Module 2

- Big Data Storage Concepts- Clusters - File Systems and Distributed File Systems- Sharding – Replication – Sharding and Replication – CAP Theorem – ACID – BASE

- Big Data Storage Technology : On-Disk Storage Devices – Distributed File Systems-RDBMS Databases - NoSQL Databases-NewSQL Databases -In-Memory Storage Devices -In-Memory Data Grids -In-Memory Databases

Big Data Storage Concepts

- Data acquired from external sources is often not in a format or structure that can be directly processed.
- To overcome these incompatibilities and prepare data for storage and processing, data wrangling is necessary.
- Data wrangling includes steps to filter, cleanse and otherwise prepare the data for downstream analysis.
- From a storage perspective, a copy of the data is first stored in its acquired format, and, after wrangling, the prepared data needs to be stored again.

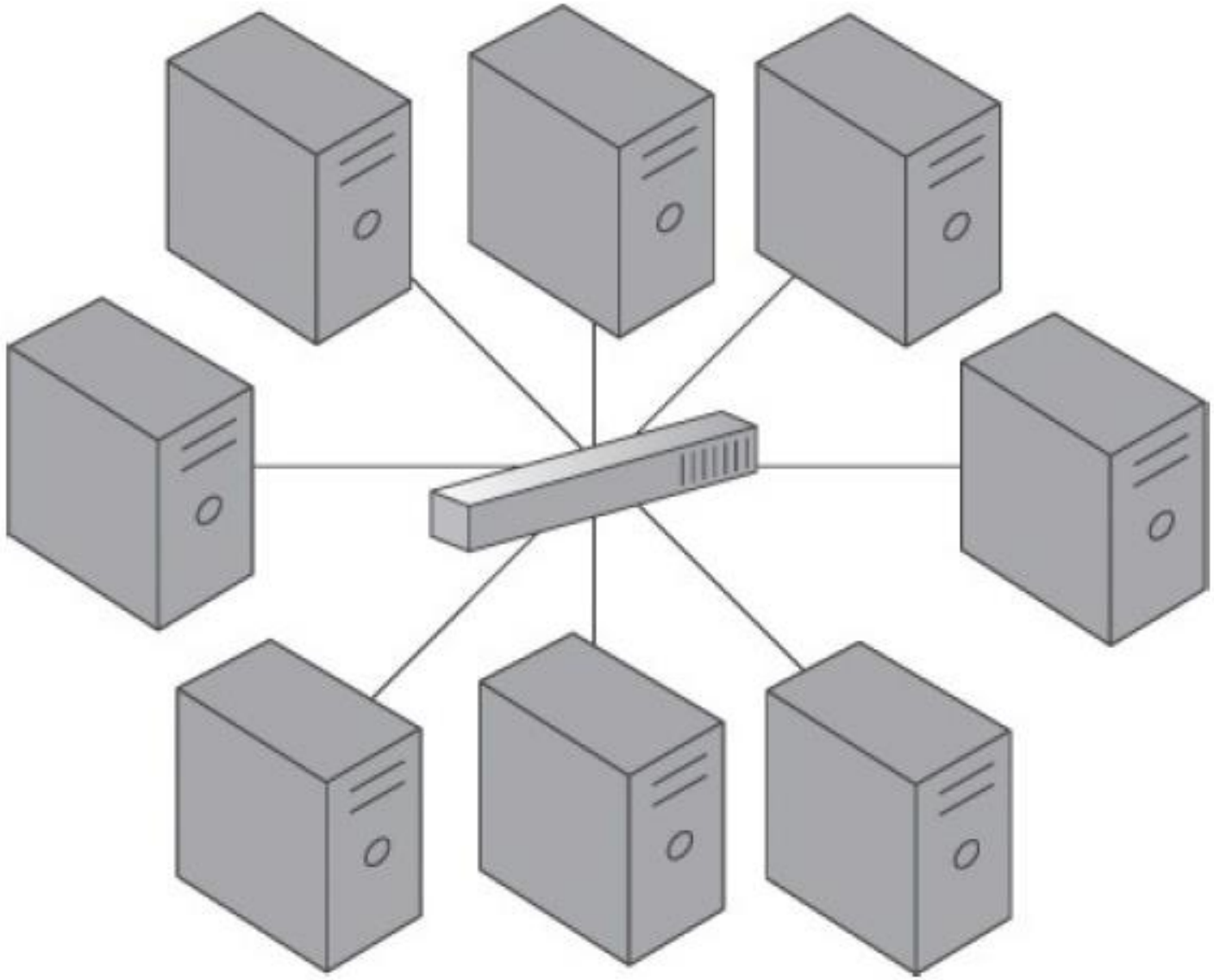
Typically, storage is required whenever the following occurs:

- external datasets are acquired, or internal data will be used in a Big Data environment
- data is manipulated to be made amenable for data analysis
- data is processed via an ETL activity, or output is generated as a result of an analytical operation

- clusters
- file systems and distributed files systems
- sharding
- replication
- CAP theorem
- ACID
- BASE

Clusters

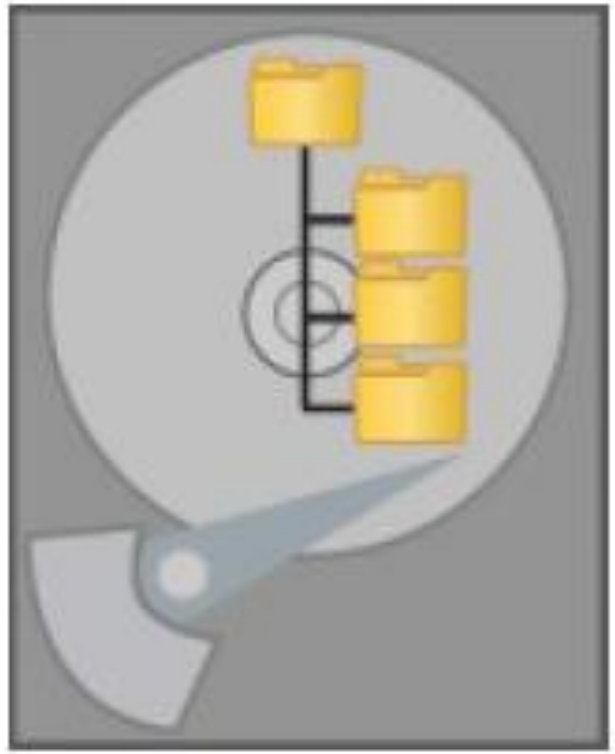
- A cluster is a tightly coupled collection of servers, or nodes.
- These servers usually have the same hardware specifications and are connected together via a network to work as a single unit.
- Each node in the cluster has its own dedicated resources, such as memory, a processor, and a hard drive. A cluster can execute a task by splitting it into small pieces and distributing their execution onto different computers that belong to the cluster.



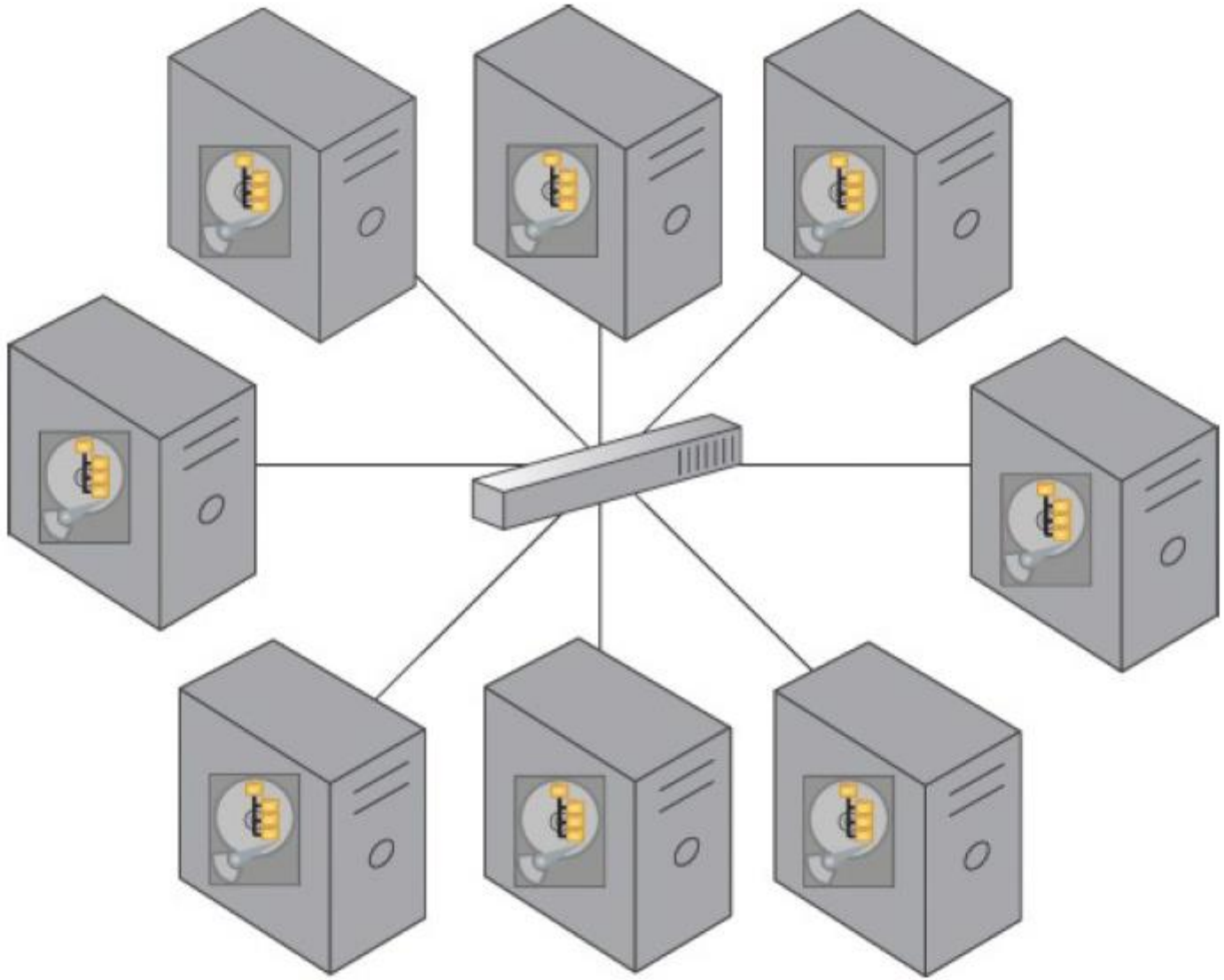
The symbol used to represent a cluster.

File Systems and Distributed File Systems

- A file system is the method of storing and organizing data on a storage device, such as flash drives, DVDs and hard drives.
- A file is an atomic unit of storage used by the file system to store data.
- A file system provides a logical view of the data stored on the storage device and presents it as a tree structure of directories and files as pictured in Figure
- Operating systems employ file systems to store and retrieve data on behalf of applications.
- Each operating system provides support for one or more file systems, for example NTFS on Microsoft Windows and ext on Linux



The symbol used to represent a file system.



The symbol used to represent distributed file systems.

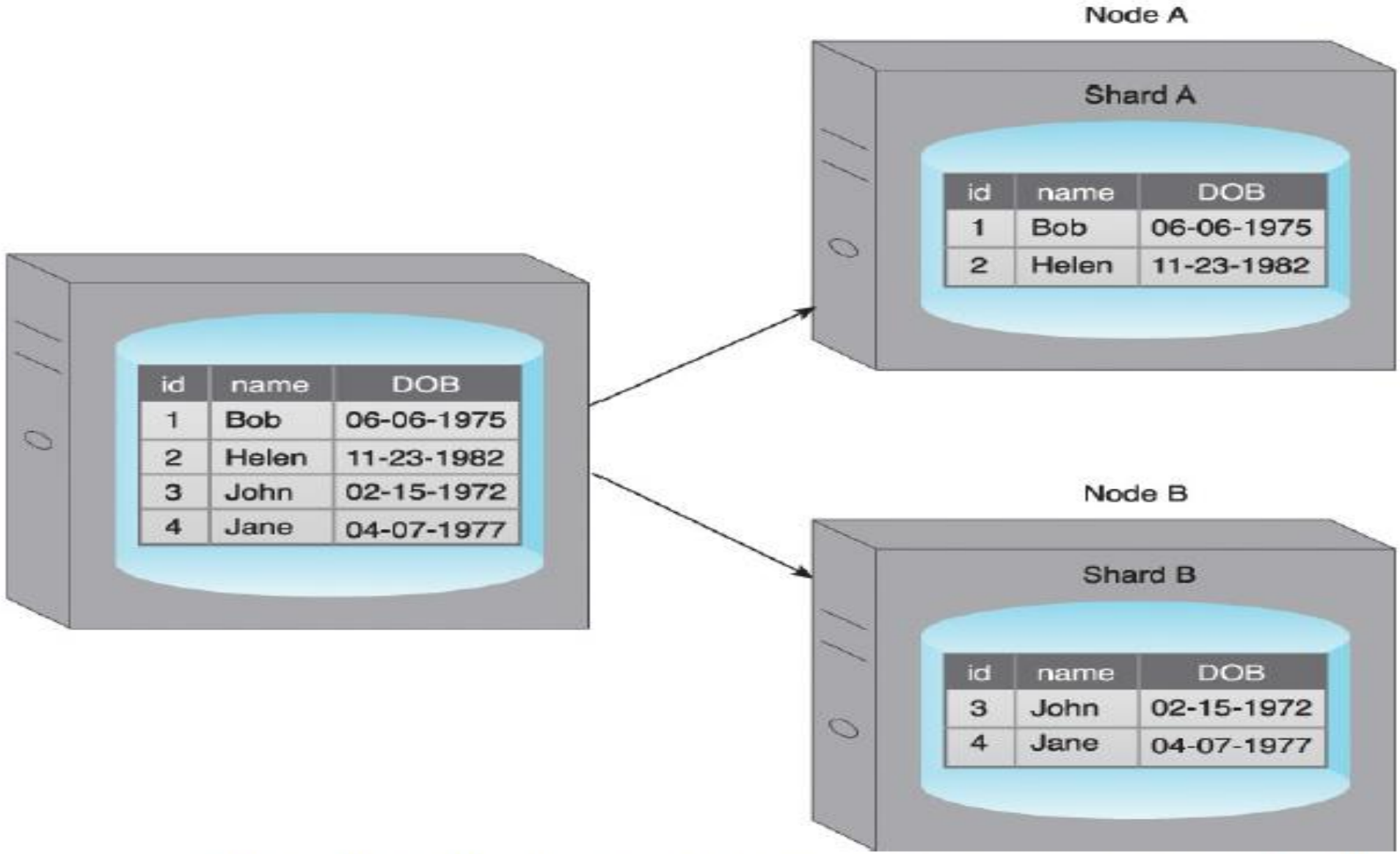
Sharding

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.



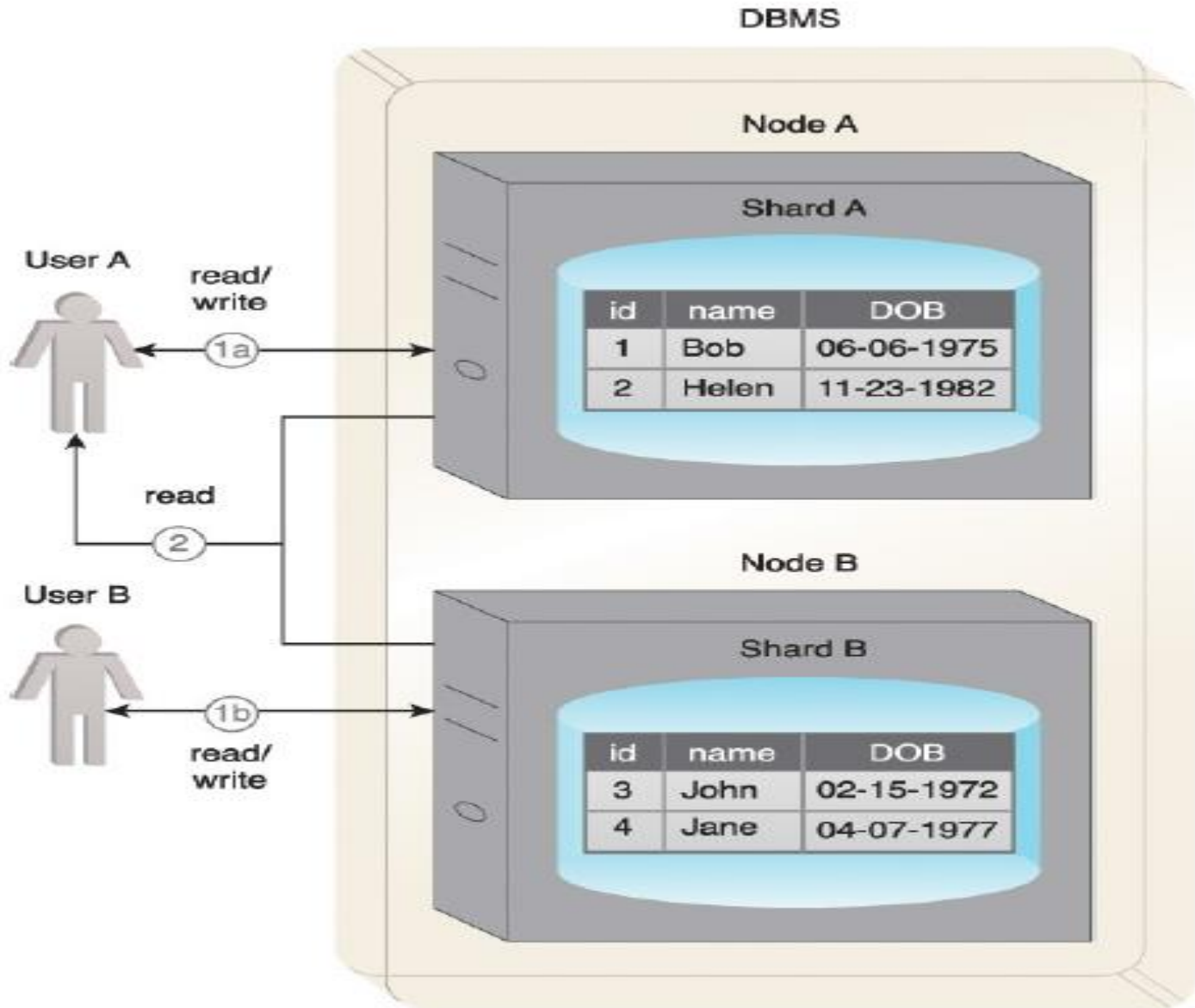
An example of sharding where a dataset is spread across Node A and Node B, resulting in Shard A and Shard B, respectively.

Sharding

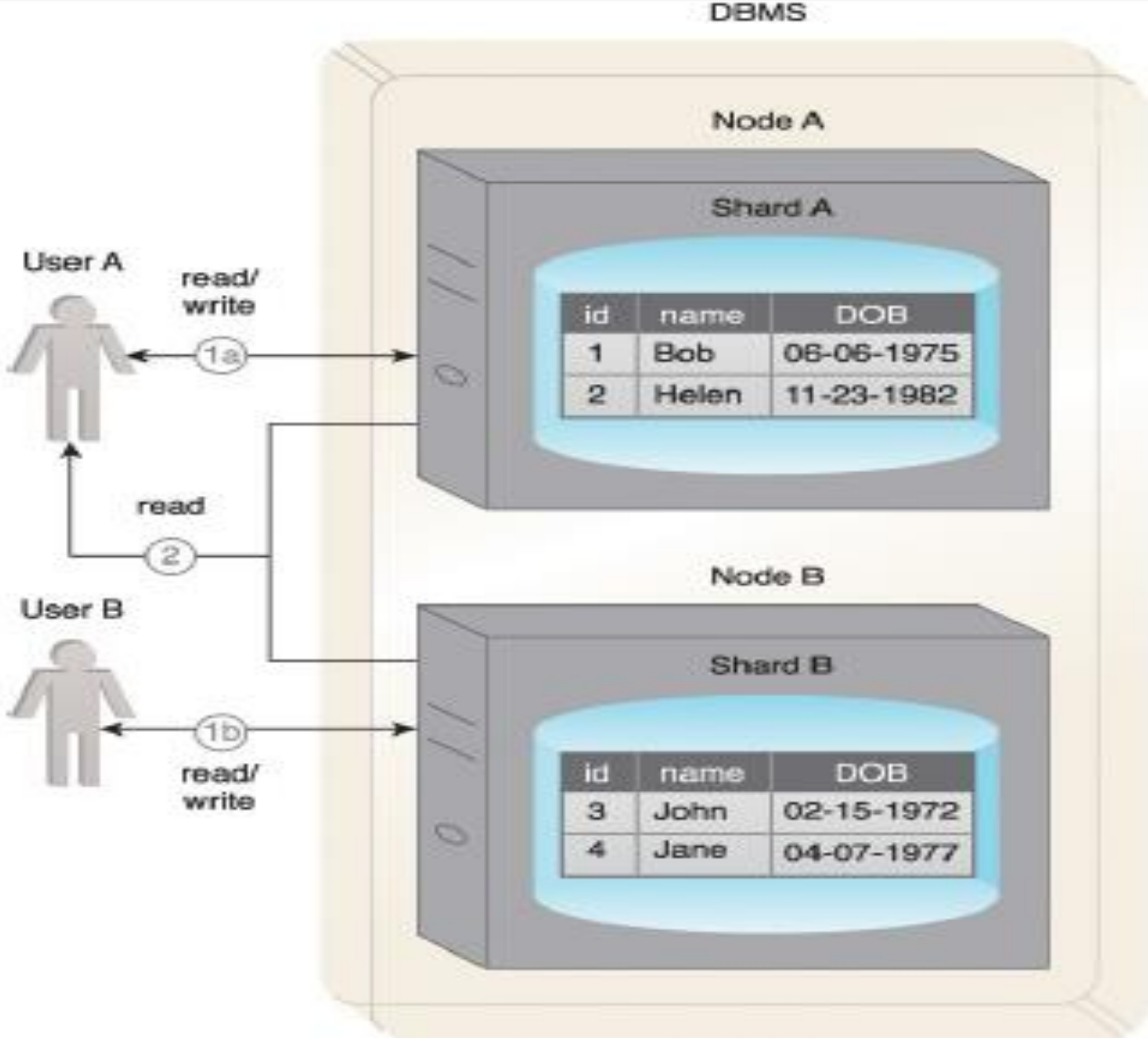
- Sharding is often transparent to the client, but this is not a requirement.
- Sharding allows the distribution of processing loads across multiple nodes to achieve horizontal scalability.
- Horizontal scaling is a method for increasing a system's capacity by adding similar or higher capacity resources alongside existing resources. Since each node is responsible for only a part of the whole dataset, read/write times are greatly improved.

Figure presents an illustration of how sharding works in practice:

- **1.** Each shard can independently service reads and writes for the specific subset of data that it is responsible for.
- **2.** Depending on the query, data may need to be fetched from both shards.



A sharding example where data is fetched from both Node A and Node B.



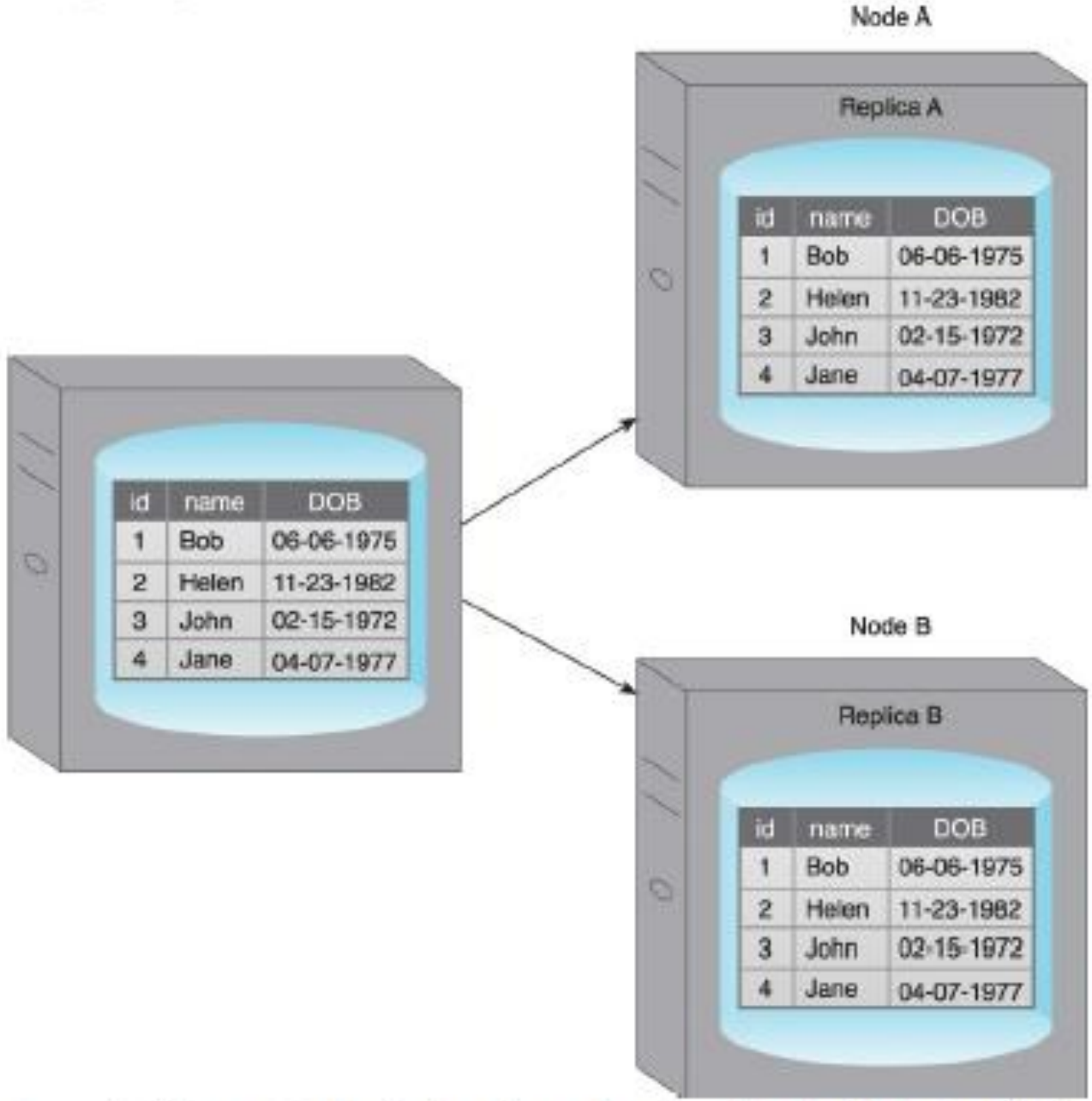
A sharding example where data is fetched from both Node A and Node B.

Benefit of sharding

- 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

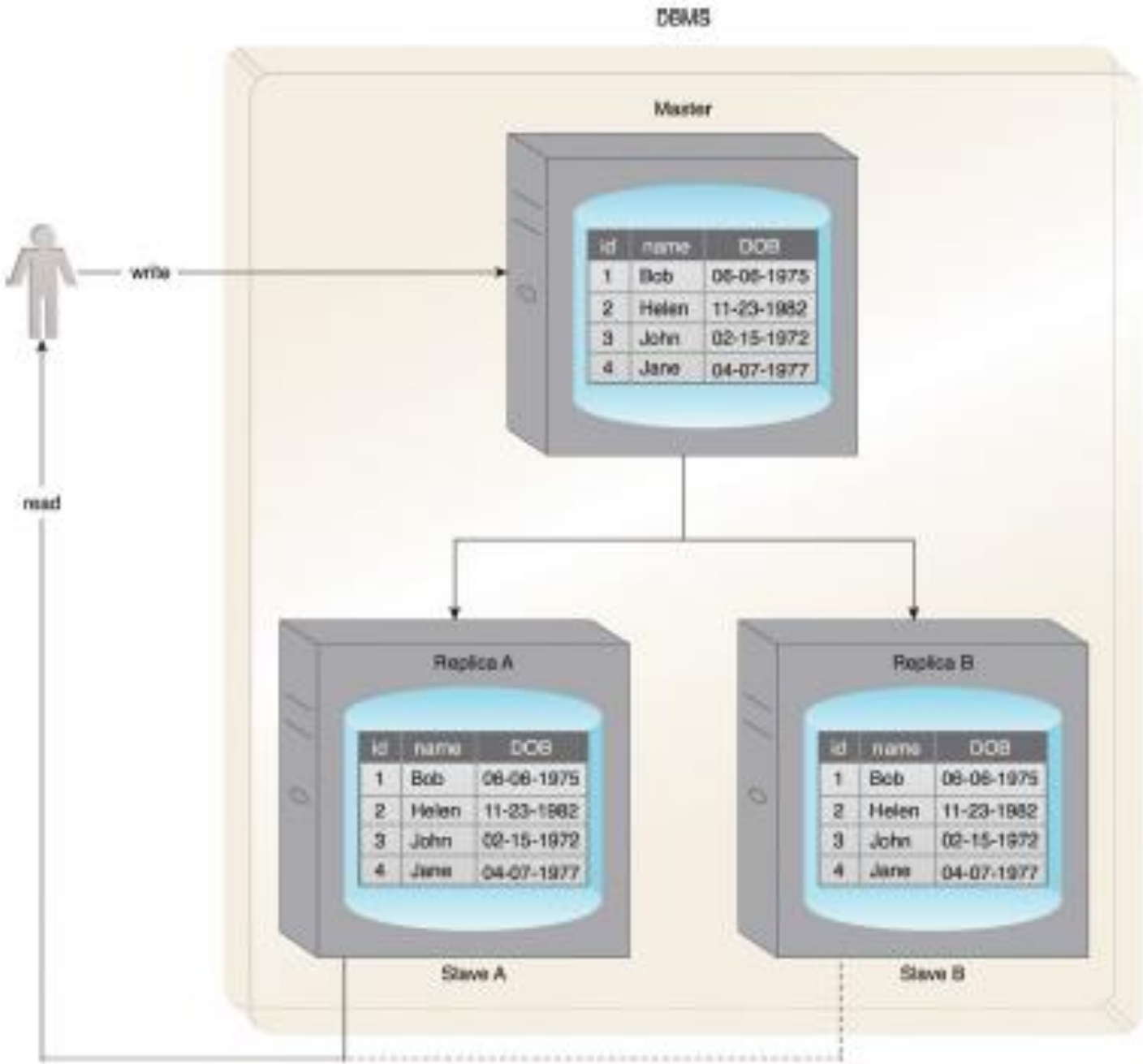
- Replication stores multiple copies of a dataset, known as *replicas*, on multiple nodes
- Replication provides scalability and availability due to the fact that the same data is replicated on various nodes.
- Fault tolerance is also achieved since data redundancy ensures that data is not lost when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



An example of replication where a dataset is replicated to Node A and Node B, resulting in Replica A and Replica B.

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.
- In Figure writes are managed by the master node and data can be read from either Slave A or Slave B.



An example of master-slave replication where Master A is the single point of contact for all writes, and data can be read from Slave A and Slave B.

- 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.
- 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 resurrected from a backup of the master node, or a new master node is chosen from the slave nodes.
- 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.

- Figure illustrates a scenario where read inconsistency occurs.

  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.

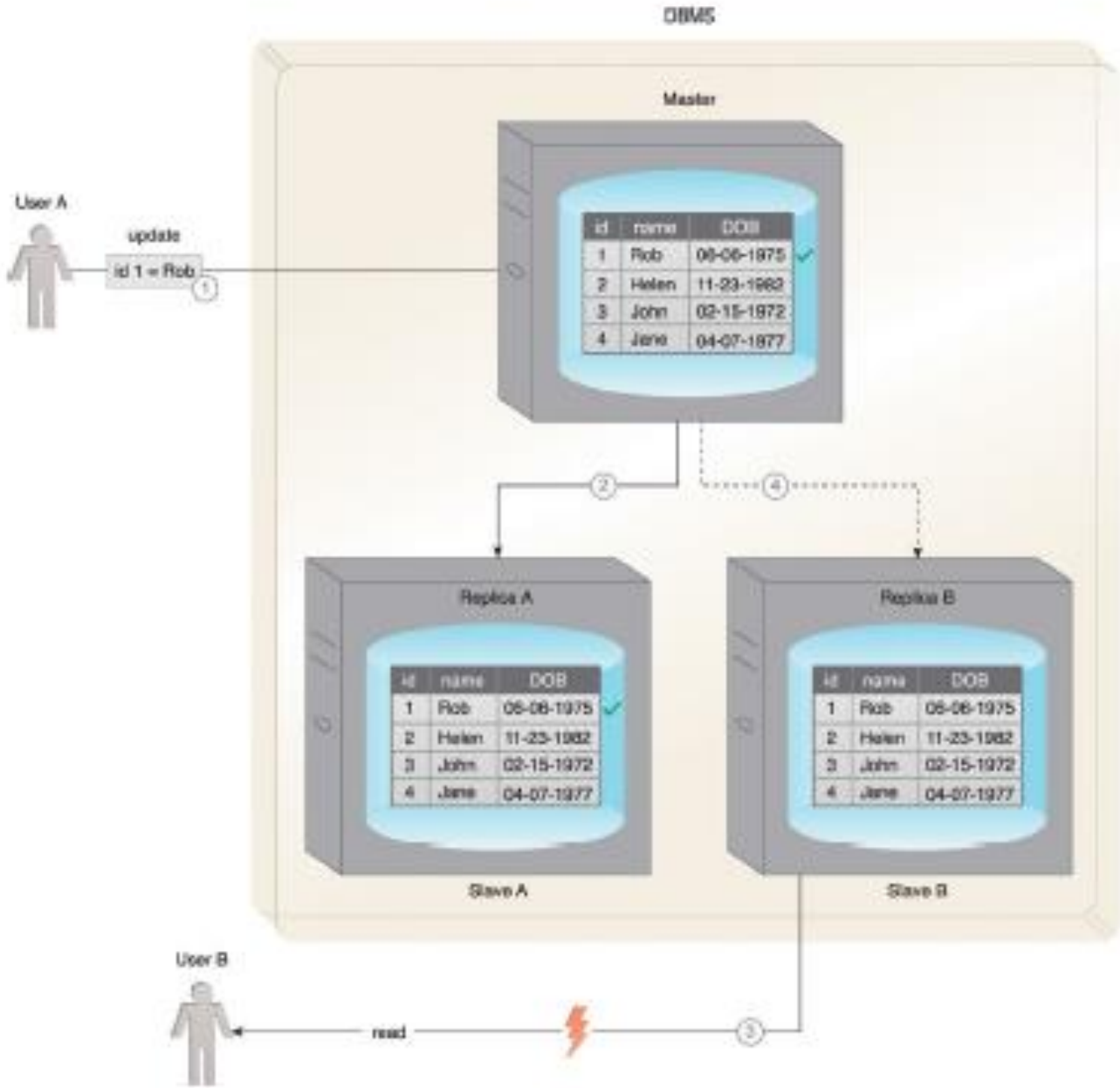


Figure 5.9 An example of master-slave replication where read inconsistency occurs.

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, as illustrated

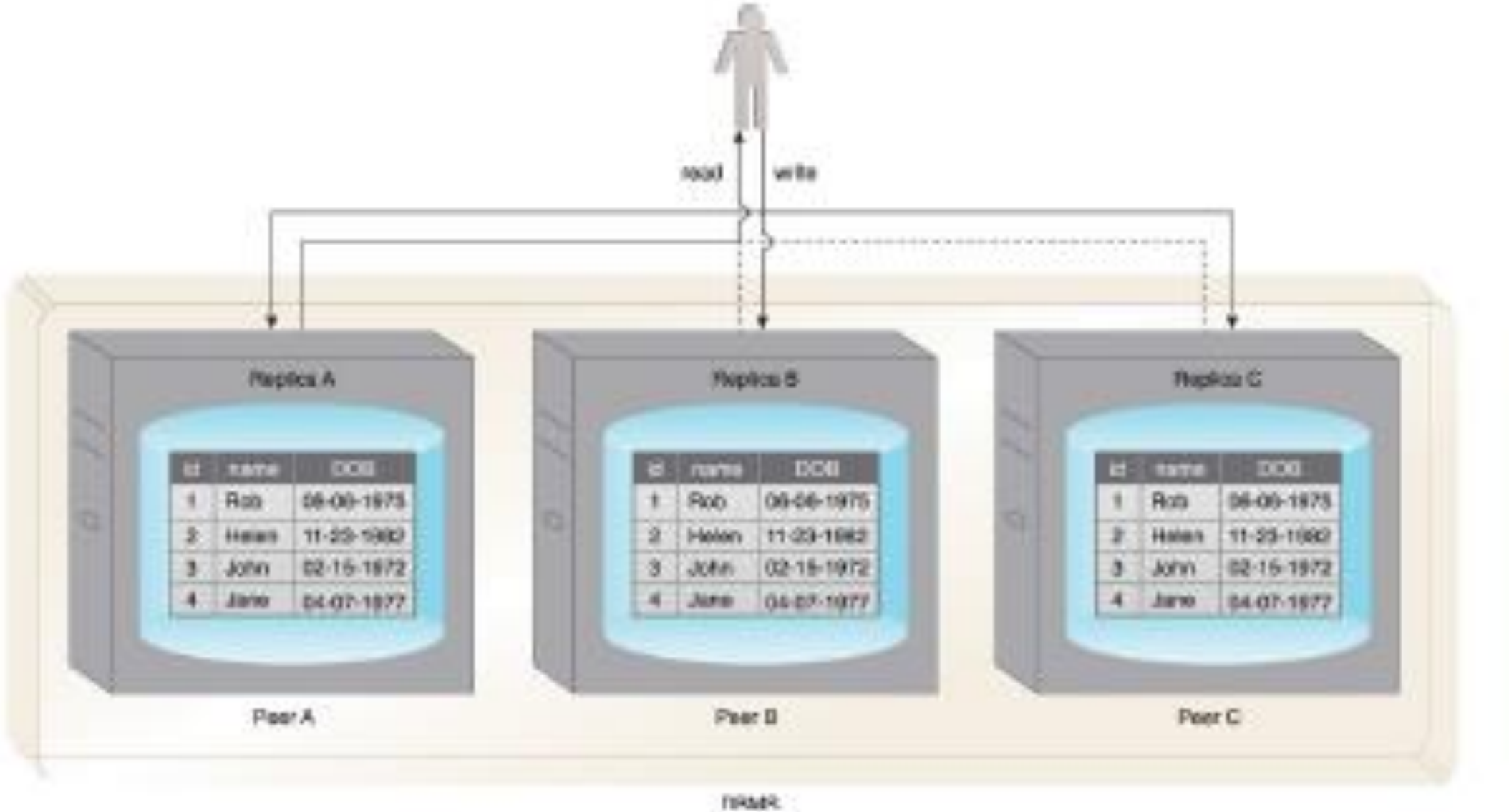


Figure 5.10 Writes are copied to Peers A, B and C simultaneously. Data is read from Peer A, but it can also be read from Peers B or C.

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.

CAP THEOREM

- 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 database system can tolerate communication outages that split the cluster into multiple silos and can still service read/write requests

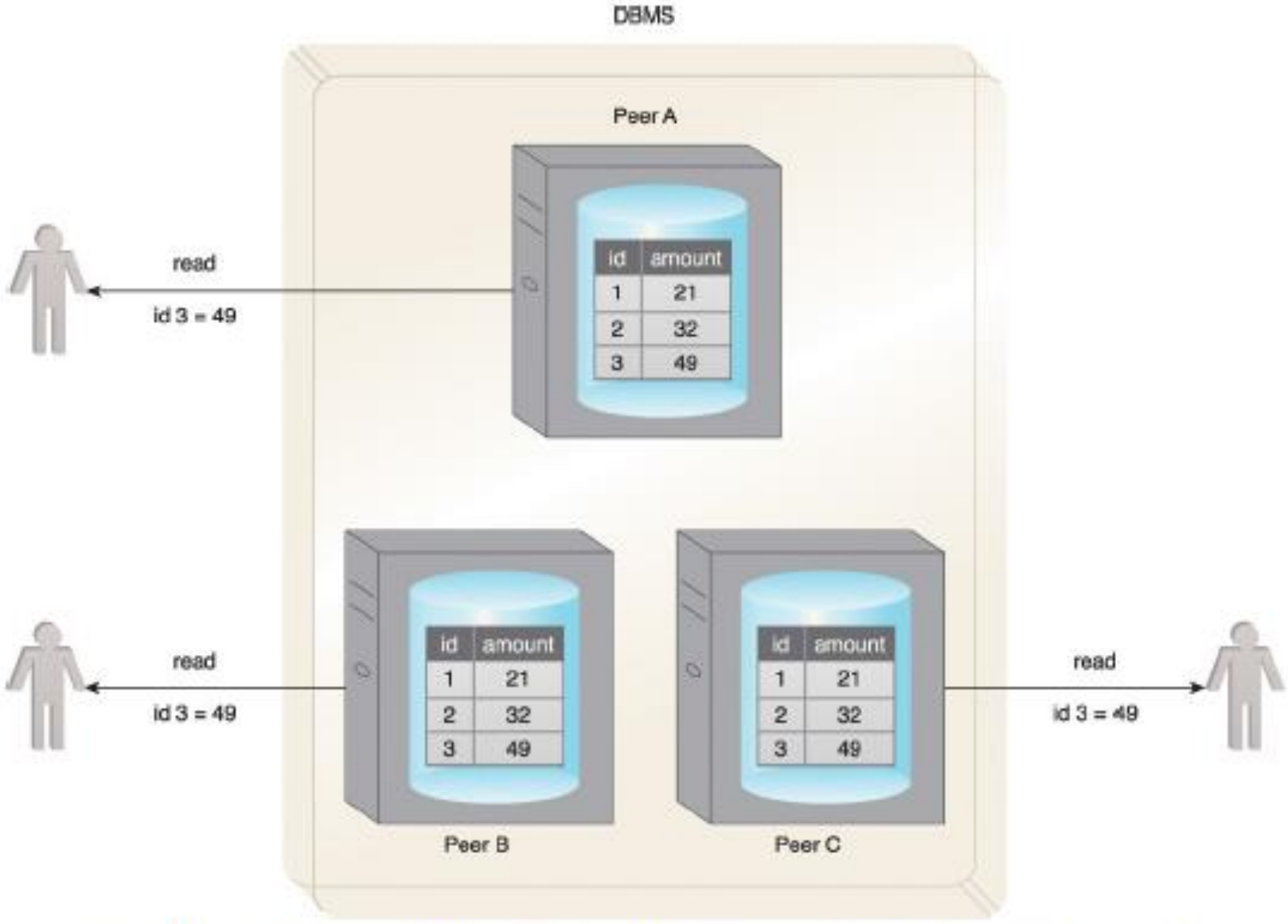


Figure 5.15 Consistency: all three users get the same value for the amount column even though three different nodes are serving the record.

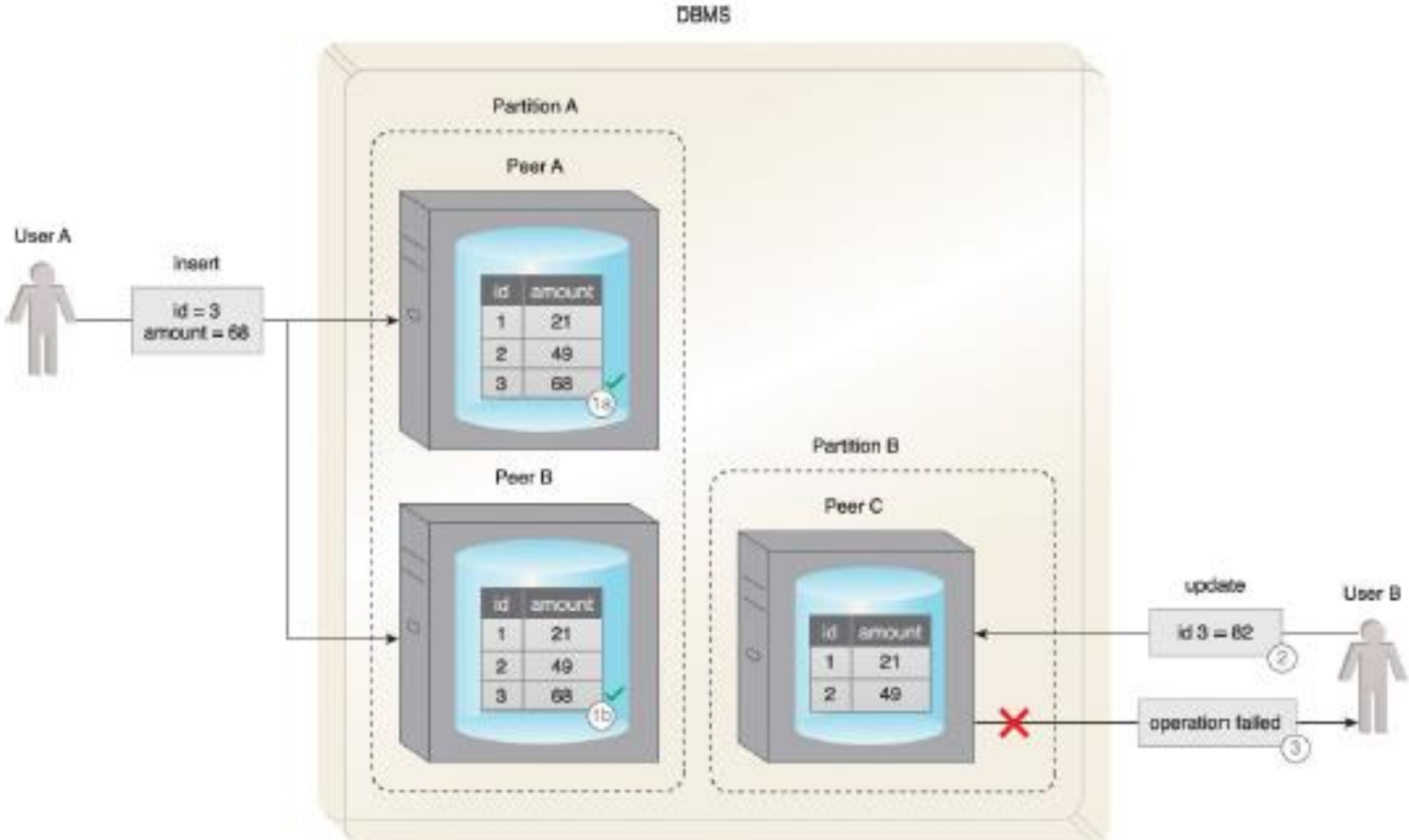


Figure 5.16 Availability and partition tolerance: in the event of a communication failure, requests from both users are still serviced (1, 2). However, with User B, the update fails as the record with id = 3 has not been copied over to Peer C. The user is duly notified (3) that the update has failed.

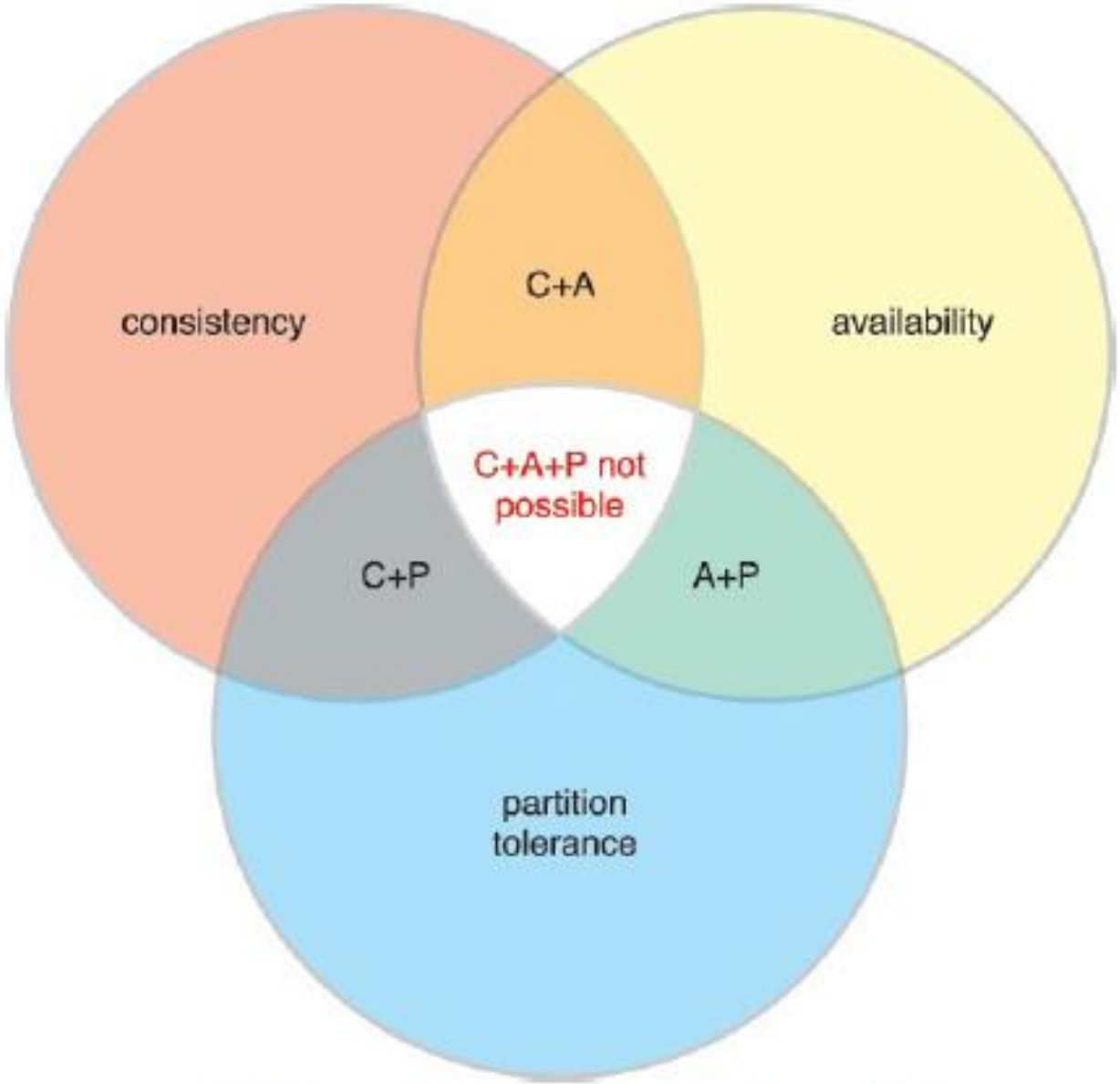


Figure 5.17 A Venn diagram summarizing the CAP theorem.

- 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.
- In a distributed database, scalability and fault tolerance can be improved through additional nodes, although this challenges consistency (C). The addition of nodes can also cause availability (A) to suffer due to the latency caused by increased communication between nodes.
- 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.

ACID

- ACID is a database design principle related to transaction management. It is an acronym that stands for:
  - atomicity
  - consistency
  - isolation
  - durability

Atomicity

- ACID is the traditional approach to database transaction management as it is leveraged by relational database management systems.
- Atomicity ensures that all operations will always succeed or fail completely. In other words, there are no partial transactions.
- The following steps are illustrated in Figure 5.18:
    1. A user attempts to update three records as a part of a transaction.
    2. Two records are successfully updated before the occurrence of an error.
    3. As a result, the database roll backs any partial effects of the transaction and puts the system back to its prior state.

• consistency

- Consistency ensures that the database will always remain in a consistent state by ensuring that only data that conforms to the constraints of the database schema can be written to the database. Thus a database that is in a consistent state will remain in a consistent state following a successful transaction.
- In Figure 5.19:
- **1.** A user attempts to update the amount column of the table that is of type float with a
- varchar value.
- **2.** The database applies its validation check and rejects this update because the value
- violates the constraint checks for the amount column.

Isolation

- Isolation ensures that the results of a transaction are not visible to other operations until it is complete.
- In Figure 5.20:
- **1.** User A attempts to update two records as part of a transaction.
- **2.** The database successfully updates the first record.
- **3.** However, before it can update the second record, User B attempts to update the
- same record. The database does not permit User B's update until User A's update
- succeeds or fails in full. This occurs because the record with id3 is locked by the
- database until the transaction is complete.

Durability

- Durability ensures that the results of an operation are permanent. In other words, once a transaction has been committed, it cannot be rolled back. This is irrespective of any system failure.
- In Figure 5.21:
- **1.** A user updates a record as part of a transaction.
- **2.** The database successfully updates the record.
- **3.** Right after this update, a power failure occurs. The database maintains its state while there is no power.
- **4.** The power is resumed.
- **5.** The database serves the record as per last update when requested by the user.

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.
- A+P from a CAP perspective.
- BASE leverages optimistic concurrency by relaxing the strong consistency constraints mandated by the ACID properties.

basically available

- 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

- 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.
- In Figure 5.24:
- **1.** User A updates a record on Peer A.
- **2.** Before the other peers are updated, User B requests the same record from Peer C.
- **3.** The database is now in a soft state, and stale data is returned to User B.

Eventual consistency

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.

- In Figure 5.25:
- **1.** User A updates a record.
- **2.** The record only gets updated at Peer A, but before the other peers can be updated,
- User B requests the same record.
- **3.** The database is now in a soft state. Stale data is returned to User B from Peer C.
- **4.** However, the consistency is eventually attained, and User C gets the correct value.

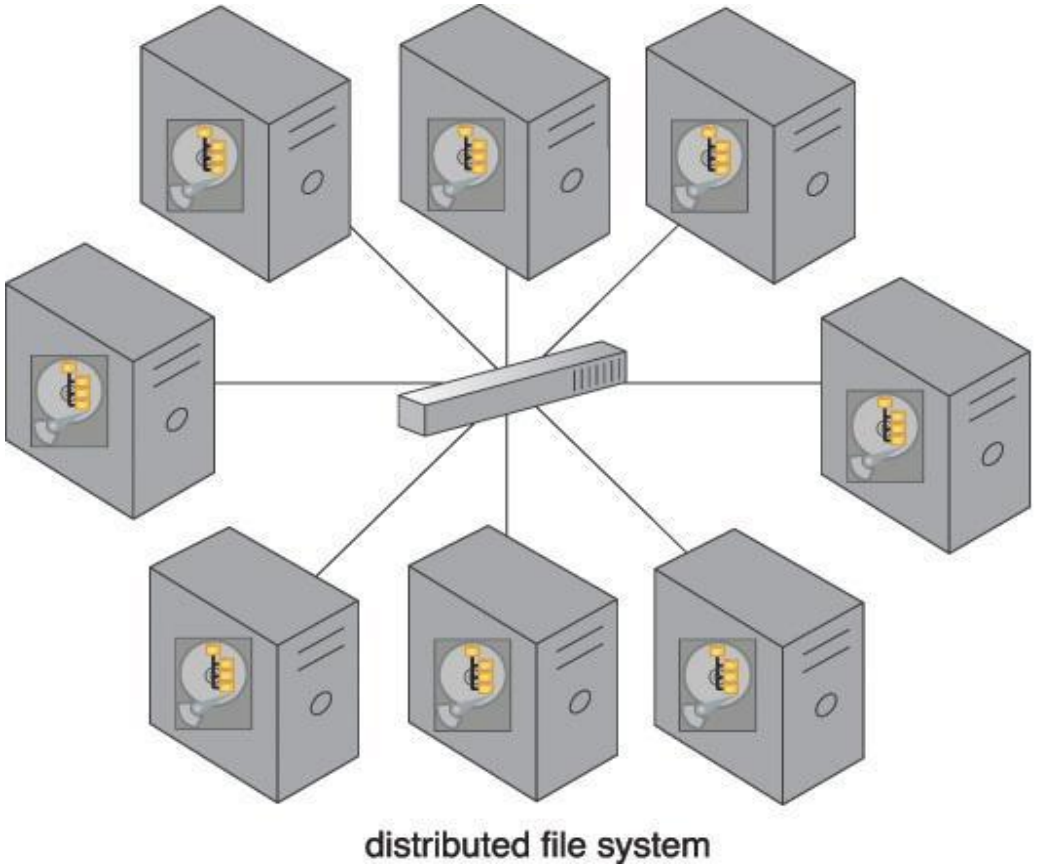
- BASE emphasizes availability over immediate consistency, in contrast to ACID, which ensures immediate consistency at the expense of availability due to record locking.
- This soft approach toward consistency allows BASE compliant databases to serve multiple
- clients without any latency albeit serving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

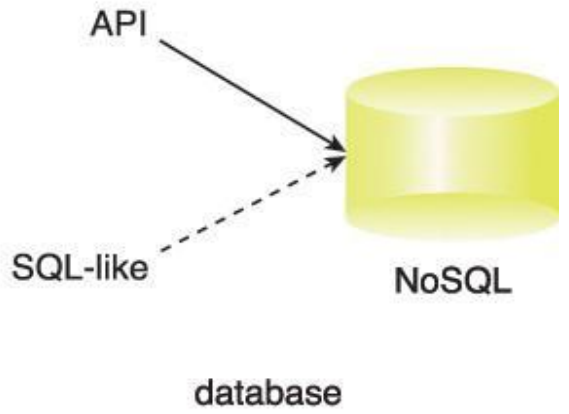
Big Data Storage Technology – On-Disk Storage Devices – Distributed File Systems, RDBMS Databases, NoSQL Databases, NewSQL Databases – In-Memory Storage Devices: In-Memory Data Grids, In-Memory Databases.

Big Data Storage Technology

- Big Data has pushed the storage boundary to unified views of the available memory and disk storage of a cluster.
- If more storage is needed, horizontal scalability allows the expansion of the cluster through the addition of more nodes.
- Innovative approaches deliver realtime analytics via in-memory storage.
- Batch-based processing also accelerated by the performance of Solid State Drives (SSDs)

On-Disk Storage Device





On-disk storage can be implemented with a distributed file system or a database.

On-Disk Storage Device

- generally utilizes low cost hard-disk drives for long-term storage.
- can be implemented via a distributed file system or a database

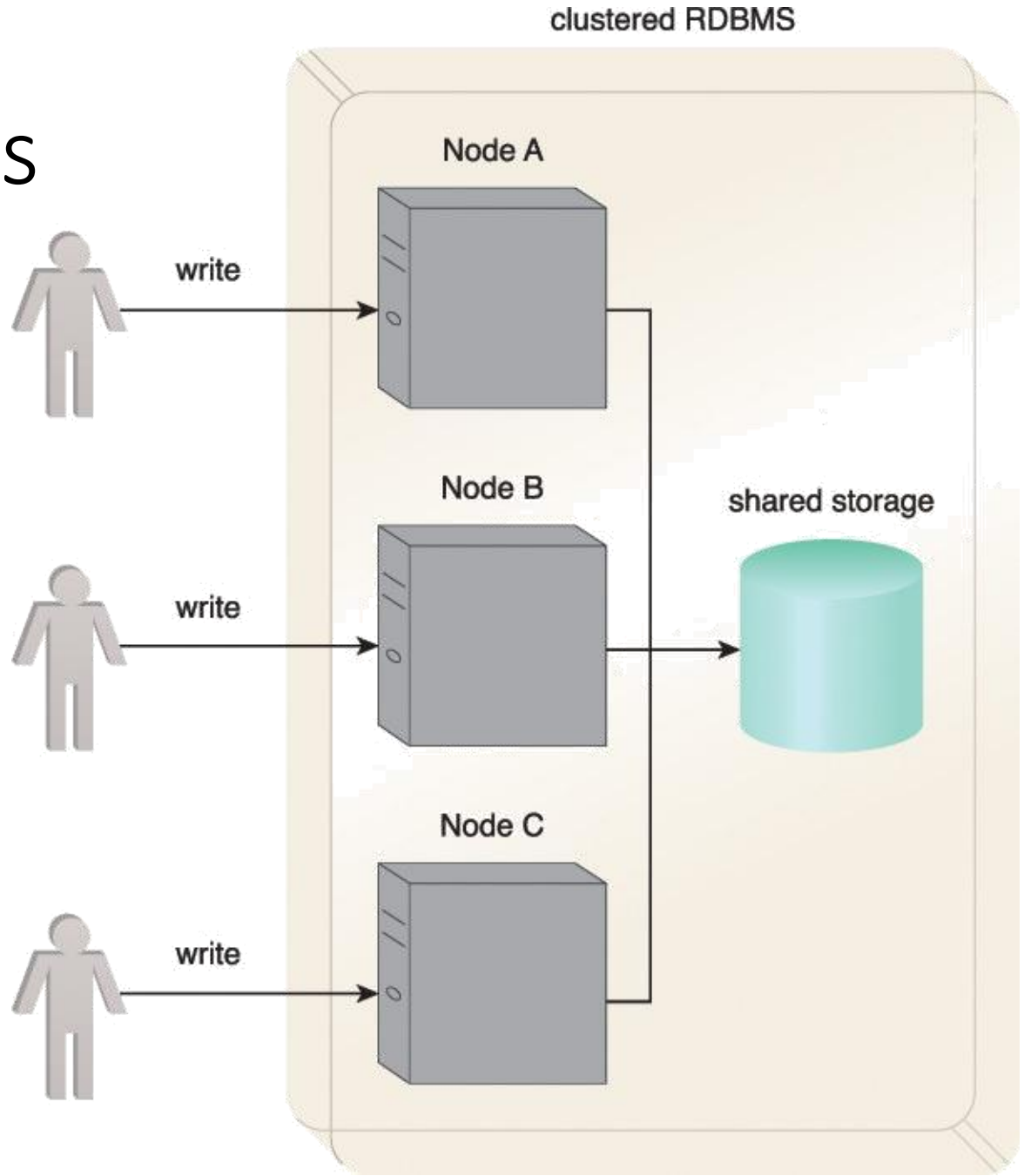
Distributed File Systems

- Distributed file systems support schema-less data storage.
- copy data to multiple locations via replication and thus provides
  - out of box redundancy
  - high availability
- provides simple, fast access data storage
  - capable of storing large datasets that are non-relational in nature, such as semi-structured and unstructured data.
- work best with fewer but larger files accessed in a sequential manner.
- To enable optimum storage and processing
  - Multiple smaller files are generally combined into a single file.
- This increases performance when data must be accessed in streaming mode with no random reads and writes

Distributed File Systems-Limitations

- not ideal for datasets comprising a large number of small files
  - as this creates excessive disk-seek activity, slowing down the overall data access.
- more overhead involved in processing multiple smaller files
  - as dedicated processes are generally spawned by the processing engine at runtime for processing each file before the results are synchronized from across the cluster.
- It provides an inexpensive storage option for storing large amounts of data over a long period of time that needs to remain online

RDBMS Databases



RDBMS Databases

- Relational database management systems (RDBMSs)
  - good for handling transactional workloads involving small amounts of data with random read/write properties.
- RDBMSs are ACID-compliant
  - they are generally restricted to a single node.
- For this reason, RDBMSs do not provide
  - out-of-the-box redundancy
  - fault tolerance.

RDBMS Databases

- Scalability is required to handle large volumes of data arriving at a fast pace.
- RDBMSs employ vertical scaling, not horizontal scaling, which is a more costly and disruptive scaling strategy.
- This makes RDBMSs less than ideal for long-term storage of data that accumulates over time.
- A clustered rational database uses a shared storage architecture, which is a potential single point of failure that affects the availability of the database.

RDBMS Databases

- Relational databases need to be manually sharded, mostly using application logic.
- Application logic needs to know which shard to query in order to get the required data.
- This further complicates data processing when data from multiple shards is required.

- The following steps are shown in Figure:

1. A user writes a record (id = 2).

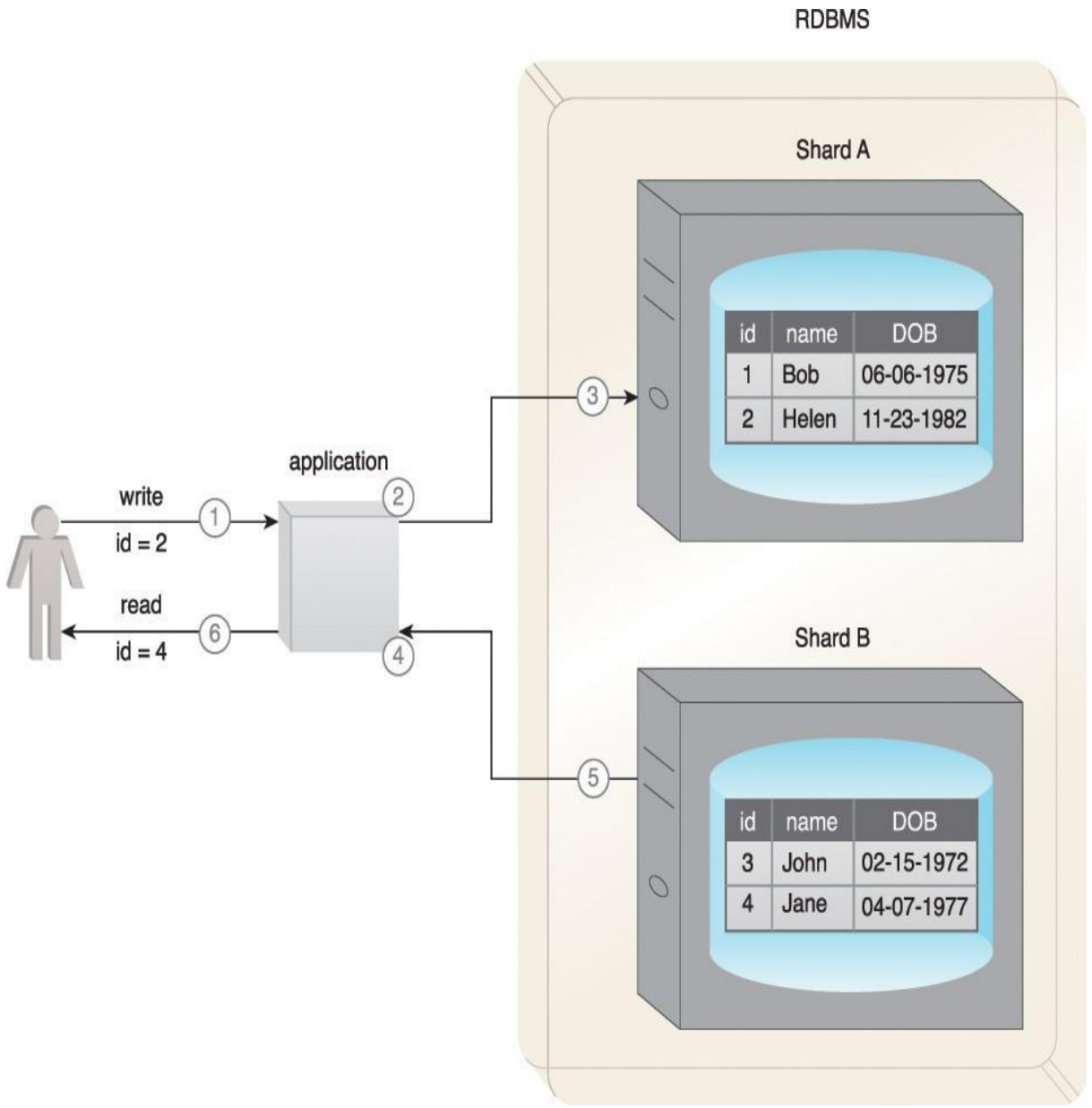
2. The application logic determines which shard it should be written to.

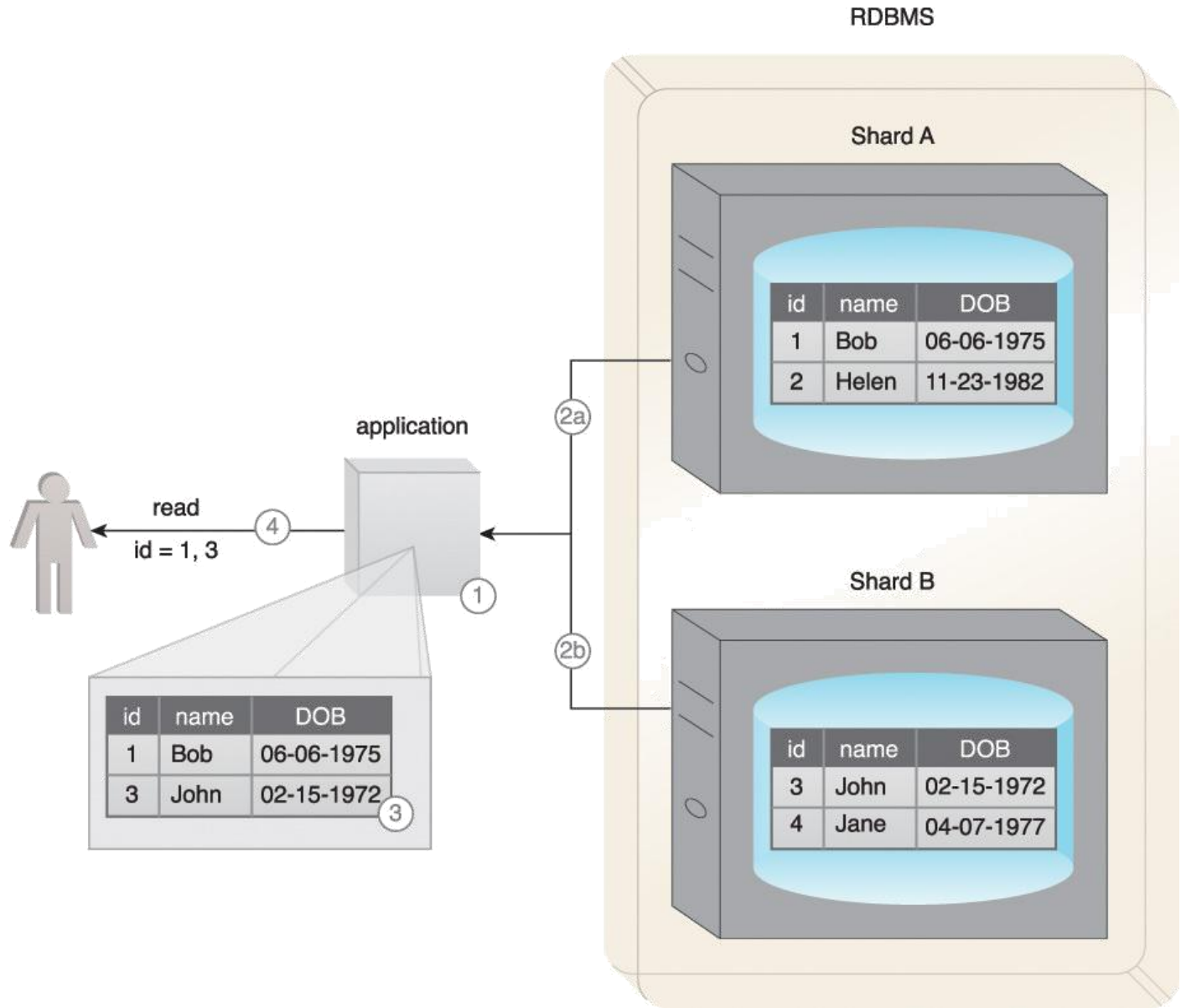
3. It is sent to the shard determined by the application logic.

4. The user reads a record (id = 4), and the application logic determines which shard contains the data.

5. The data is read and returned to the application.

6. The application then returns the record to the user.




RDBMS
Shard A
id name DOB
1 Bob 06-06-1975
2 Helen 11-23-1982
application
2a
read
id = 1, 3
4
1
Shard B
2b
id name DOB
1 Bob 06-06-1975
3 John 02-15-1972
3
id name DOB
3 John 02-15-1972
4 Jane 04-07-1977

NoSQL Databases

- Relational databases generally require data to adhere to a schema.
- As a result, storage of semi-structured and unstructured data whose schemas are non-relational is not directly supported.
- Furthermore, with a relational database schema conformance is validated at the time of data insert or update by checking the data against the constraints of the schema.

NoSQL Databases

- This latency makes relational databases a less than ideal choice for storing high velocity data that needs a highly available database storage device with fast data write capability.
- Not-only SQL (NoSQL) refers to technologies used to develop next generation non relational databases that are highly scalable and fault-tolerant

NoSQL Databases - Characteristics

Schema-less data model – Data can exist in its raw form.

Scale out rather than scale up – More nodes can be added to obtain additional storage with a NoSQL database, in contrast to having to replace the existing node with a better, higher performance/capacity one.

Highly available – This is built on cluster-based technologies that provide fault tolerance out of the box.

NoSQL Databases - Characteristics

Lower operational costs – Many NoSQL databases are built on Open Source platforms with no licensing costs. They can often be deployed on commodity hardware.

Eventual consistency – Data reads across multiple nodes but may not be consistent immediately after a write. However, all nodes will eventually be in a consistent state

NoSQL Databases - Characteristics

BASE, not ACID – BASE compliance requires a database to maintain high availability in the event of network/node failure, while not requiring the database to be in a consistent state whenever an update occurs. The database can be in a soft/inconsistent state until it eventually attains consistency

API driven data access – Data access is generally supported via API based queries, including RESTful APIs, whereas some implementations may also provide SQL-like query capability.

NoSQL Databases - Characteristics

Auto sharding and replication – To support horizontal scaling and provide high availability, a NoSQL storage device automatically employs sharding and replication techniques where the dataset is partitioned horizontally and then copied to multiple nodes.

Integrated caching – This removes the need for a third-party distributed caching layer, such as Memcached.

Distributed query support – NoSQL storage devices maintain consistent query behavior across multiple shards.

NoSQL Databases - Characteristics

Polyglot persistence – The use of NoSQL storage does not mandate retiring traditional RDBMSs. In fact, both can be used at the same time, thereby supporting polyglot persistence, which is an approach of persisting data using different types of storage technologies within the same solution architecture. This is good for developing systems requiring structured as well as semi/unstructured data.

NoSQL Databases - Characteristics

Aggregate-focused – Unlike relational databases that are most effective with fully normalized data, NoSQL storage devices store de-normalized aggregated data (an entity containing merged, often nested, data for an object) thereby eliminating the need for joins and extensive mapping between application objects and the data stored in the database. One exception, however, is that graph database storage devices (introduced shortly) are not aggregate-focused.

NoSQL Databases

Volume

•The storage requirement of ever increasing data volumes commands the use of databases that are highly scalable while keeping costs down for the business to remain competitive. NoSQL storage devices fulfill this requirement by providing scale out capability while using inexpensive commodity servers.

Velocity

•The fast influx of data requires databases with fast access data write capability. NoSQL storage devices enable fast writes by using schema-on-read rather than schema-on-write principle. Being highly available, NoSQL storage devices can ensure that write latency does not occur because of node or network failure.

NoSQL Databases

Variety

- A storage device needs to handle different data formats including documents, emails, images and videos and incomplete data.
- NoSQL storage devices can store these different forms of semi-structured and unstructured data formats.
- NoSQL databases support *schema evolution*.

NoSQL Databases

Types

•NoSQL storage devices can mainly be divided into four types based on the way they store data.

- key-value
- document
- column-family
- graph

NoSQL Databases

key	value
631	John Smith, 10.0.30.25, Good customer service
365	1001010111011011111011101010110101010011100110 10
198	<CustomerId>32195</CustomerId><Total>43.25</Total>

An example of key-value NoSQL storage.

Key-Value

- Key-value storage devices store data as key-value pairs and act like hash tables. The table is a list of values where each value is identified by a key
- Value look-up can only be performed via the keys as the database is oblivious to the details of the stored aggregate.
- Partial updates are not possible. An update is either a delete or an insert operation.
- Key-value storage devices generally do not maintain any indexes, therefore writes are quite fast. Based on a simple storage model, key-value storage devices are highly scalable.

Key-Value

- To provide some structure to the stored data, most key-value storage devices provide collections or buckets (like tables) into which key-value pairs can be organized.
- A single collection can hold multiple data formats

Document

Document storage devices also store data as key-value pairs.

However, unlike key-value storage devices, the stored value is a document that can be queried by the database.

These documents can have a complex nested structure, such as an invoice

The documents can be encoded using either a text-based encoding scheme, such as XML or JSON, or using a binary encoding scheme, such as BSON (Binary JSON).

Like key-value storage devices, most document storage devices provide collections or buckets (like tables) into which key-value pairs can be organized

A document storage device is appropriate when:

- storing semi-structured document-oriented data comprising flat or nested schema
- schema evolution is a requirement as the structure of the document is either unknown or is likely to change
- applications require a partial update of the aggregate stored as a document
- searches need to be performed on different fields of the documents
- storing domain objects, such as customers, in serialized object form
- query patterns involve insert, select, update and delete operations

•A document storage device is inappropriate when:

- multiple documents need to be updated as part of a single transaction
- performing operations that need joins between multiple documents or storing data that is normalized
- schema enforcement for achieving consistent query design is required as the document structure may change between successive query runs, which will require restructuring the query
- the stored value is not self-describing and does not have a reference to a schema
- binary data needs to be stored

•Examples of document storage devices include MongoDB, CouchDB, and Terrastore.

Column-Family

studentId	personal details	address	modules history
821	FirstName: Cristie LastName: Augustin DoB: 03-15-1992 Gender: Female Ethnicity: French	Street: 123 New Ave City: Portland State: Oregon ZipCode: 12345 Country: USA	Taken: 5 Passed: 4 Failed: 1
742	FirstName: Carlos LastName: Rodriguez MiddleName: Jose Gender: Male	Street: 456 Old Ave City: Los Angeles Country: USA	Taken: 7 Passed: 5 Failed: 2

An example of column-family NoSQL storage

Column-Family

- Column-family storage devices store data much like a traditional RDBMS but group related columns together in a row, resulting in column-families
- Each column can be a collection of related columns itself, referred to as a super-column.
- Each super-column can contain an arbitrary number of related columns that are generally retrieved or updated as a single unit

Column-Family

- Each super-column can contain an arbitrary number of related columns that are generally retrieved or updated as a single unit.
- Each row consists of multiple column-families and can have a different set of columns, thereby manifesting flexible schema support.
- Each row is identified by a row key.

Column-Family

- Column-family storage devices provide fast data access with random read/write capability.
- They store different column-families in separate physical files, which improves query responsiveness as only the required column-families are searched.
- Some column-family storage devices provide support for selectively compressing column families.
- Leaving searchable column-families uncompressed can make queries faster because the target column does not need to be decompressed for lookup.
- Most implementations support data versioning while some support specifying an expiry time for column data. When the expiry time has passed, the data is automatically removed.

•A column-family storage device is appropriate when:

- realtime random read/write capability is needed and data being stored has some defined structure
- data represents a tabular structure, each row consists of a large number of columns and nested groups of interrelated data exist
- support for schema evolution is required as column families can be added or removed without any system downtime
- certain fields are mostly accessed together, and searches need to be performed using field values

A column-family storage device is appropriate when:

- efficient use of storage is required when the data consists of sparsely populated rows .

since column-family databases only allocate storage space if a column exists for a row. If no column is present, no space is allocated.

query patterns involve insert, select, update and delete operations

A column-family storage device is inappropriate when:

•relational data access is required; for example, joins

• ACID transactional support is required

• binary data needs to be stored

• SQL-compliant queries need to be executed

• query patterns are likely to change frequently because that could initiate a corresponding restructuring of how column-families are arranged

•Examples of column-family storage devices include Cassandra, HBase and Amazon SimpleDB.

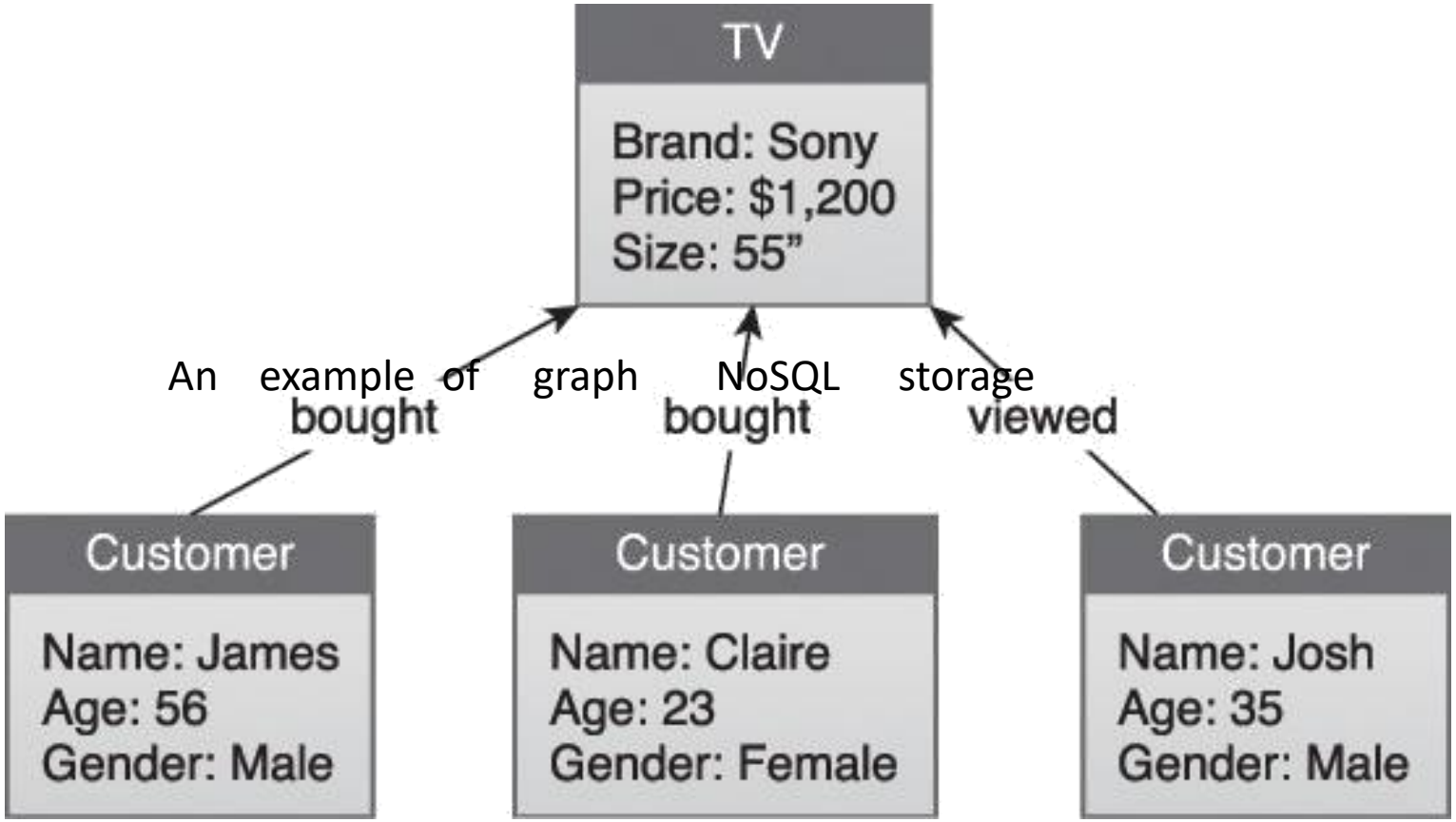
Graph

•Graph storage devices are used to persist inter-connected entities. Unlike other NoSQL storage devices, where the emphasis is on the structure of the entities, graph storage devices place emphasis on storing the linkages between entities

•Entities are stored as nodes (not to be confused with *cluster nodes) and are also called* vertices, while the linkages between entities are stored as edges.

•Each node can be thought of a single row while the edge denotes a join.

Graph



An example of graph NoSQL storage.

Graph

•Nodes can have more than one type of link between them through multiple edges. Each node can have attribute data as key-value pairs, such as a customer node with ID, name and age attributes.

•Each edge can have its own attribute data as key-value pairs, which can be used to further filter query results.

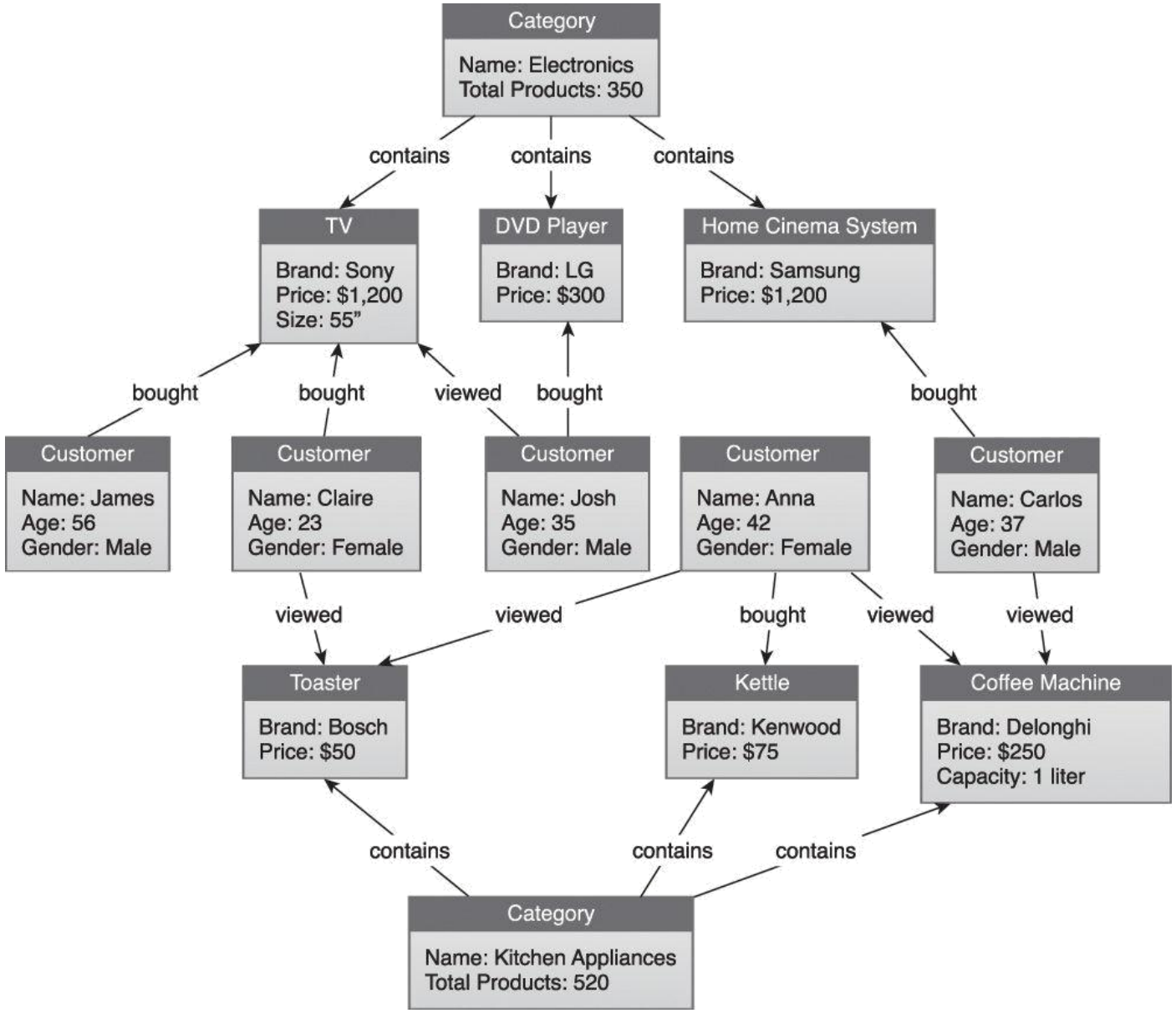
•Having multiple edges are similar to defining multiple foreign keys in

•an RDBMS; however, not every node is required to have the same edges.

Graph

- Queries generally involve finding interconnected nodes based on node attributes and/or edge attributes, commonly referred to as node traversal.
- Edges can be unidirectional or bidirectional, setting the node traversal direction.
- Generally, graph storage devices provide consistency via ACID compliance.

Graph



Graph storage devices store entities and their relationships

Graph

- The degree of usefulness of a graph storage device depends on the number and types of edges defined between the nodes.
- The greater the number and more diverse the edges are, the more diverse the types of queries it can handle
- Graph storage devices generally allow adding new types of nodes without making changes to the database.
- This also enables defining additional links between nodes as new types of relationships or nodes appear in the database.

A graph storage device is appropriate when:

- interconnected entities need to be stored
- querying entities based on the type of relationship with each other rather than the attributes of the entities
- finding groups of interconnected entities
- finding distances between entities in terms of the node traversal distance
- mining data with a view toward finding patterns

- A graph storage device is inappropriate when:

- updates are required to a large number of node attributes or edge attributes, which is a costly operation compared to performing node traversals
- entities have a large number of attributes or nested data—it is best to store lightweight entities in a graph storage device while storing the rest of the attribute data in a separate non-graph NoSQL storage device
- binary storage is required
- queries based on the selection of node/edge attributes dominate node traversal queries. Examples include Neo4J, Infinite Graph and Orient DB.

NewSQL Databases

- NewSQL storage devices combine the ACID properties of RDBMS with the scalability and fault tolerance offered by NoSQL storage devices.
- SQL compliant syntax for data definition and data manipulation operations,
- use a logical relational data model for data storage.
- Used for developing OLTP systems with very high volumes of transactions
- used for realtime analytics,
- provides an easier transition from a traditional RDBMS to a highly scalable database due to its support for SQL.
- Examples of NewSQL databases include VoltDB, NuoDB and InnoDB.

In-Memory Storage Devices

An in-memory storage device generally utilizes RAM, the main memory of a computer, as its storage medium to provide fast data access. The growing capacity and decreasing cost of RAM, coupled with the increasing read/write speed of solid state hard drives, has made it possible to develop in-memory data storage solutions.

Storage of data in memory eliminates the latency of disk I/O and the data transfer time between the main memory and the hard drive.

capacity can be increased by horizontally scaling the cluster

Cluster-based memory enables storage of large amounts of data, including Big Data datasets

In-Memory Storage Devices

- in-memory storage is approximately 80 times faster than on-disk storage
- enables in-memory analytics
- A Big Data in-memory storage device is implemented over a cluster, providing high availability and redundancy. Therefore
- horizontal scalability can be achieved by adding more nodes or memory
- in-memory storage devices are expensive. Consequently, only
- up-to-date and fresh data or data that has the most value is kept in memory, whereas stale data gets replaced with newer, fresher data

An in-memory storage device is appropriate when:

- data arrives at a fast pace and requires realtime analytics or event stream processing
- continuous or always-on analytics is required, such as operational BI and operational analytics
- interactive query processing and realtime data visualization needs to be performed, including what-if analysis and drill-down operations
- the same dataset is required by multiple data processing jobs
- performing exploratory data analysis, as the same dataset does not need to be reloaded from disk if the algorithm changes
- data processing involves iterative access to the same dataset graph-based algorithms
- developing low latency Big Data solutions with ACID transaction support

An in-memory storage device is inappropriate when:

- data processing consists of batch processing
- very large amounts of data need to be persisted in-memory for a long time in order to perform in-depth data analysis
- performing strategic BI or strategic analytics that involves access to very large amounts of data and involves batch data processing
- datasets are extremely large and do not fit into the available memory
- making the transition from traditional data analysis toward Big Data analysis, as incorporating an in-memory storage device may require additional skills and involves a complex setup
- an enterprise has a limited budget

In-memory storage devices can be implemented as:

- In-Memory Data Grid (IMDG)
- In-Memory Database (IMDB)

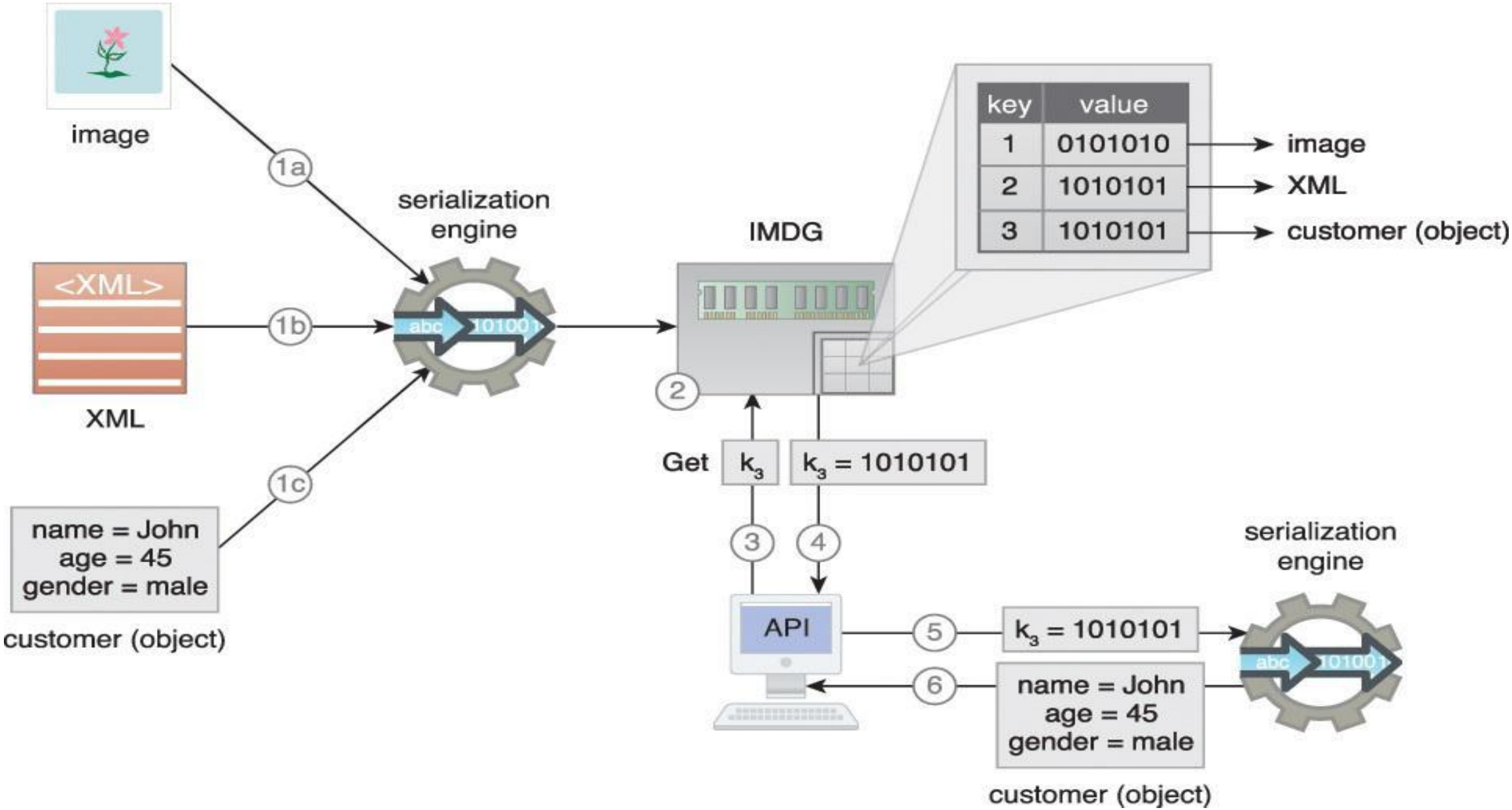
what makes them distinct is the way data is stored in the memory. Key features of each of these technologies are discussed next.

In-Memory Data Grids

IMDGs store data in memory as key-value pairs across multiple nodes where the keys and values can be any business object or application data in serialized form.

This supports schema-less data storage through storage of semi/unstructured data. Data access is typically provided via APIs

An IMDG storage device



In-Memory Data Grids

1. An image (a), XML data (b) and a customer object (c) are first serialized using a serialization engine.
2. They are then stored as key-value pairs in an IMDG.
3. A client requests the customer object via its key.
4. The value is then returned by the IMDG in serialized form.
5. The client then utilizes a serialization engine to deserialize the value to obtain the customer object…
6. … in order to manipulate the customer object

In-Memory Data Grids

Nodes in IMDGs keep themselves synchronized and collectively provide high availability, fault tolerance and consistency.

Unlike NoSQL's, IMDGs support immediate consistency.

IMDGs provide faster data access as IMDGs store non-relational data as objects.

IMDGs scale horizontally by implementing data partitioning and data replication

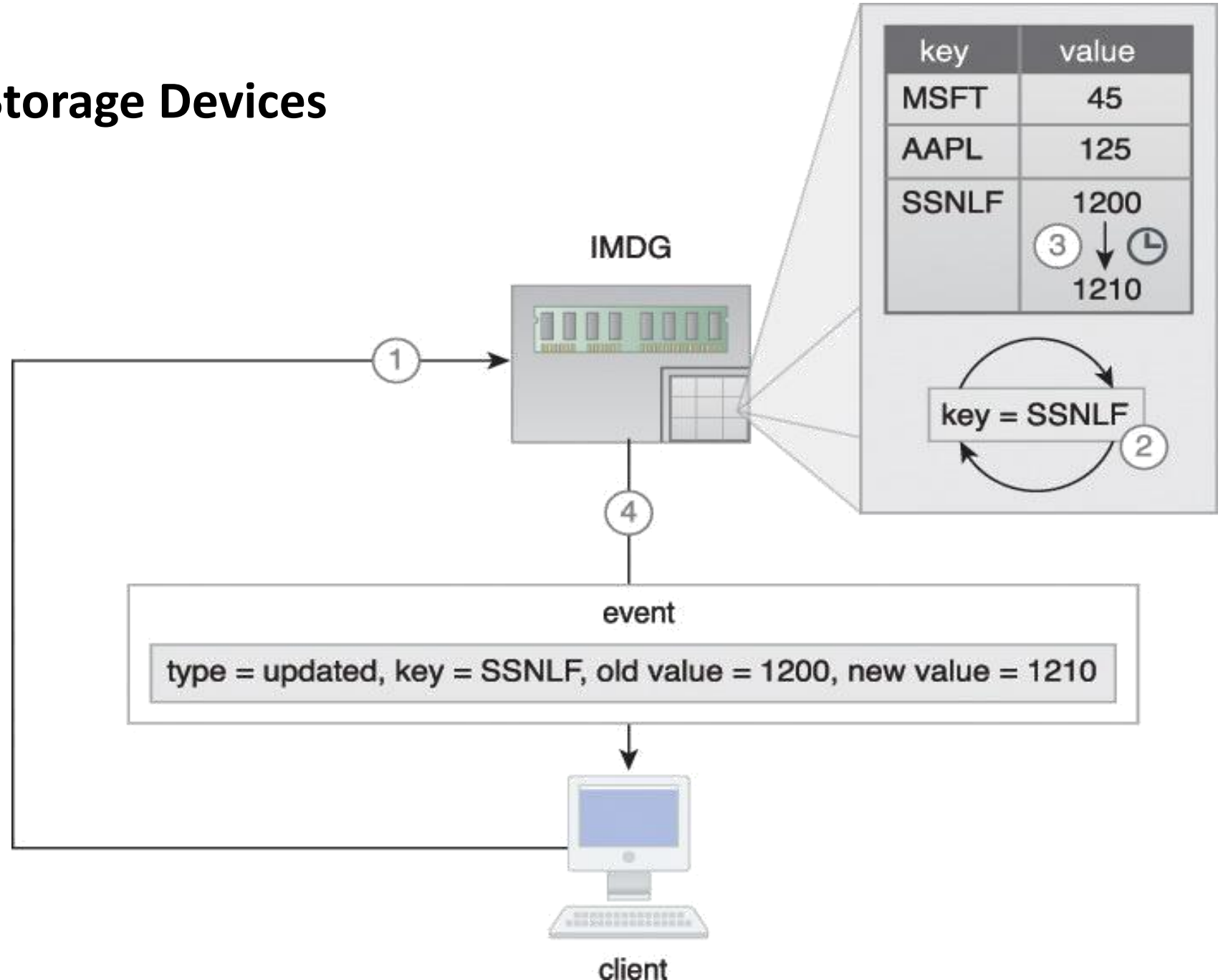
IMDGs automatically re-create lost copies of data from replicas as part of the recovery process.

In-Memory Data Grids

IMDGs are heavily used for realtime analytics because they support Complex Event Processing (CEP) This is achieved through a feature called *continuous querying*, also known as active querying, where a filter for event(s) of interest is registered with the IMDG. The IMDG then continuously evaluates the filter and whenever the filter is satisfied as a result of insert/update/delete operations, subscribing clients are informed pairs, such as *old* and *new* values.

However, unlike a distributed cache, an IMDG provides built in support for replication and high availability

In-Memory Storage Devices



In-Memory Storage Devices

An IMDG can also be deployed within a cloud based environment where it provides a flexible storage medium that can scale out or scale in automatically as the storage demand increases or decreases

In-Memory Storage Devices

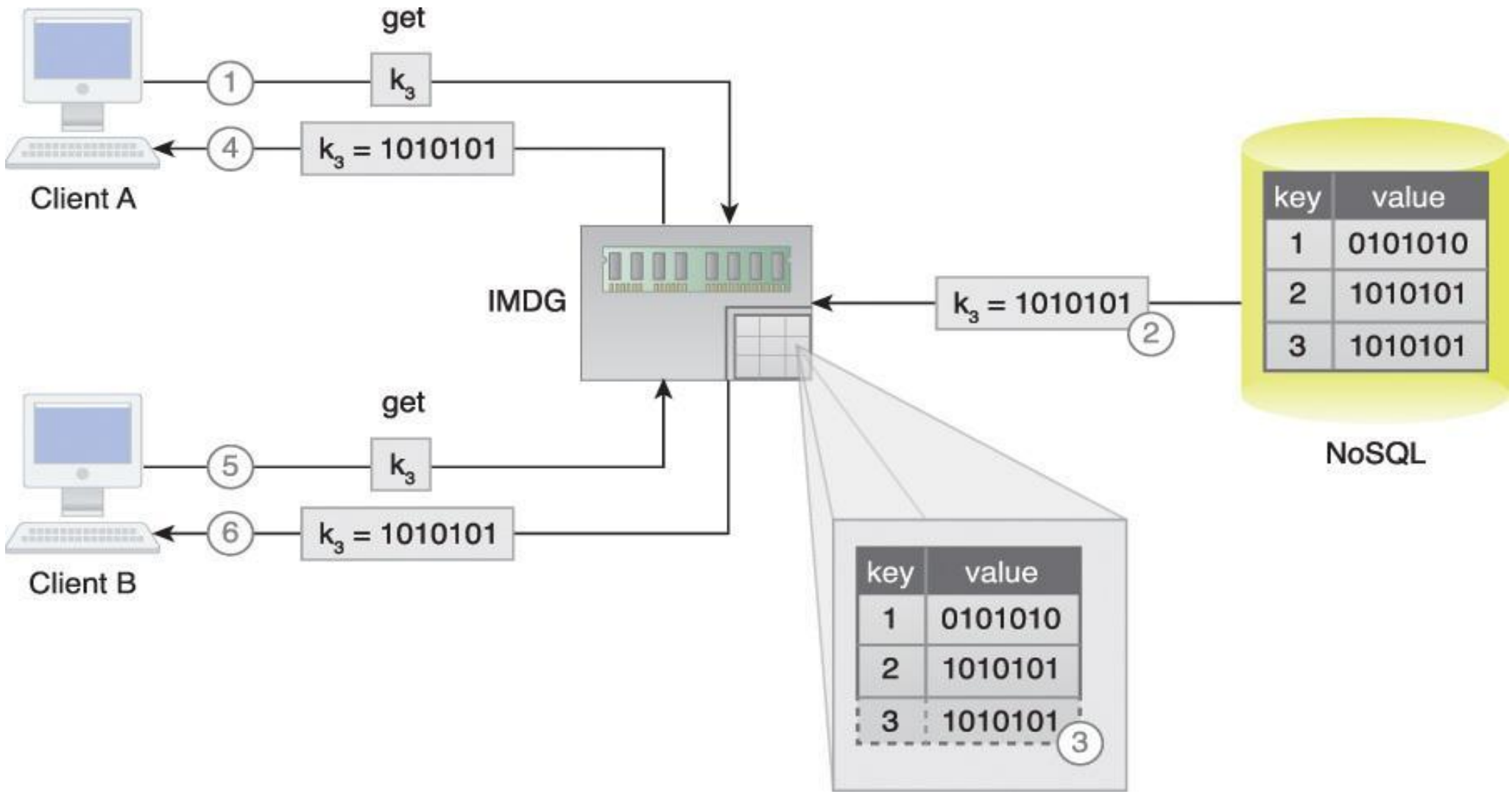
In a Big Data solution environment, IMDGs are often deployed together with on-disk storage devices that act as the backend storage. This is achieved via the following approaches that can be combined as necessary to support read/write performance, consistency and simplicity requirements:

- read-through
- write-through
- write-behind
- refresh-ahead

Read-through

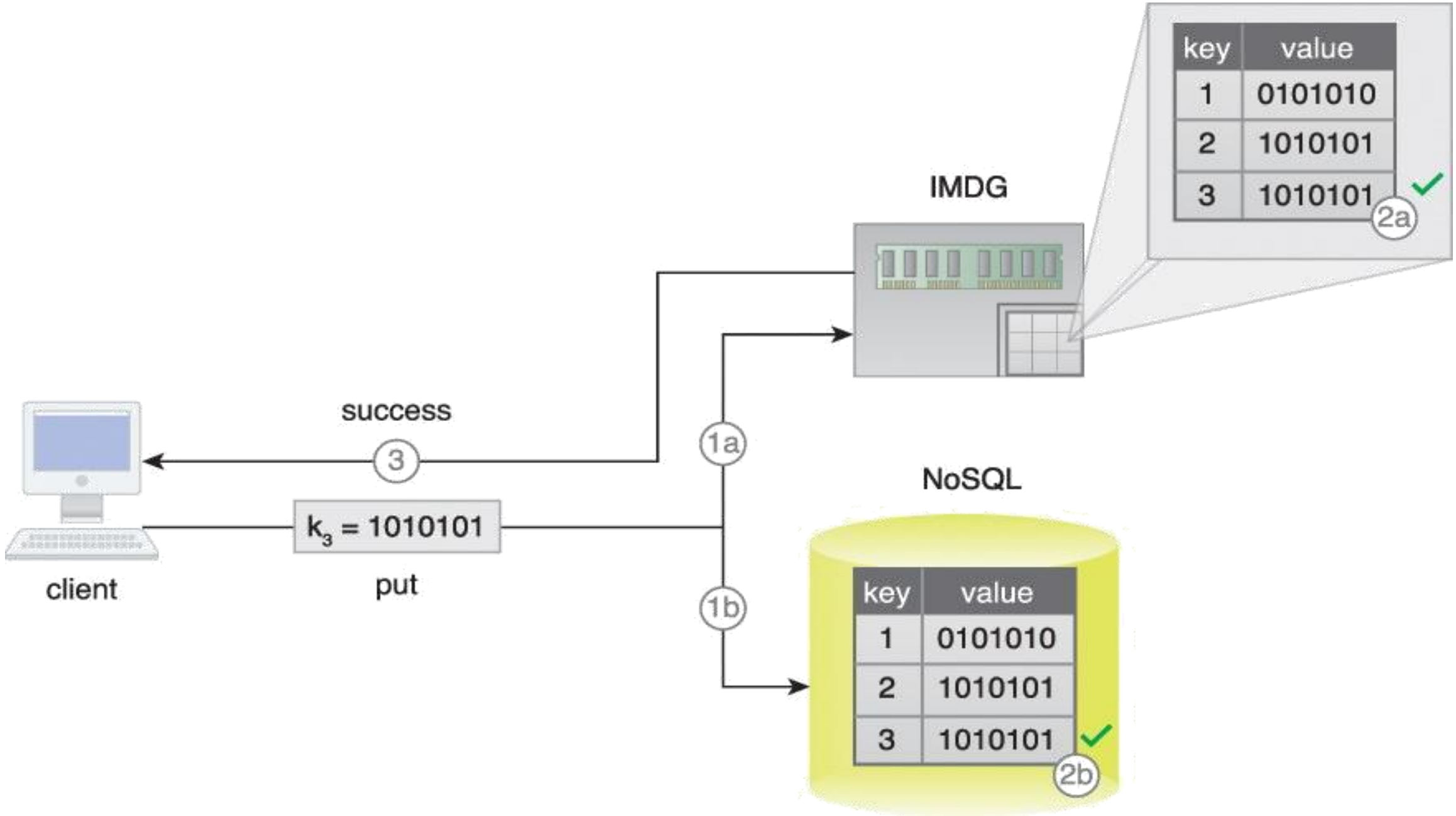
If a requested value for a key is not found in the IMDG, then it is synchronously read from the backend on-disk storage device, such as a database. Upon a successful read from the backend on-disk storage device, the key-value pair is inserted into the IMDG, and the

requested value is returned to the client. Any subsequent requests for the same key are then served by the IMDG directly, instead of the backend storage. Although it is a simple approach, its synchronous nature may introduce read latency.


get
1
k_3
4
k_3 = 1010101
Client A
IMDG
k_3 = 1010101
2
key
value
1
0101010
2
1010101
3
1010101
NoSQL
get
5
k_3
6
k_3 = 1010101
Client B
key
value
1
0101010
2
1010101
3
1010101
3

Write-through

Any write (insert/update/delete) to the IMDG is written synchronously in a transactional manner to the backend on-disk storage device, such as a database. If the write to the backend on-disk storage device fails, the IMDG's update is rolled back. Due to this transactional nature, data consistency is achieved immediately between the two data stores. However, this transactional support is provided at the expense of write latency as any write operation is considered complete only when feedback (write success/failure) from the backend storage is received

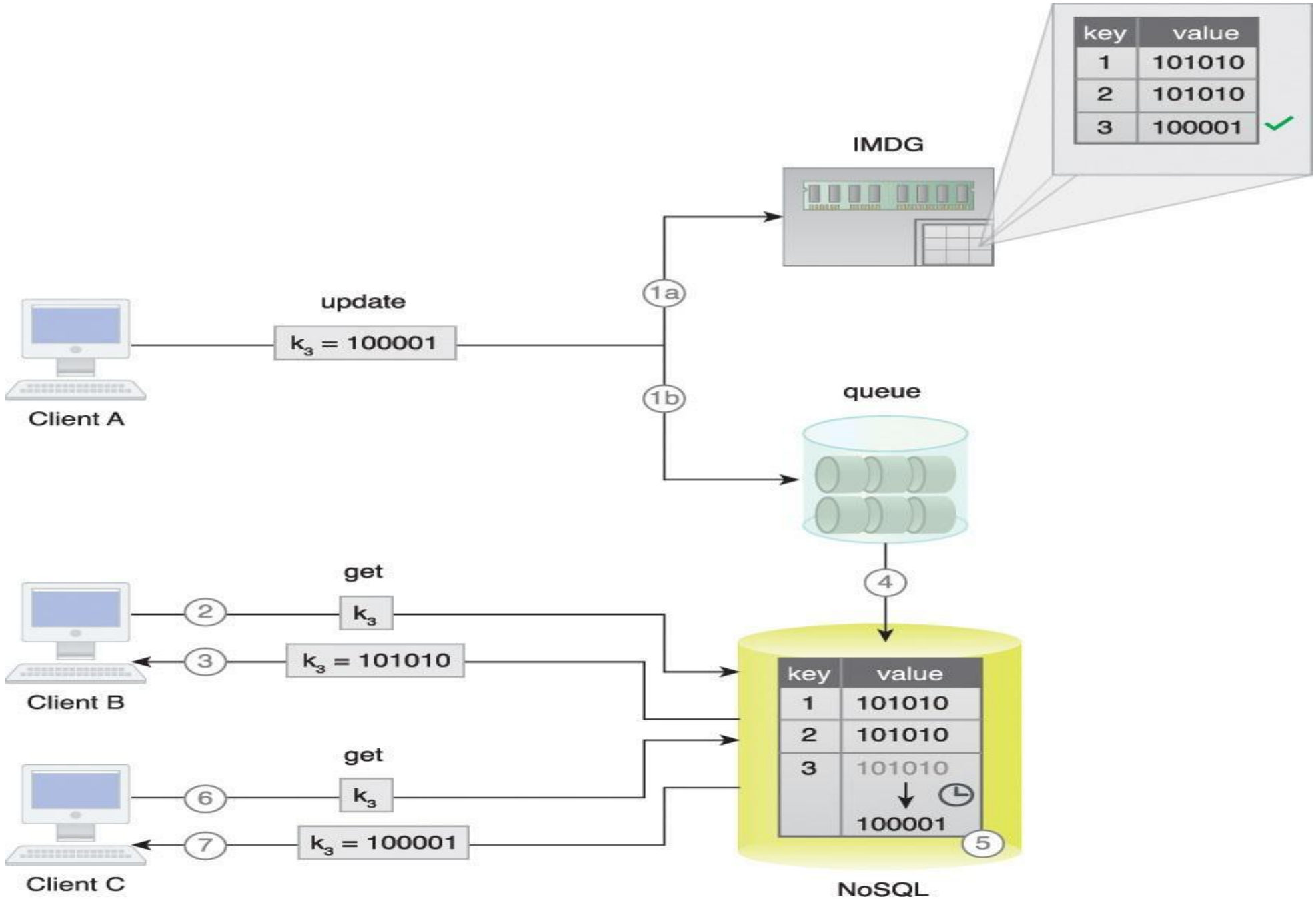

key
value
1
0101010
2
1010101
3
1010101
2a
IMDG
success
3
1a
client
k_3 = 1010101
put
1b
NoSQL
key
value
1
0101010
2
1010101
3
1010101
2b

Write-behind

Any write to the IMDG is written asynchronously in a batch manner to the backend on-disk storage device.

A queue is generally placed between the IMDG and the backend storage for keeping track of the required changes to the backend storage.

The asynchronous nature increases both write performance (the write operation is considered completed as soon as it is written to the IMDG) and read performance (data can be read from the IMDG as soon as it is written to the IMDG) and scalability/availability in general. However, the asynchronous nature introduces inconsistency until the backend storage is updated at the specified interval

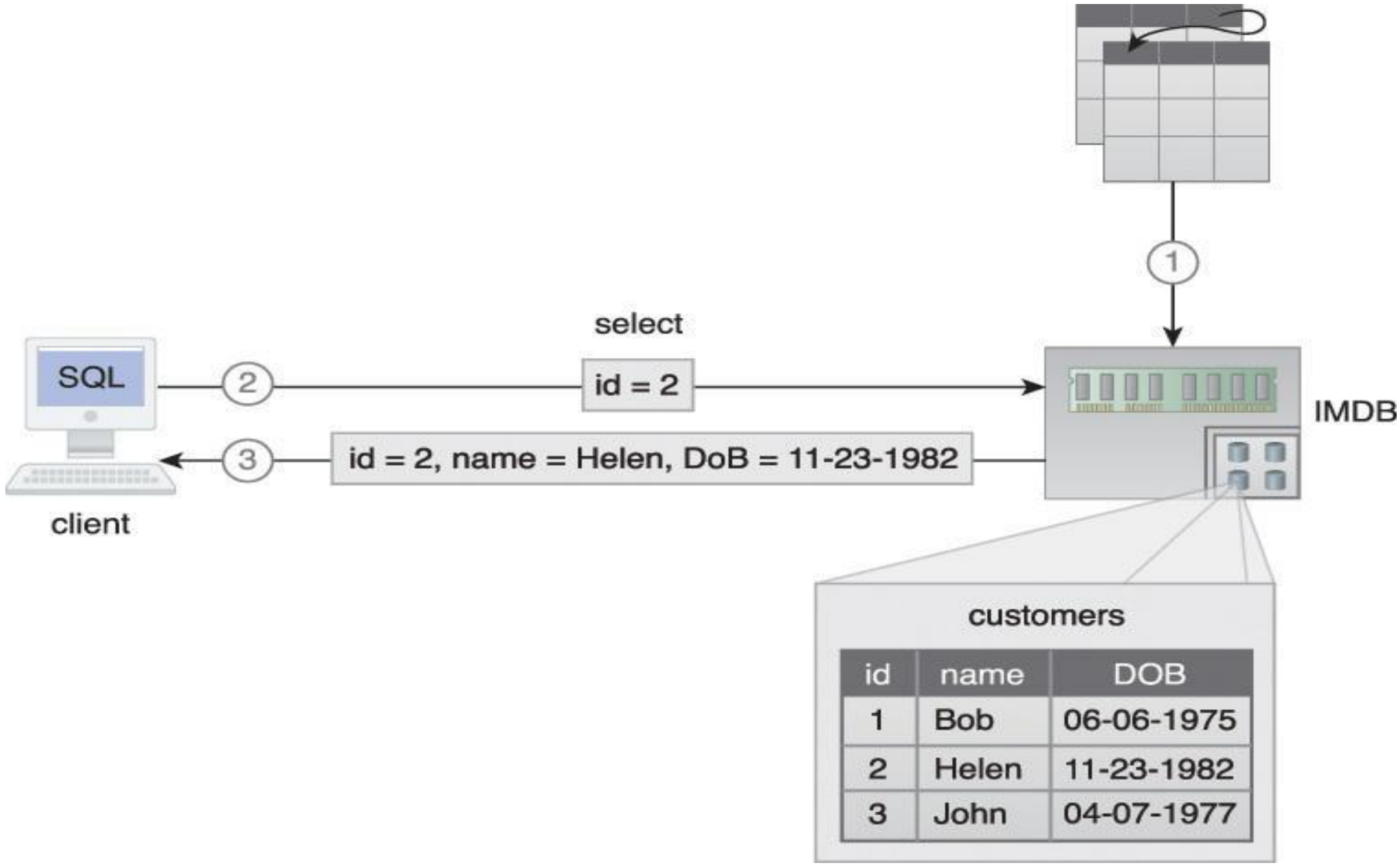

key
value
1
101010
2
101010
3
100001
IMDG
update
k_3 = 100001
1a
1b
Client A
queue
get
2
k_3
3
k_3 = 101010
Client B
4
key
value
1
101010
2
101010
3
101010
100001
5
get
6
k_3
7
k_3 = 100001
Client C
NoSQL

Refresh-ahead

Refresh-ahead is a proactive approach where any frequently accessed values are automatically, asynchronously refreshed in the IMDG, provided that the value is accessed before its expiry time as configured in the IMDG.

If a value is accessed after its expiry time, the value, like in the read-through approach, is synchronously read from the backend storage and updated in the IMDG before being returned to the client.

Compared to the read-through approach, where a value is served from the IMDG until its expiry, data inconsistency between the IMDG and the backend storage is minimized as values are refreshed before they expire


1
select
SQL
2
id = 2
IMDB
3
id = 2, name = Helen, DoB = 11-23-1982
client
customers
id
name
DOB
1
Bob
06-06-1975
2
Helen
11-23-1982
3
John
04-07-1977

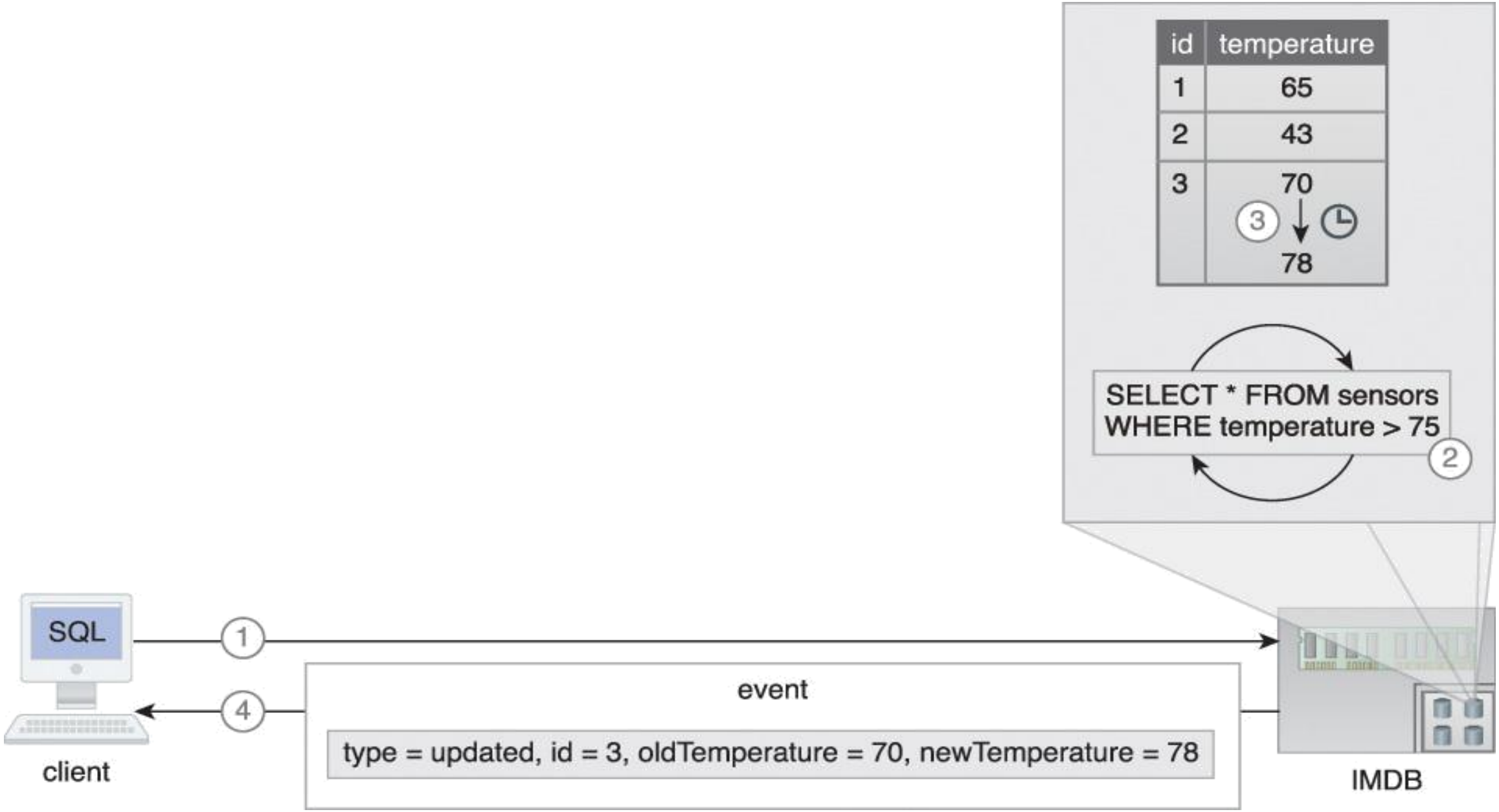
In-Memory Databases

IMDBs are in-memory storage devices that employ database technology and leverage the performance of RAM to overcome runtime latency issues that plague on-disk storage devices.

1. A relational dataset is stored into an IMDB.

2. A client requests a customer record (id = 2) via SQL.

3. The relevant customer record is then returned by the IMDB, which is directly manipulated by the client without the need for any deserialization.


id
temperature
1
65
2
43
3
70
3
78
SELECT * FROM sensors
WHERE temperature > 75
2
SQL
1
4
client
event
type = updated, id = 3, oldTemperature = 70, newTemperature = 78
IMDB

In-Memory Databases

An IMDB can be relational in nature (relational IMDB) or may leverage NoSQL technology relational IMDBs make use of the more familiar SQL language which helps data analysts or data scientists that do not have advanced programming skills.

NoSQL-based IMDBs generally provide API-based access, which may be as simple as put, get and delete operations.

Depending on the underlying implementation, some IMDBs scale-out, while others scale-up, to achieve scalability.

Not all IMDB implementations directly support durability, but instead leverage various strategies for providing durability in the face of machine failures or memory corruption.

These strategies include the following:

- Use of Non-volatile RAM (NVRAM) for storing data permanently.
- Database transaction logs can be periodically stored to a non-volatile medium, such as disk.
- Snapshot files, which capture database state at a certain point in time, are saved to disk.
- An IMDB may leverage sharding and replication to support increasing availability and reliability as a substitute for durability.
- IMDBs can be used in conjunction with on-disk storage devices such as NoSQL databases and RDBMSs for durability

Like an IMDG, an IMDB may also support the continuous query feature The IMDB then continuously executes the query in an iterative manner

IMDBs are heavily used in realtime analytics and can further be used for developing low latency applications

requiring full ACID transaction support Introduction of IMDBs into an existing Big Data solution on generally requires replacement of existing on-disk storage devices, including any RDBMSs if used.

In the case of replacing an RDBMS with a relational IMDB, little or no application code change is required due to SQL support provided by the relational IMDB.

However, when replacing an RDBMS with a NoSQL IMDB, code change may be required due to the need to implement the IMDB's NoSQL APIs.

Relational IMDBs are generally less scalable than IMDGs, as relational IMDBs need to support distributed queries and transactions across the cluster. Some IMDB implementations may benefit from scaling up, which helps to address the latency that occurs when executing queries and transactions in a scale-out environment.

Examples include Aerospike, MemSQL, Altibase HDB, eXtreme DB and Pivotal GemFire XD.

An IMDB storage device is appropriate when

- relational data needs to be stored in memory with ACID support
- adding realtime support to an existing Big Data solution currently using on-disk storage
- the existing on-disk storage device can be replaced with an in-memory equivalent technology
- it is required to minimize changes to the data access layer of the application code, such as when the application consists of an SQL-based data access layer
- relational storage is more important than scalability