1964

An example that shows the combination of sharding and master-slave replication.

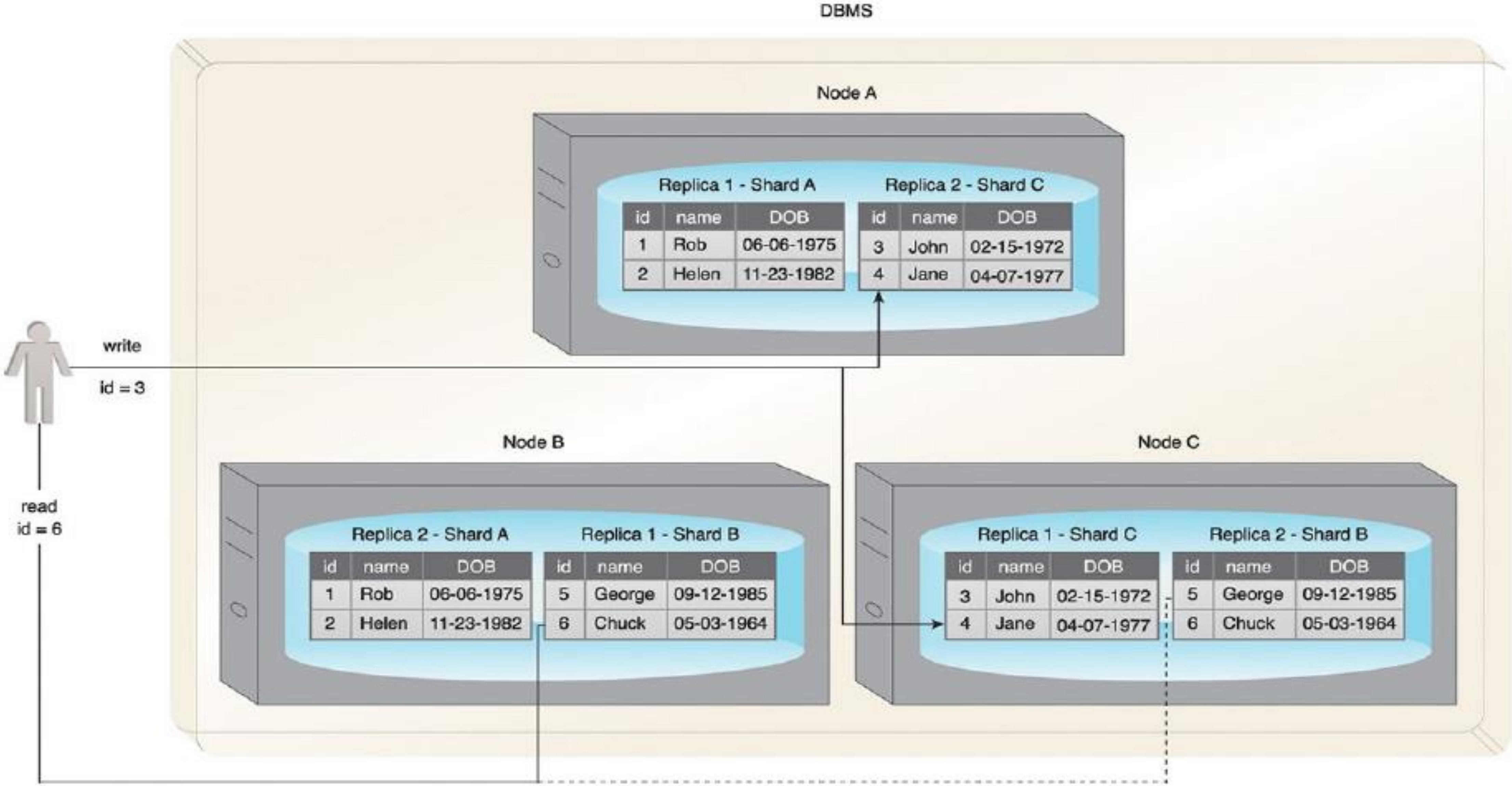
Sharding and master-slave replication

- The previous figure illustrates the following:
  - Each node acts both as a **master** and a **slave** for different shards.
  - Writes (id = 2) to **Shard A** are regulated by **Node A**, as it is the **master for Shard A**.
  - **Node A** replicates data (id = 2) to **Node B**, which is a slave for **Shard A**.
  - Reads (id = 4) can be served directly by either **Node B** or **Node C** as they each contain **Shard B**.

Sharding and peer-to-peer replication

- When combining sharding with peer-to-peer replication, **each shard is replicated** to multiple peers, and each peer is only responsible for a subset of the overall dataset.

- Collectively, this helps achieve increased **scalability** and **fault tolerance**. As there is no master involved, there is no single point of failure and fault-tolerance for both read and write operations is supported.



Node A

Replica 1 - Shard A

id	name	DOB
1	Rob	06-06-1975
2	Helen	11-23-1982

Replica 2 - Shard C

id	name	DOB
3	John	02-15-1972
4	Jane	04-07-1977

Node B

Replica 2 - Shard A

id	name	DOB
1	Rob	06-06-1975
2	Helen	11-23-1982

Replica 1 - Shard B

id	name	DOB
5	George	09-12-1985
6	Chuck	05-03-1964

Node C

Replica 1 - Shard C

id	name	DOB
3	John	02-15-1972
4	Jane	04-07-1977

Replica 2 - Shard B

id	name	DOB
5	George	09-12-1985
6	Chuck	05-03-1964

An example of the combination of sharding and peer-to-peer replication.

Sharding and peer-to-peer replication

- The previous figure illustrates the following:
  - Each node contains replicas of two different shards.
  - Writes (id = 3) are replicated to both **Node A** and **Node C** (Peers) as they are responsible for **Shard C**.
  - Reads (id = 6) can be served by either **Node B** or **Node C** as they each contain **Shard B**.

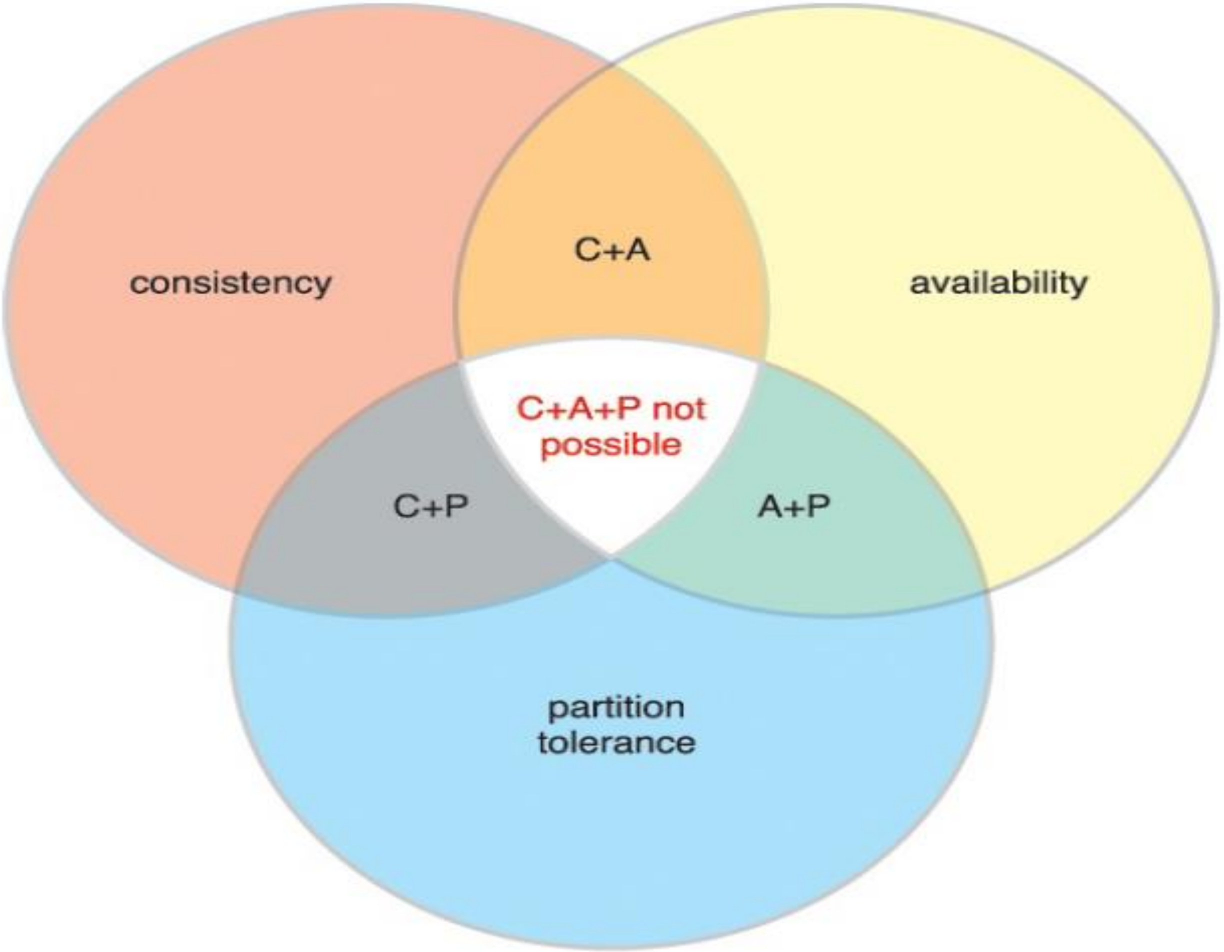
CAP theorem and BASE

CAP theorem

- The Consistency, Availability, and Partition tolerance (CAP) theorem, also known as Brewer’s theorem, expresses a triple constraint related to distributed database systems.
- It states that a distributed database system, running on a cluster, **can only provide two** of the following three properties:
  - **Consistency** – A read from any node results in the same data across multiple nodes.
  - **Availability** – A read/write request will always be acknowledged in the form of a success or a failure.
  - **Partition tolerance** – The cluster continues to function even if there is a "partition" (communication break) between two nodes.

CAP theorem

- A Venn diagram summarizing the CAP theorem:



CAP theorem

- The following scenarios demonstrate why only two of the three properties of the CAP theorem are simultaneously supportable:
  - If consistency (C) and availability (A) are required, available nodes need to communicate to ensure consistency (C). Therefore, partition tolerance (P) is not possible.
  - If consistency (C) and partition tolerance (P) are required, nodes cannot remain available (A) as the nodes will become unavailable while achieving a state of consistency (C).
  - If availability (A) and partition tolerance (P) are required, then consistency (C) is not possible because of the data communication requirement between the nodes. So, the database can remain available (A) but with inconsistent results.

CAP theorem

- In a distributed database, scalability and fault tolerance can be improved through additional nodes, although this challenges consistency (C).
- Distributed database systems cannot be 100% partition tolerant (P).
- Although communication outages are rare and temporary, partition tolerance (P) must always be supported by a distributed database; therefore, CAP is generally a choice between choosing either **C+P** or **A+P**. The requirements of the system will dictate which is chosen.

BASE

- BASE is a database design principle based on the CAP theorem and leveraged by database systems that use distributed technology. BASE stands for:
  - basically available
  - soft state
  - eventual consistency
- When a database supports BASE, it favors availability over consistency. In other words, the database is **A+P** from a CAP perspective.
- In essence, BASE leverages optimistic concurrency by relaxing the strong consistency constraints mandated by the ACID properties (recall ACID: **A**tomicity **C**onsistency **I**solation **D**urability).

BASE

- If a database is "**basically available**," that database will always acknowledge a client's request, either in the form of the requested data or a success/failure notification.
- **Soft state** means that a database may be in an **inconsistent** state when data is read; thus, the results may change if the same data is requested again.
  - This is because the data could be updated for consistency, even though no user has written to the database between the two reads.
  - This property is closely related to eventual consistency.

BASE

- **Eventual consistency** is the state in which reads by different clients, immediately following a write to the database, may not return consistent results.
  - The database only attains consistency once the changes have been propagated to all nodes.
  - While the database is in the process of attaining the state of eventual consistency, it will be in a soft state.

BASE

- BASE emphasizes availability over immediate consistency, in contrast to ACID, which ensures immediate consistency at the expense of availability due to *record locking*.
- This soft approach toward consistency allows BASE compliant databases to serve multiple clients **without any latency** in spite of serving **inconsistent** results.
- However, BASE-compliant databases are <u>not useful </u>for transactional systems where lack of <u>consistency is a concern</u>.

Thank You