Module 7: Fundamental Big Data Engineering

Introduction

This is the official workbook for the Big Data Science Certified Professional Course **Module 7: Fundamental Big Data Engineering** and the corresponding Pearson VUE **Exam B90.07**.

The purpose of this document is to establish an understanding of fundamental Big Data engineering, which includes but is not limited to:

- Big Data Storage Terminology and Concepts
- Big Data Storage Device Characteristics
- On-disk Storage Devices
- Big Data Processing Engine Characteristics
- Fundamental Big Data Processing
- Introducing the MapReduce Processing Engine
- Fundamental MapReduce Algorithm Design

Data Engineering

Data engineering is the field of developing, testing, deploying and maintaining data processing solutions via collecting, parsing, transforming, joining, processing and managing data.

Two main activities that comprise data engineering include storage and processing of data, which is typically structured. A data engineer is tasked with making data amenable to various types of data analyses, including model development (data mining and other business-process specific algorithms) and reporting.

Big Data Engineering

Within the realm of Big Data, data engineering involves developing highly distributed, scalable, fault-tolerant data processing solutions to process large amounts of data in order to garner insights. Big Data engineering comprises data processing in support of the Big Data analysis lifecycle, as covered in Module 2.

Big Data engineers make data available for data scientists to develop models and data products. They are required to have knowledge of various data storage and processing technology alternatives for acquiring, storing and processing data that is often semi-structured and unstructured in nature.

Big Data Engineering Challenges & Characteristics

Many of the challenges faced in Big Data engineering are related to managing the three primary V's of Big Data:

- volume – internet-scale datasets and associated batch and realtime data processing

- velocity – processing large amounts of structured, unstructured, and semi-structured data arriving at a fast pace, including extraction of relevant data from semi-structured and unstructured datasets
- variety – collection and aggregation of data from multiple sources with disparate schemas or without any schema
- importing/exporting large amounts of data from/to traditional storage technologies, including OLTP (CRM, ERP, SCM systems) and OLAP systems (data warehouse)
- validating and cleansing data in realtime or batch mode and creating efficient data models
- establishing an optimal data storage and processing environment based on the type of data and its processing requirements
- developing efficient data processing algorithms that run over clusters of computers
- developing Big Data pipelines and Big Data applications that may include meaningful data visualizations

Big Data Engineering

The Big Data Engineer certification provides in-depth knowledge of the concepts and characteristics of the storage device and processing engine mechanisms first introduced in Module 2. Understanding these key mechanisms is essential to the Big Data Engineer as they are the core of any Big Data solution environment.

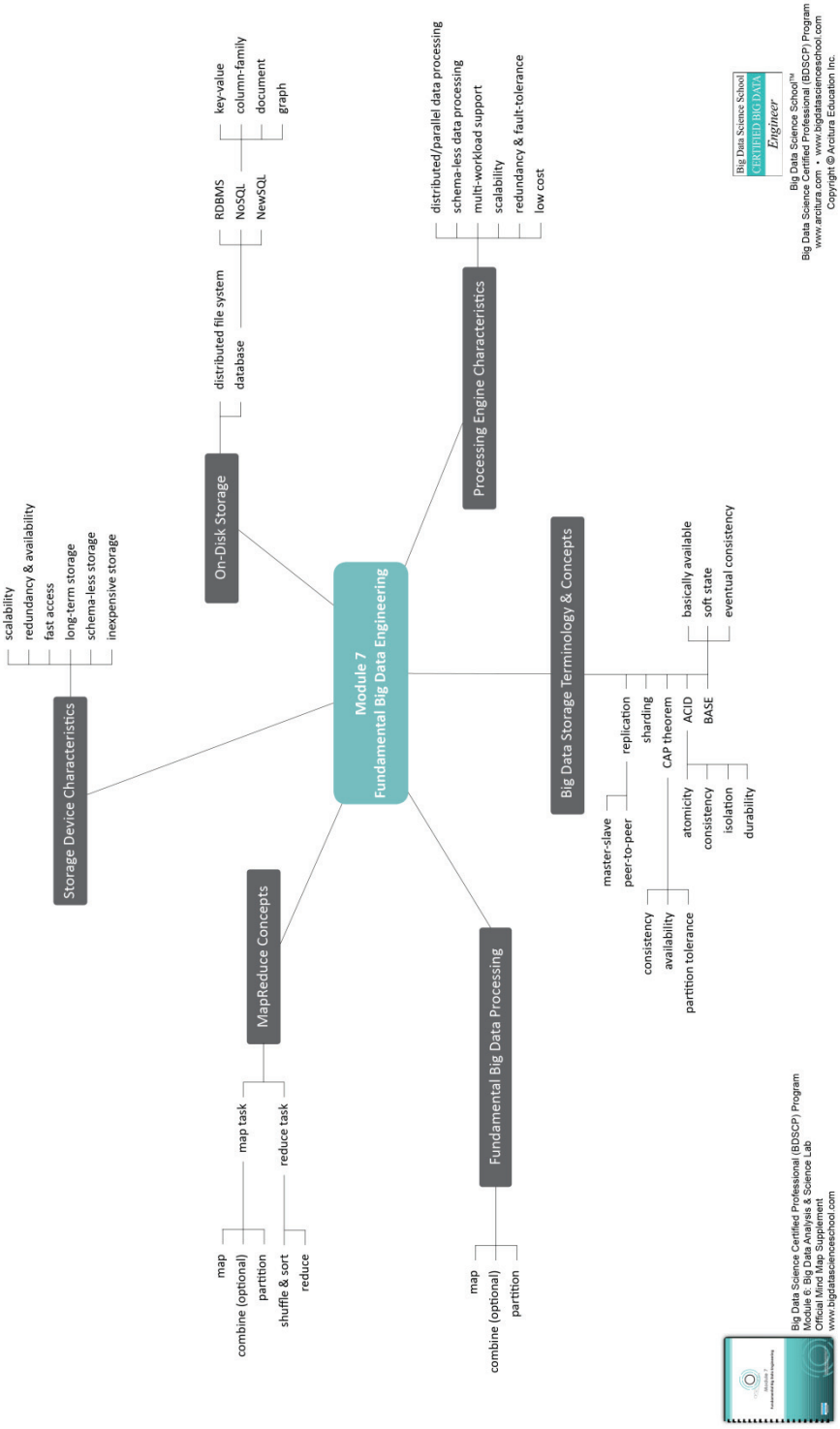
The objective of the entire set of topics, covered between Modules 7 and 8, is to enable Big Data engineers, consultants, and other related practitioners to design and build data processing solutions in a Big Data environment using contemporary technologies.

Specifically, Module 7 explores topics that support addressing the unique set of Big Data engineering challenges and characteristics. These topics include:

- the concepts, characteristics and on-disk technologies in support of batch processing of Big Data datasets
- an introduction to a data processing framework that supports batch processing of Big Data datasets
- the fundamental considerations for designing data processing algorithms that run over clusters of computers

Mind Map Poster

The *BDSCP Module 7: Mind Map Poster* that accompanies this course booklet provides an alternative visual representation of all primary topics covered in this course.



Big Data Storage Terminology and Concepts

The ability to store a variety of voluminous data arriving at a fast pace (volume, velocity, and variety characteristics of Big Data) is pivotal to generating any value out of Big Data datasets. Data can either be stored using disk-based or memory-based devices.

Generally, data needs to be stored to a disk before it can be processed. However, this only applies to batch processing mode. In realtime processing mode, data is first processed in memory and then stored to the disk.

The acquired data is generally not in a format or a structure that can be directly processed. Therefore, as a result of data wrangling activity (data cleansing, data filtering, data preparing), it needs to be stored again.

Data also needs to be stored once it gets processed, as a result of analytics and for archival purposes. Storage is generally required:

- when datasets are acquired or when data gets generated inside the enterprise boundary
- when data is manipulated for making it amenable to data analysis
- when data gets processed as a result of an ETL activity or output is generated as a result of an analytical operation

NOTE
Big Data datasets can neither be processed in batch more or realtime mode. Batch mode is covered in the upcoming *Introducing the MapReduce Processing Engine* section, while realtime mode is covered in Module 8.

Before exploring the Big Data storage device mechanism types, the following basic terminology and concepts will be discussed:

- sharding
- replication
- CAP theorem
- ACID
- BASE

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 (Figure 7.1). 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 database.

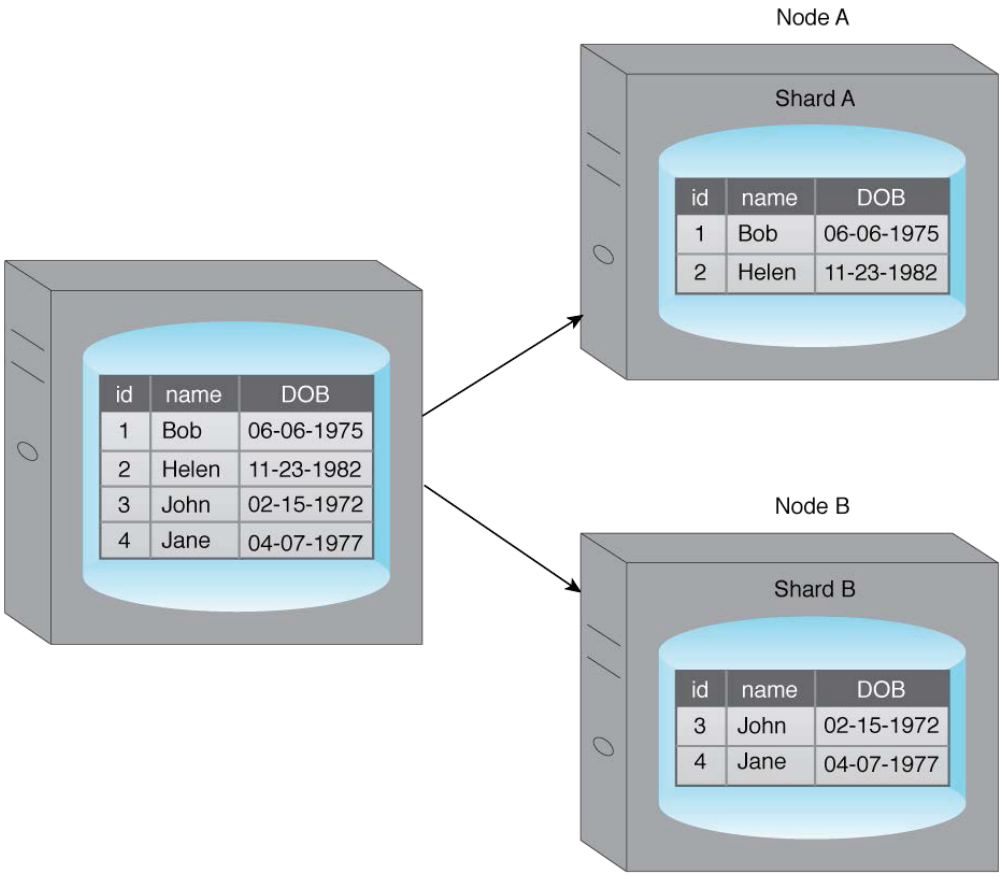


Figure 7.1 – An example of sharding where a dataset is spread across Node A and Node B, resulting in Shard A and Shard B respectively.

Sharding may or may not be transparent to the client. It distributes a processing load across multiple nodes to achieve horizontal scalability, which is supported as a method for increasing capacity by adding similar or higher capacity resources alongside existing resources.

Horizontal scaling is a method for increasing capacity by adding similar or higher capacity resources alongside existing resources. As each node is responsible for a part of the whole dataset, read/write times are greatly improved.

The following steps are shown in Figure 7.2:

1. a, b. 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 nodes.

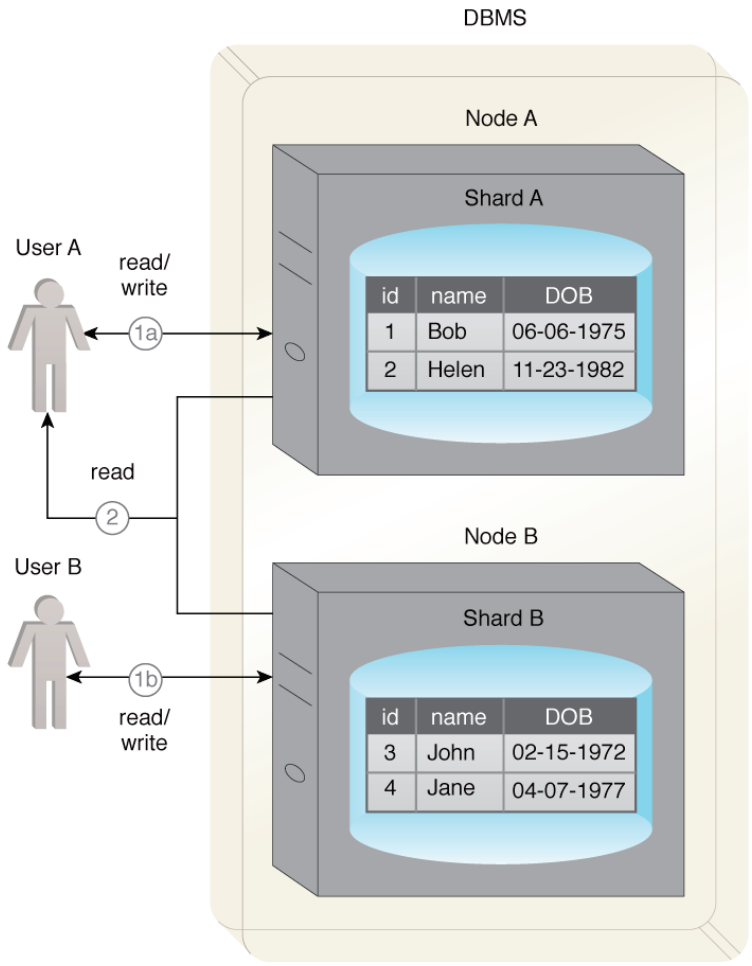


Figure 7.2 – An example of sharding where data is fetched from both Node A and Node B.

A benefit of sharding is that it provides partial tolerance towards 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 or keeping commonly accessed data collocated on a single shard helps to counter such performance issues.

Replication

Replication stores multiple copies of a dataset, known as *replicas*, on multiple nodes (Figure 7.3). This provides for scalability, availability, and fault tolerance. There are two different methods of implementing replication:

- master-slave
- peer-to-peer

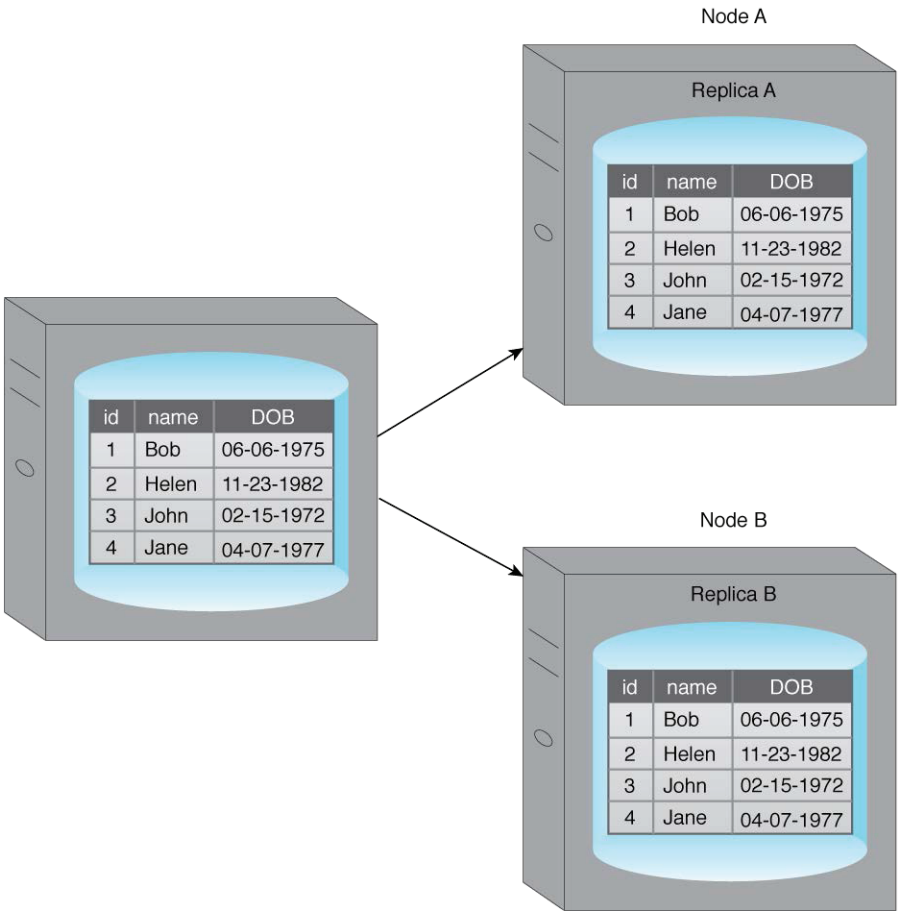


Figure 7.3 – An example of replication where a dataset is replicated across Node A and Node B, resulting in Replica A and Replica B.

Replication: Master-Slave

Nodes are arranged in a master-slave configuration where all data is inserted and updated via a master node. Once saved, the data is replicated over to multiple slave nodes. All external write requests (insert, update, delete) occur on the master node, whereas data can be read from any slave node. In Figure 7.4, writes are managed by the master node. Data can be read from either Slave A or Slave B nodes.

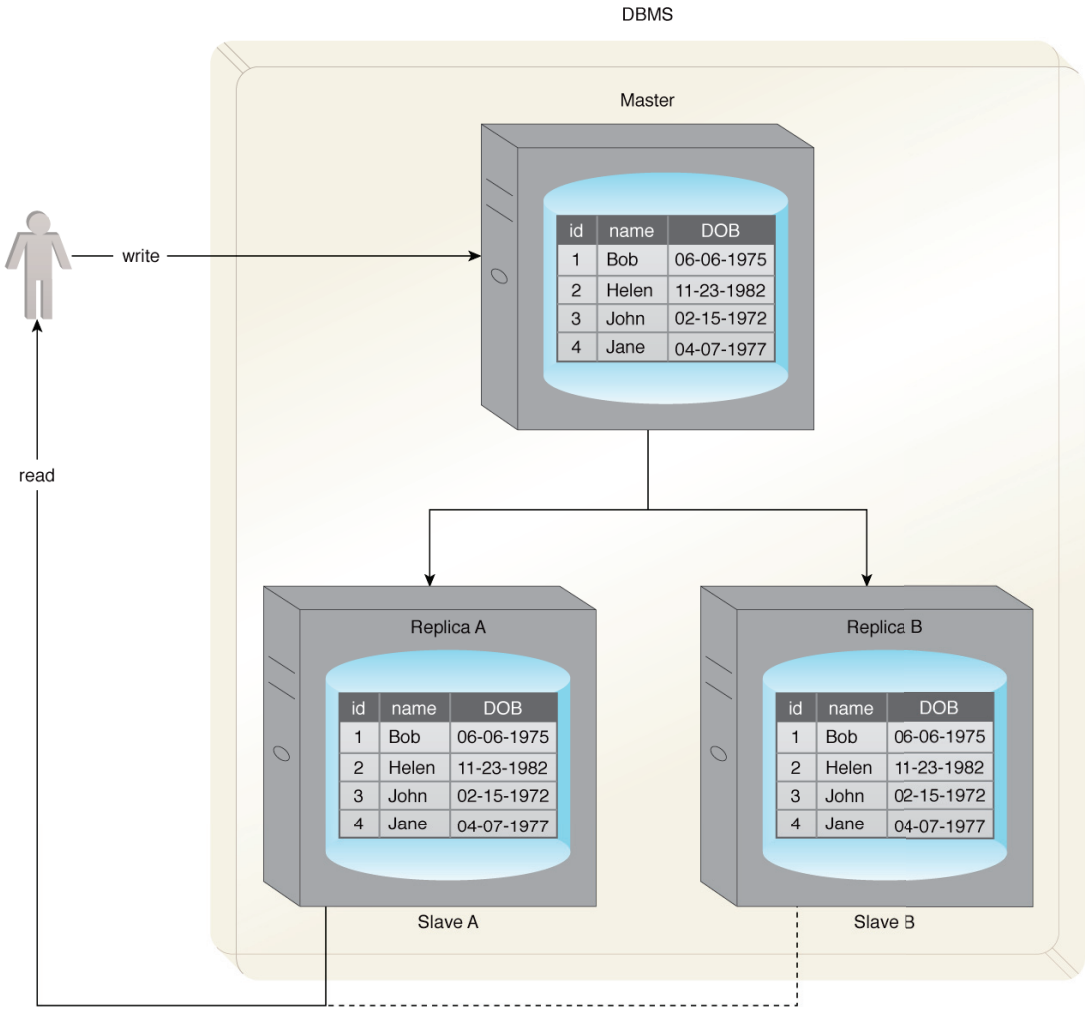


Figure 7.4 – An example of master-slave replication where Master A is the single point of contact for all writes, while data can be read from Slave A and Slave B.

Master-slave replication is ideal for read intensive loads rather than write intensive loads, as growing read demands can be fulfilled simply by horizontal scaling through additional slave nodes. Writes are consistent, as all writes are coordinated by the master node. This means that write performance suffers as the amount of writes increases. In case 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 case of master node failure, writes are not supported until a master node is reestablished. It is either resurrected from the backup master node, or a new master node is chosen from the slave nodes. Read inconsistency can be an issue if a slave node is read prior to the update being copied over to it from the master node.

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.

In Figure 7.5:

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 updated data from Slave B, resulting in an inconsistent read.
4. The data will eventually be made consistent on Slave B with an update from the Master.

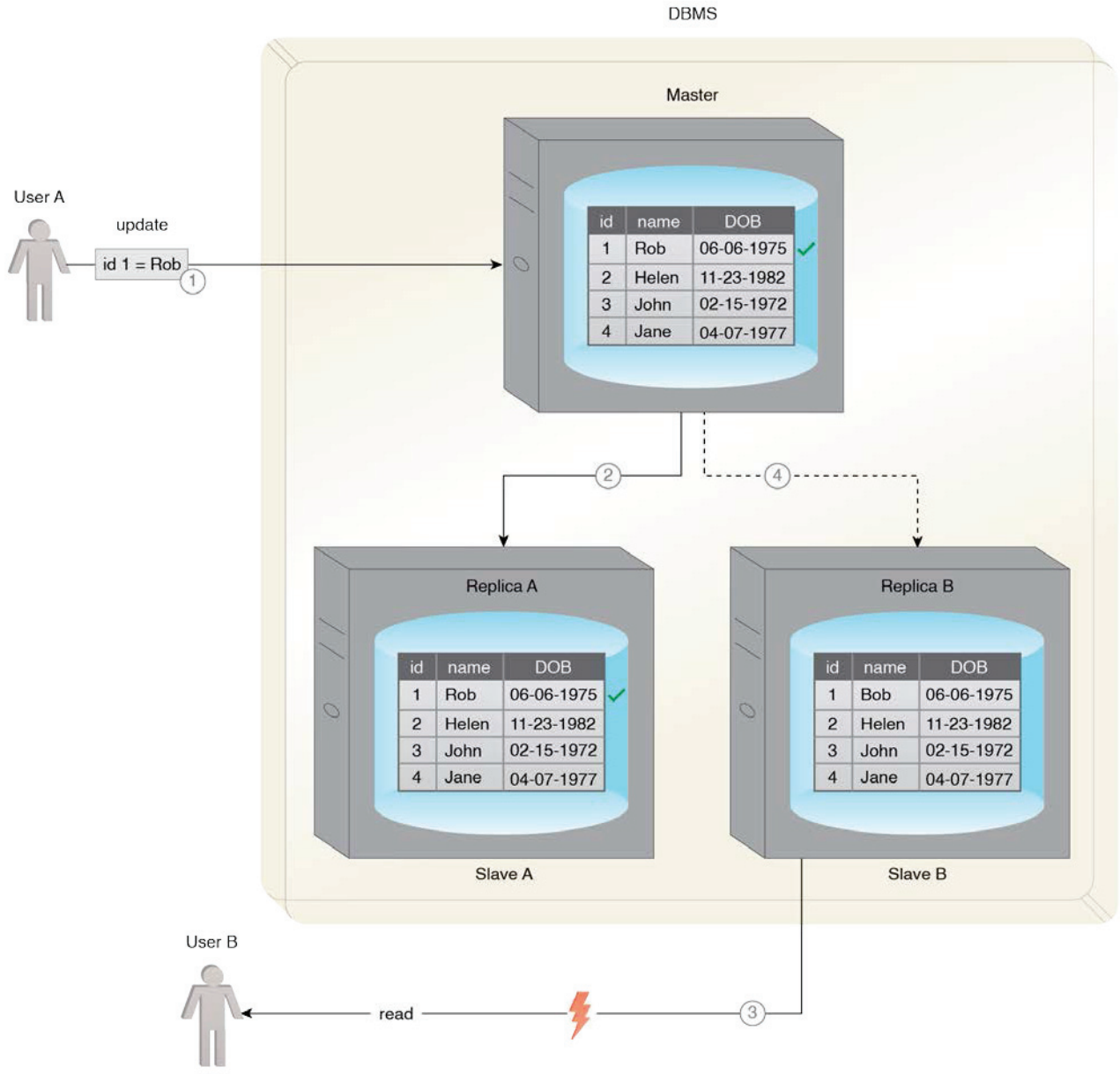


Figure 7.5 – An example of master-slave replication where read inconsistency occurs.

Replication: Peer-to-Peer

In peer-to-peer replication, all nodes operate at the same level. In other words, there is no master-slave configuration. Each node, known as a peer, is equally capable of handling reads and writes. Each write is copied to all peers, as illustrated in Figure 7.6.

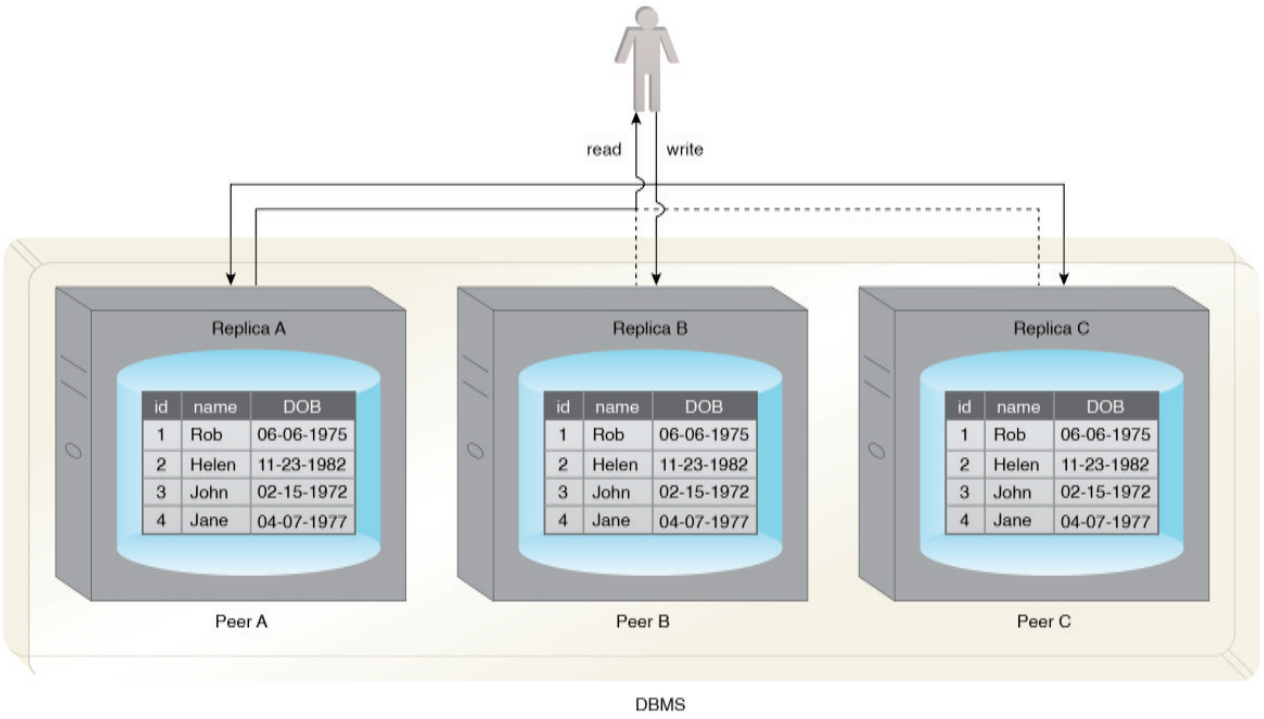


Figure 7.6 – Writes are copied to Peers A, B, and C simultaneously. Data is read from Peer A, but can also be read from Peers B or C.

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 pessimistic or optimistic concurrency.

- Pessimistic concurrency is a proactive approach that uses locking to ensure that only one update succeeds at a time. However, this affects availability as the database remains unavailable until all locks are released.
- Optimistic concurrency is a reactive approach that does not use locking. Instead, it allows the inconsistency to happen first and restores consistency after the fact.

Due to this, 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, reads can be inconsistent between the time period when some of the peers have been updated while the others are being updated. However, reads eventually become consistent when the updates have been copied over to all peers.

To ensure read consistency, a voting system can be implemented where a read is declared consistent if the majority of the peers contain the same version of the record. Implementation of such a voting system requires a reliable and fast communication mechanism between the peers.

In Figure 7.7:

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.

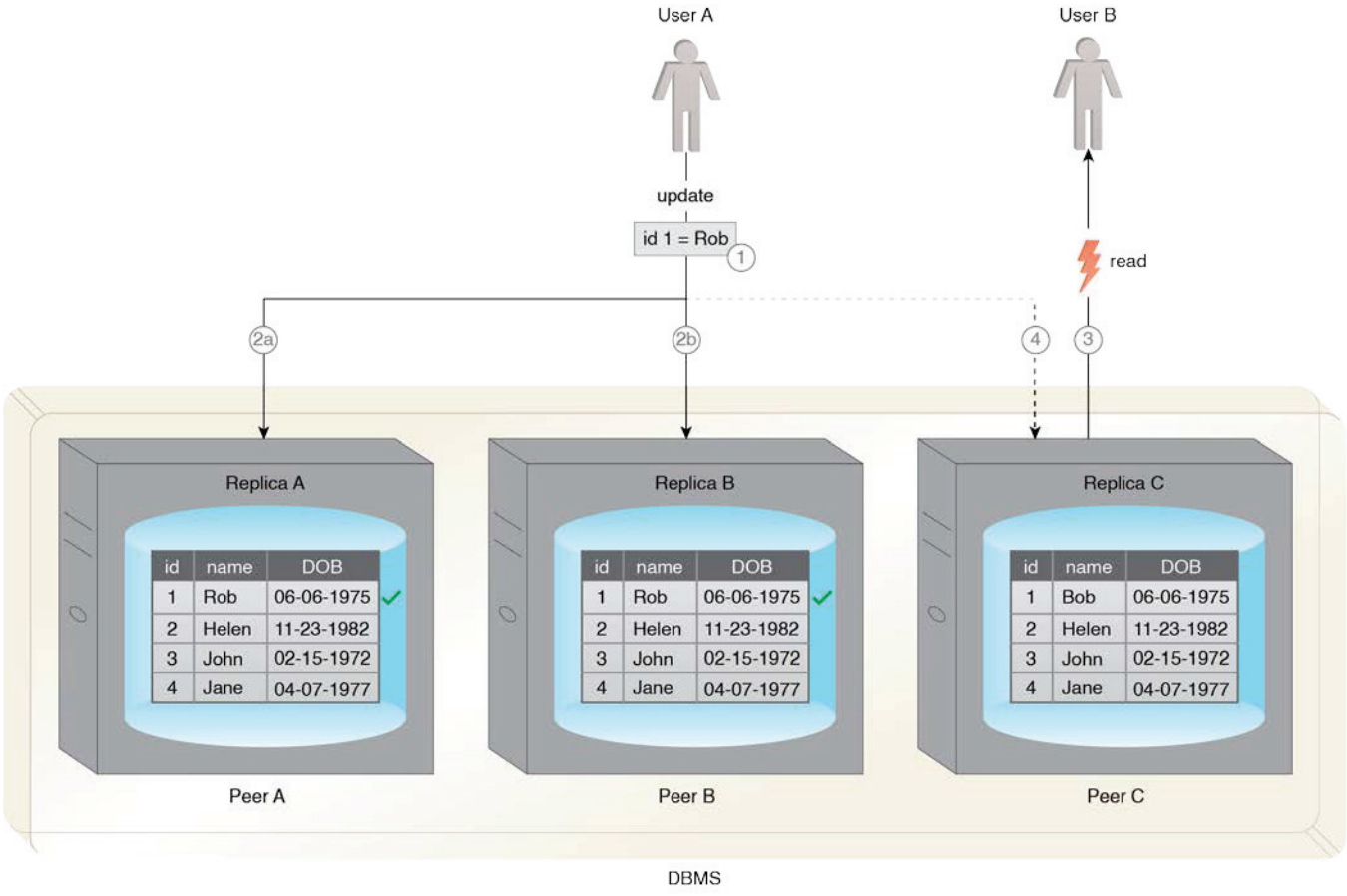


Figure 7.7 – An example of peer-to-peer replication where an inconsistent read occurs.

Sharding & Replication

To improve on the limited fault tolerance offered by sharding, while additionally benefiting from the increased availability and scalability of replication, both sharding and replication can be combined (Figure 7.8).

This section covers the following combinations:

- sharding and master-slave replication
- sharding and peer-to-peer replication

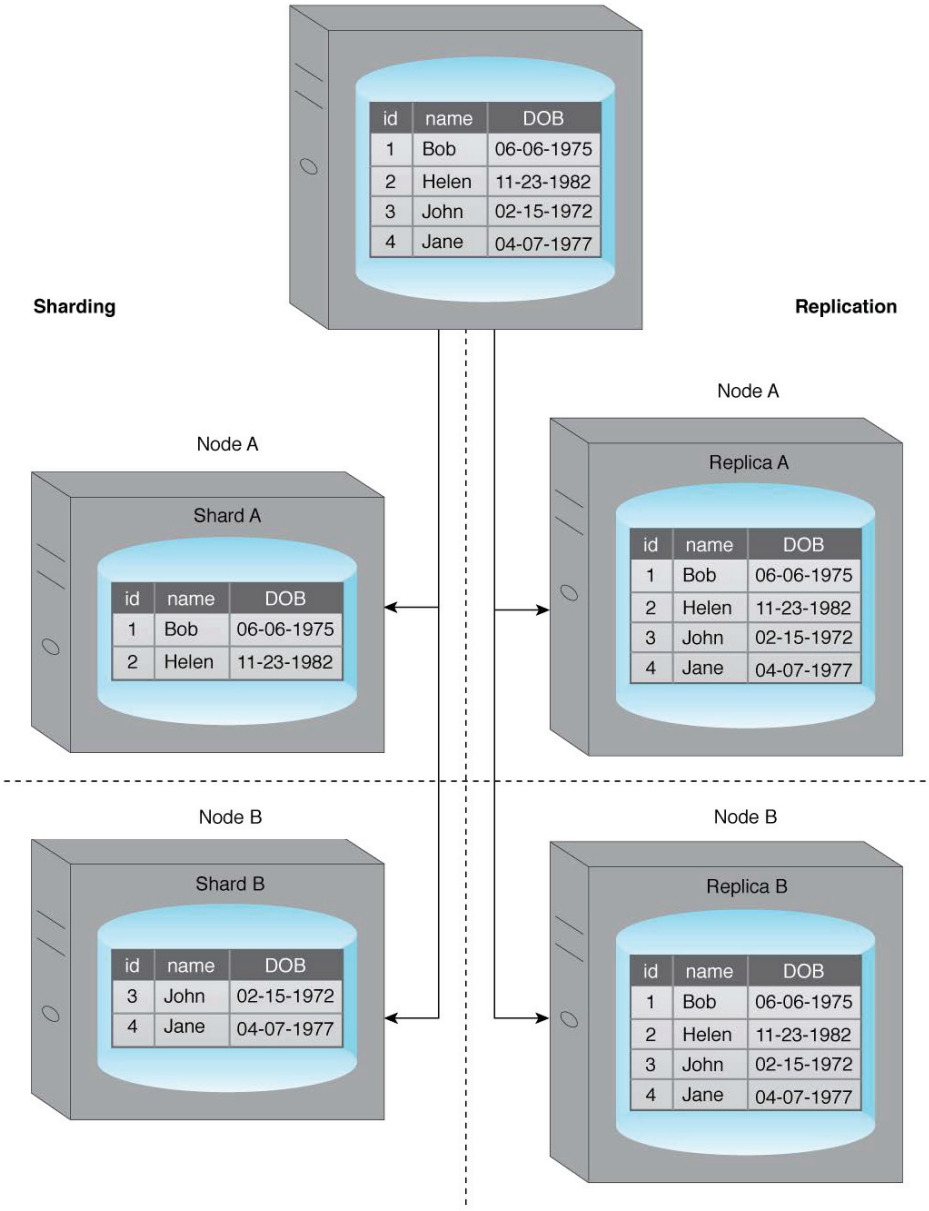


Figure 7.8 – A comparison of sharding and replication that displays how a dataset is distributed between two nodes when following each different approach.

Combining Sharding & Master-Slave Replication

Multiple shards become slaves of a single master, whereas the master itself is a shard. Although more than one master is possible, a single slave-shard can only be managed by a single master-shard.

Write consistency is maintained by the master-shard. However, this means that fault tolerance (with regards to write operations) is affected if the master-shard becomes non-operational or a network outage occurs. Replicas of shards are kept on multiple slave nodes to provide scalability and fault tolerance for read operations.

In Figure 7.9:

- 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.

Combining Sharding & Peer-to-Peer Replication

Each shard gets replicated to multiple peers, and each peer is only responsible for a subset of data rather than the complete dataset. Collectively, this helps to achieve increased scalability and fault tolerance. As there is no master involved, there is no single point of failure with regards to both read and write operations.

In Figure 7.10:

- 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.

Reading

For further discussion on sharding and replication, refer to the *Sharding & Replication* section on pages 38-44 of the *NoSQL Distilled* book that accompanies this module.

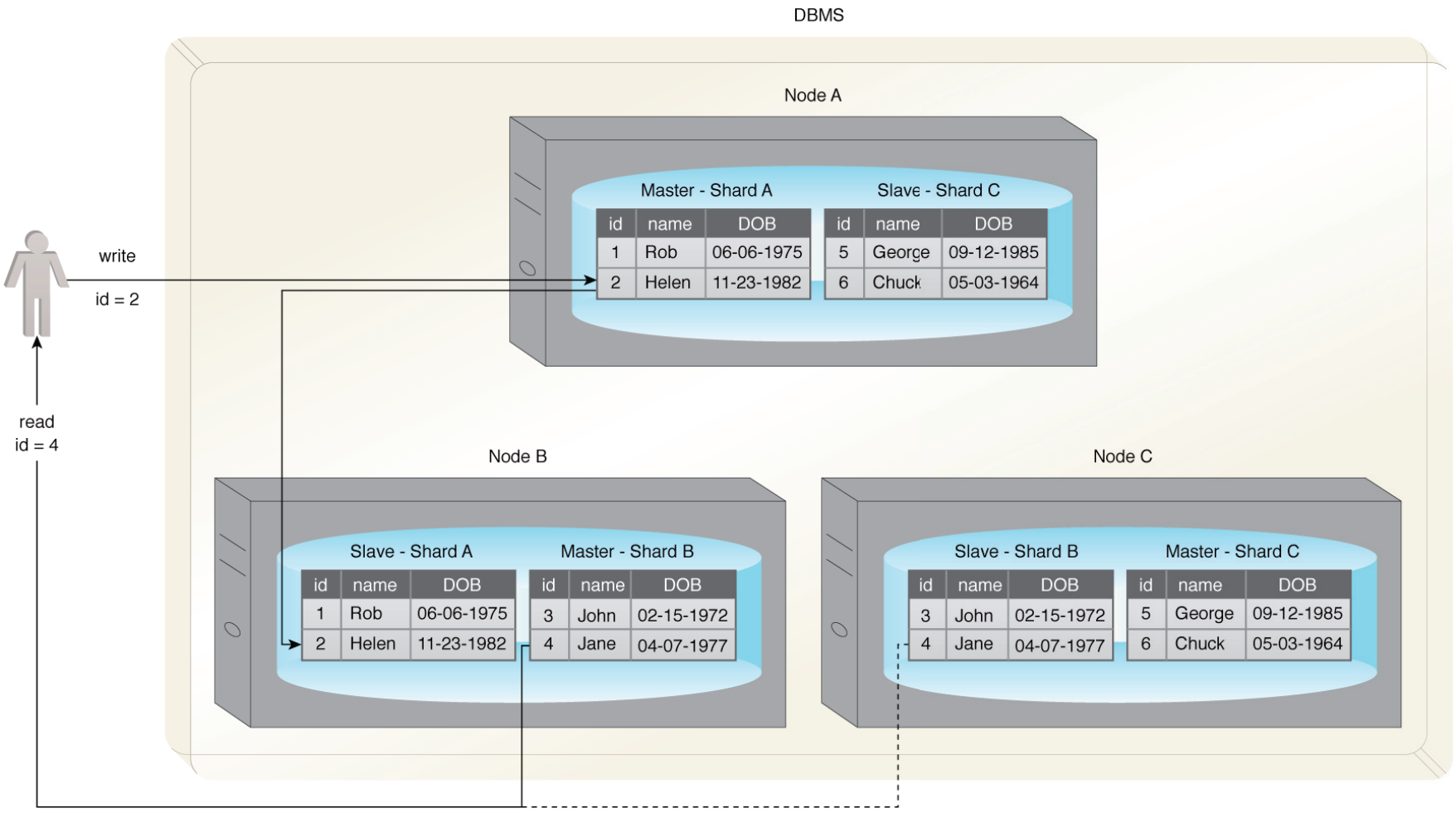


Figure 7.9 – An example of the combination of sharding and master-slave replication.

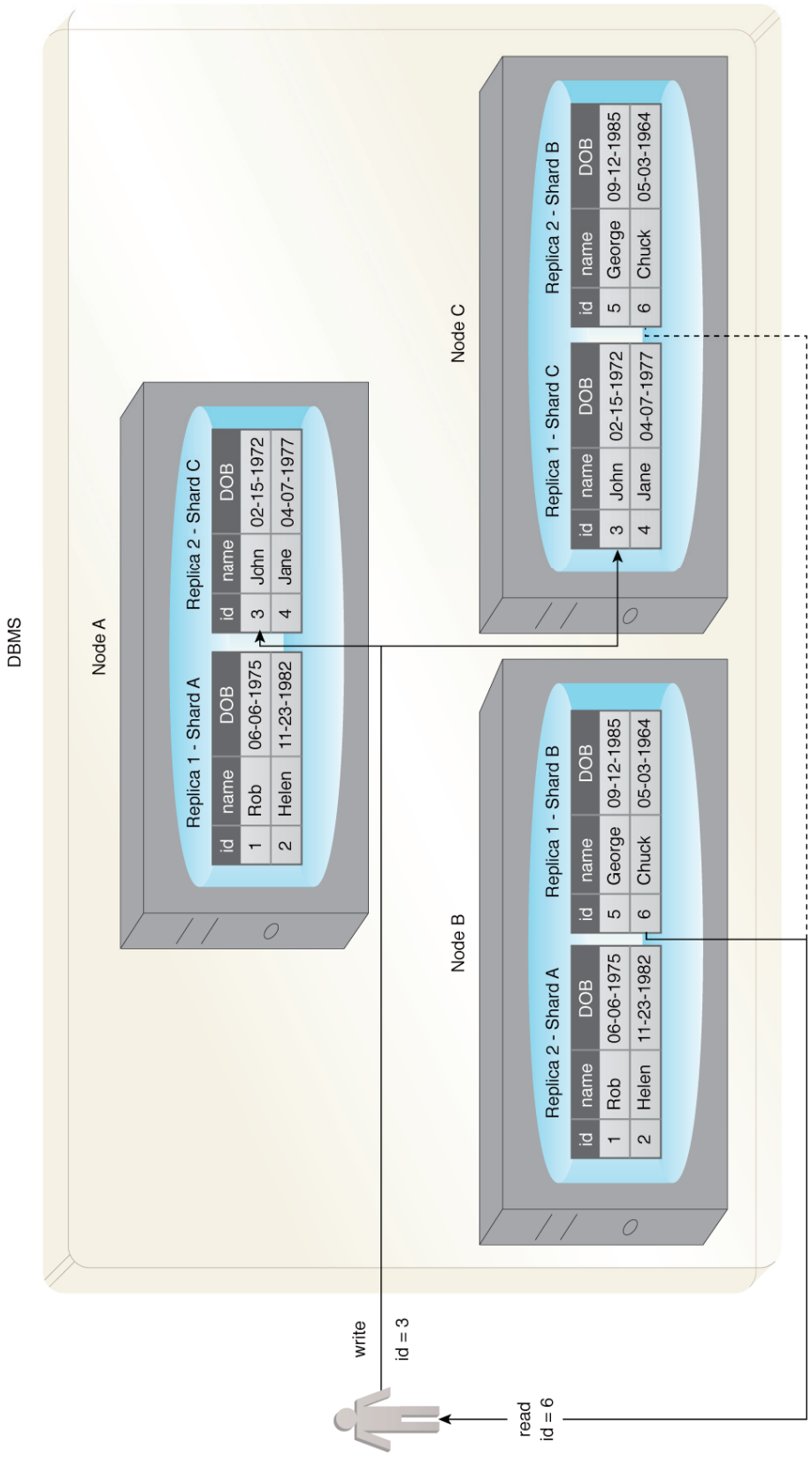


Figure 7.10 – An example of the combination of sharding and peer-to-peer replication.

CAP Theorem

The consistency, availability, and partition tolerance (CAP) theorem states that a distributed system, particularly a database, 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 (Figure 7.11).
- Availability – A read/write request will always be acknowledged in the form of a success or a failure (Figure 7.12).
- Partition tolerance – The database system can tolerate communication outages which split the cluster into multiple silos and can still service read/write requests (Figure 7.12).

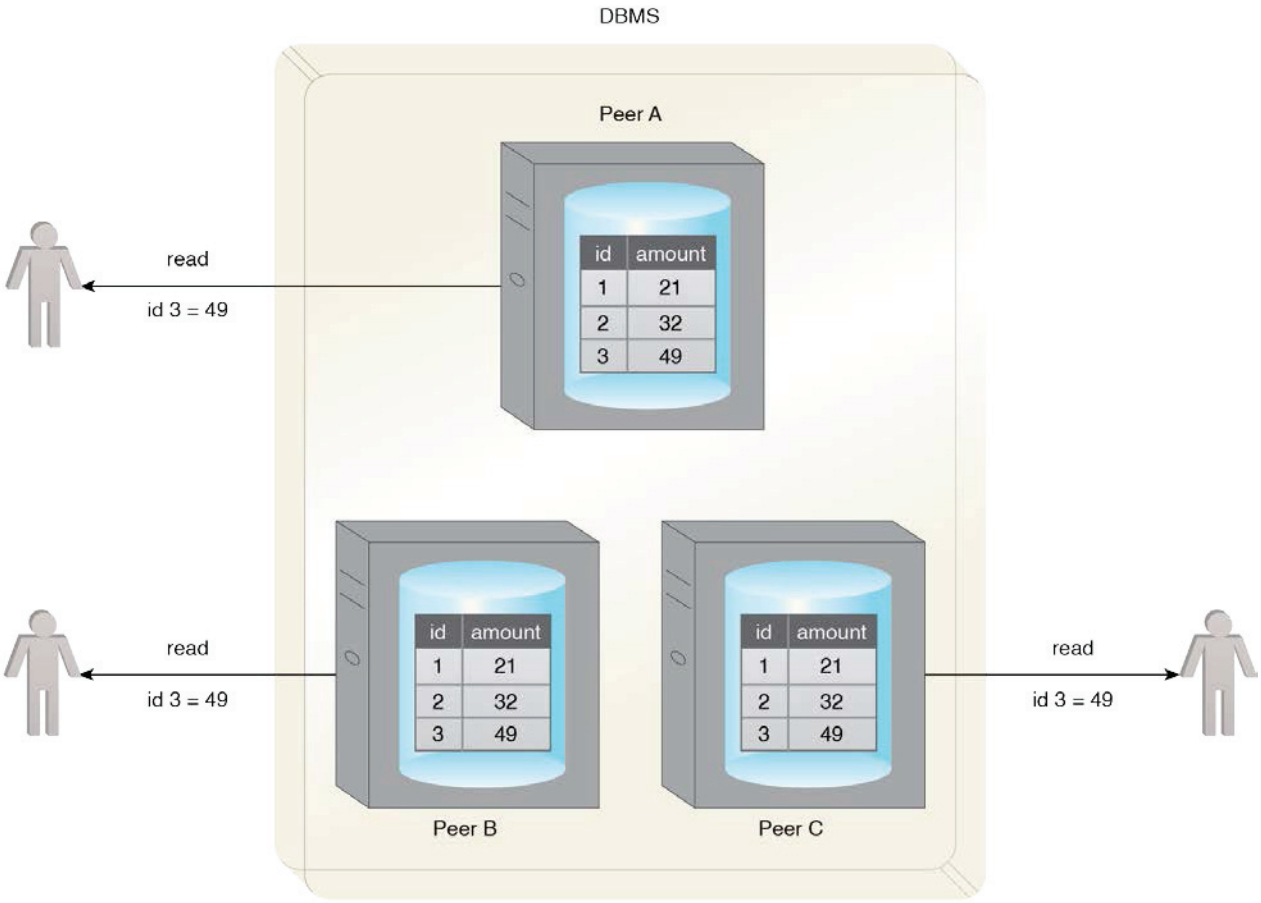


Figure 7.11 – Consistency: All three users get the same value for the amount column even though three different nodes are serving the record.

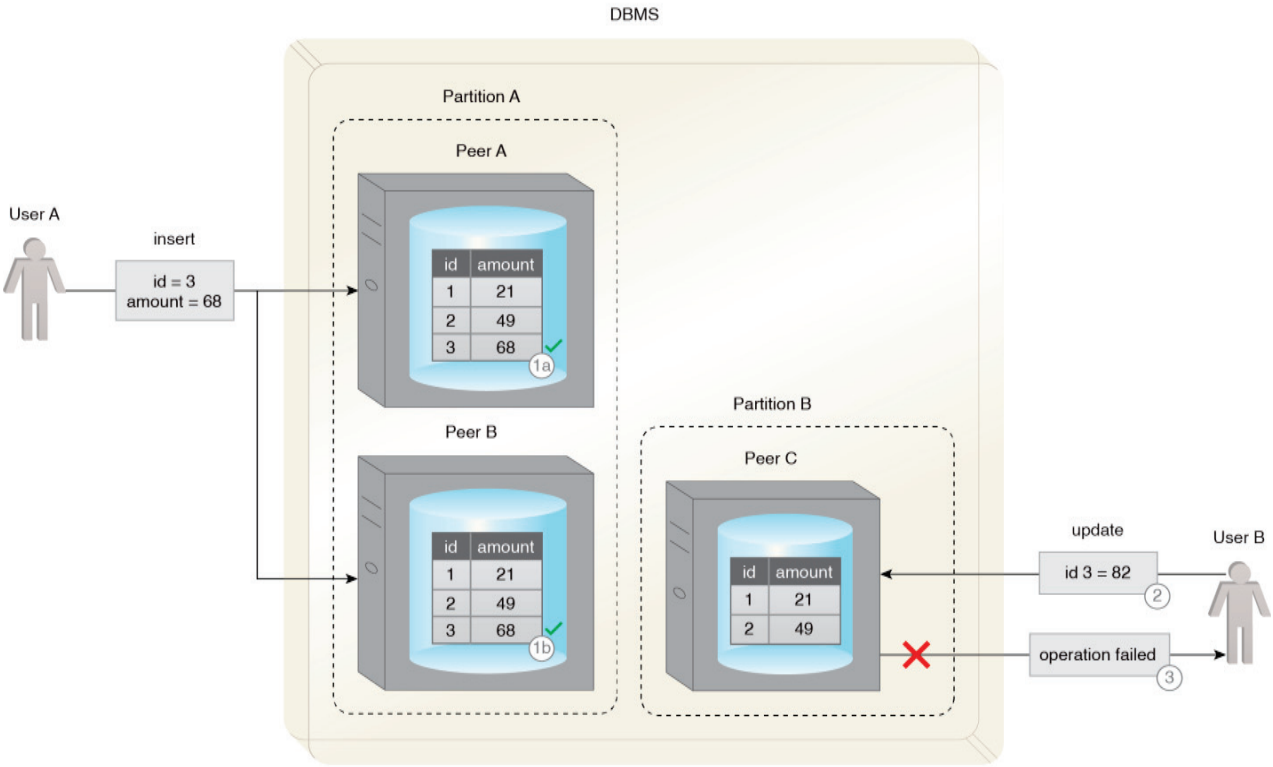


Figure 7.12 – Availability and partition tolerance: In the event of a communication failure, requests from both users are still serviced (1, 2). However, with User B, the update fails as the record with id = 3 has not been copied over to Peer C. The user is duly notified (3) that the update failed.

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 consistency (C) which cannot happen while supporting partition tolerance (P).

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), which 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. Thus, CAP is more about being either CP or AP.

On the other hand, RDBMSs are CA as they are generally available (A) while being consistent (C) at the same time. RDBMSs generally run on a single node, thus partition tolerance (P) is not a large consideration. It should be noted that choosing between CP or AP completely depends

on system requirements. This theorem is also known as Brewer's theorem after the name of its proposer.

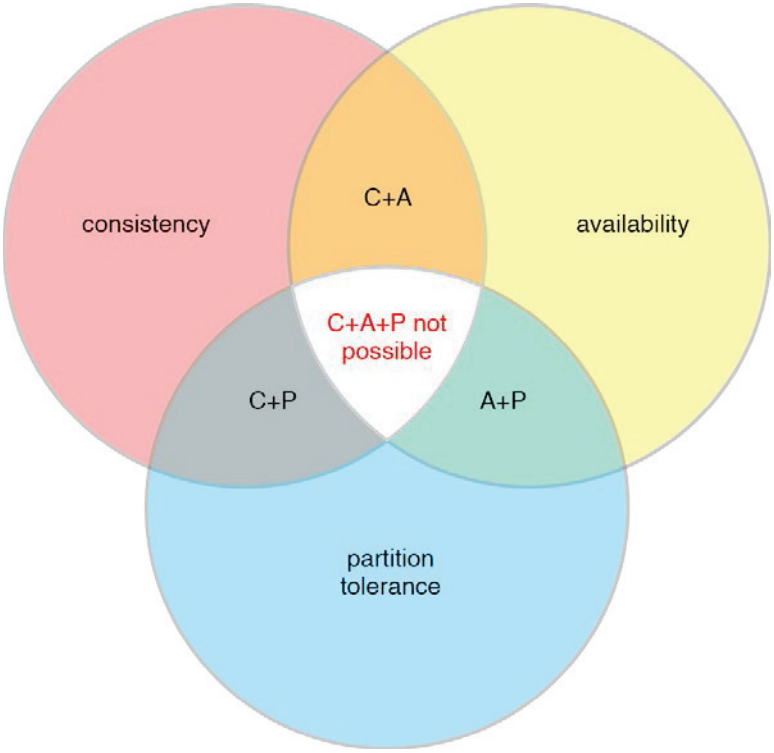


Figure 7.13 – A Venn diagram summarizing the CAP theorem.

Reading

For further discussion on the CAP theorem, refer to the *Additional Background on CAP Theorem* section on pages 53-56 of the *NoSQL Distilled* book that accompanies this module.

ACID

ACID is a database design principle that stands for:

- atomicity
- consistency
- isolation
- durability

Atomicity ensures that all operations will always succeed or fail completely. In other words, there are no partial transactions.

In Figure 7.14:

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 rollbacks any partial effects of the transaction and puts the system back to its prior state.

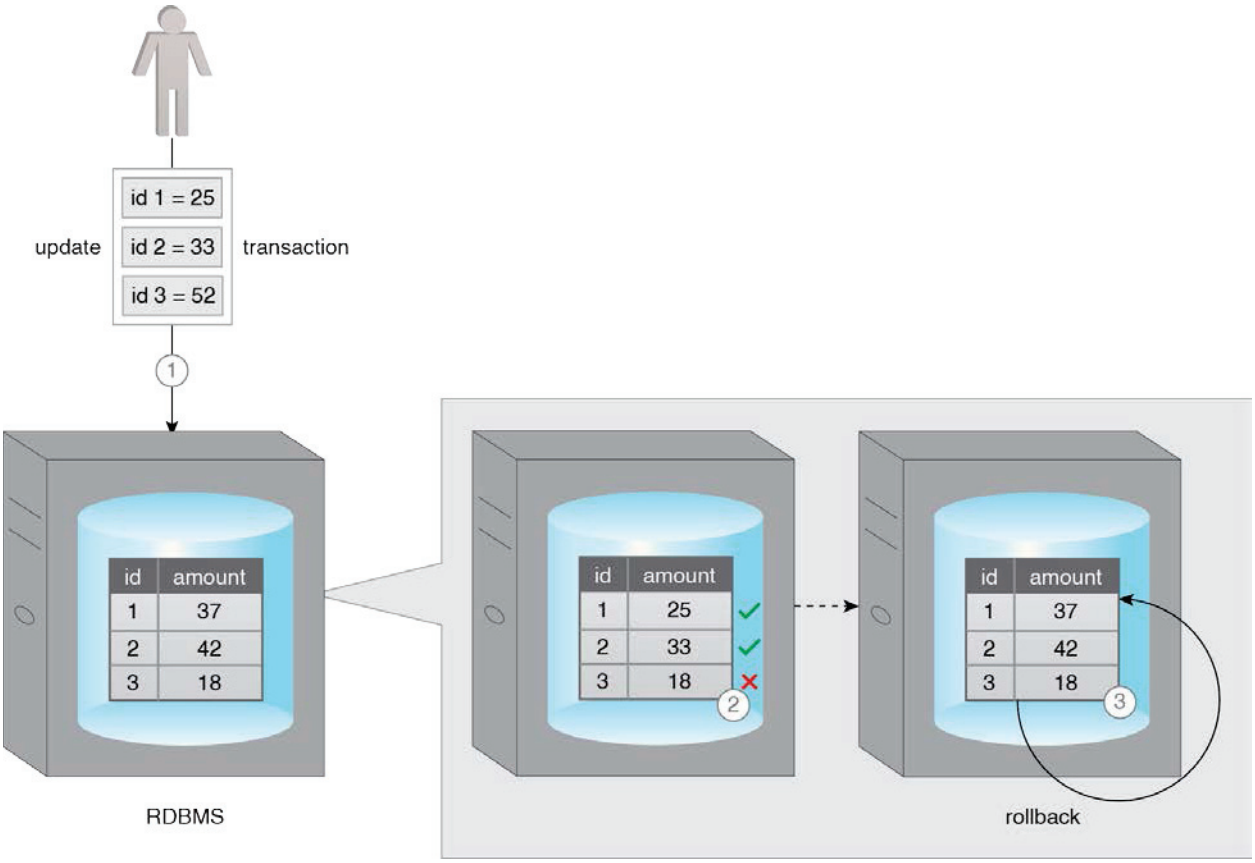


Figure 7.14 – An example of the atomicity property of ACID.

Consistency ensures that the database will only allow valid data, and will always be in a consistent state after an operation. Any write followed by an immediate read is guaranteed to be consistent across multiple clients.

In Figure 7.15:

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 of an invalid value for the amount column.

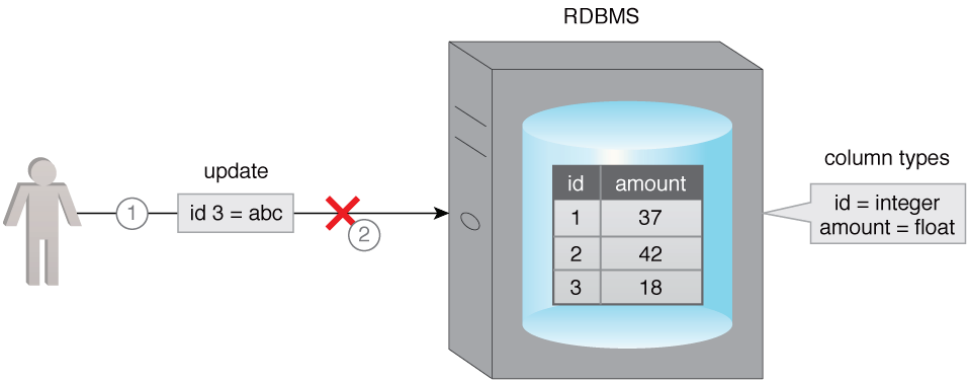
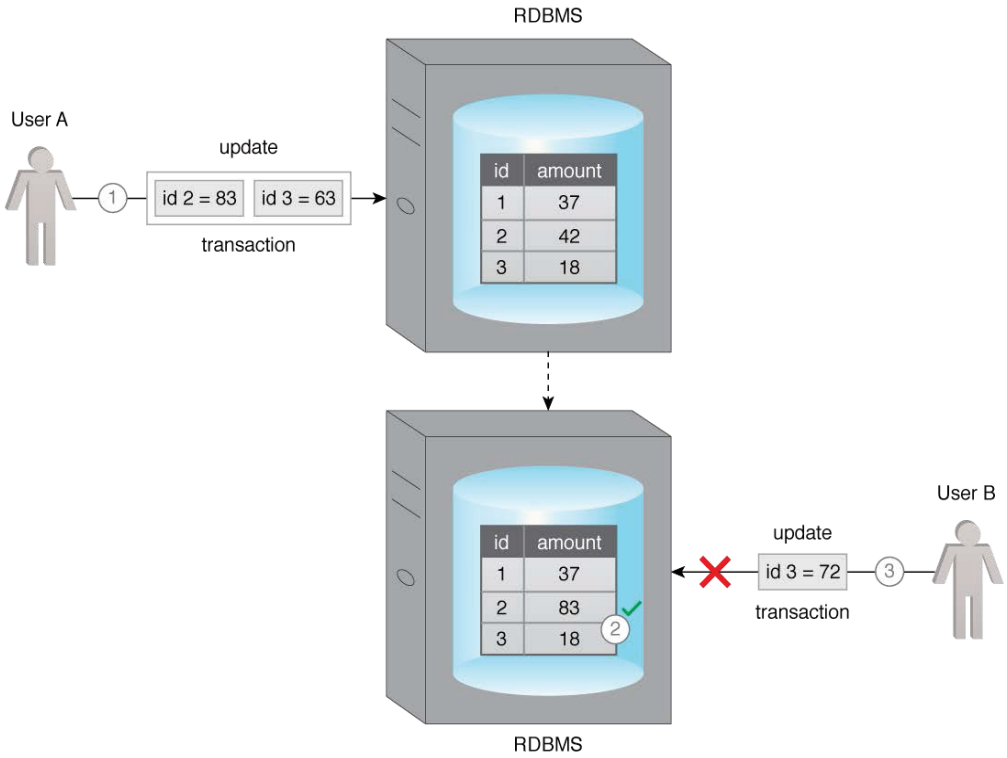


Figure 7.15 – An example of the consistency property of ACID.

Isolation ensures that the results of a transaction are not visible to other operations until it is complete.

In Figure 7.16:

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 one, 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.



7.16 – An example of the isolation property of ACID.

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 7.17:

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. It serves the record as per last update when requested by the user.

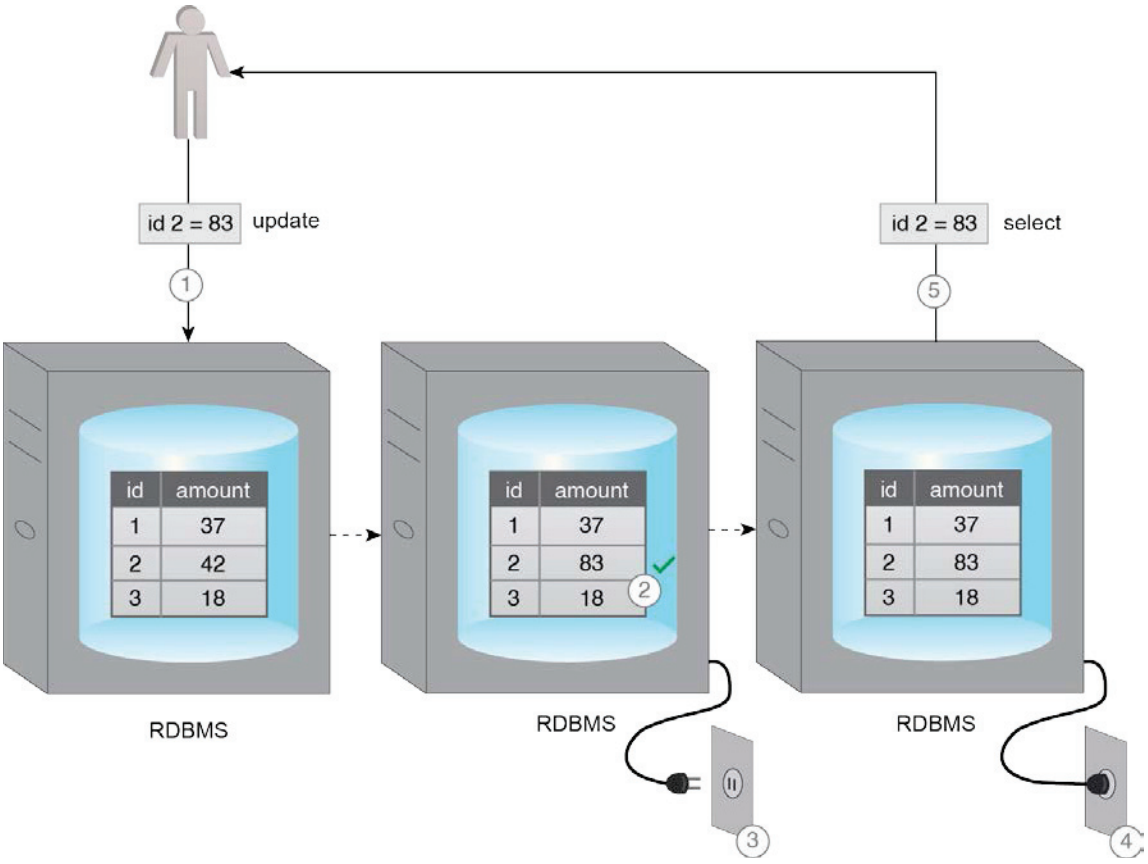


Figure 7.17 – An example of the durability property of ACID.

Figure 7.18 shows the results of the application of the ACID principle:

1. User A attempts to update a record as part of a transaction.
2. The database validates the value and the update is successfully applied.
3. After the successful completion of the transaction, when Users B and C request the same record, the database provides the updated value to both the users.

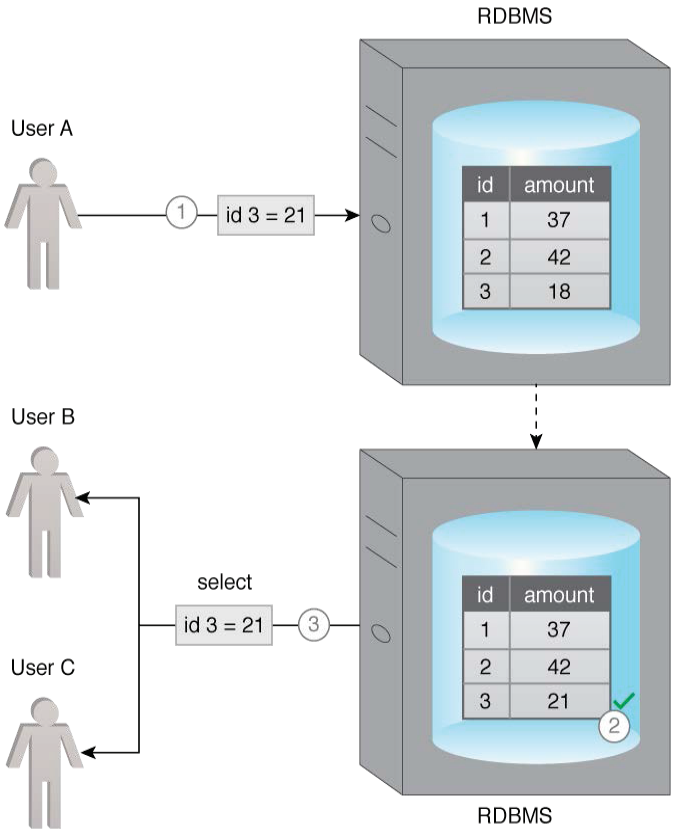


Figure 7.18 – An example illustrating the results of the application of the ACID principle.

Reading

The *ACID & NoSQL Databases* section on pages 19-20 of the *NoSQL Distilled* book that accompanies this module provides further discussion on ACID and NoSQL storage devices.

BASE

BASE is a database design principle based on the CAP theorem and followed by database systems that leverage distributed technology. BASE stands for:

- basically available
- soft state
- eventual consistency

Basically available means that the database will always acknowledge the client's request either in the form of requested data or a success/failure notification.

In Figure 7.19, both users are serviced even though the database system has been partitioned as a result of a network failure.

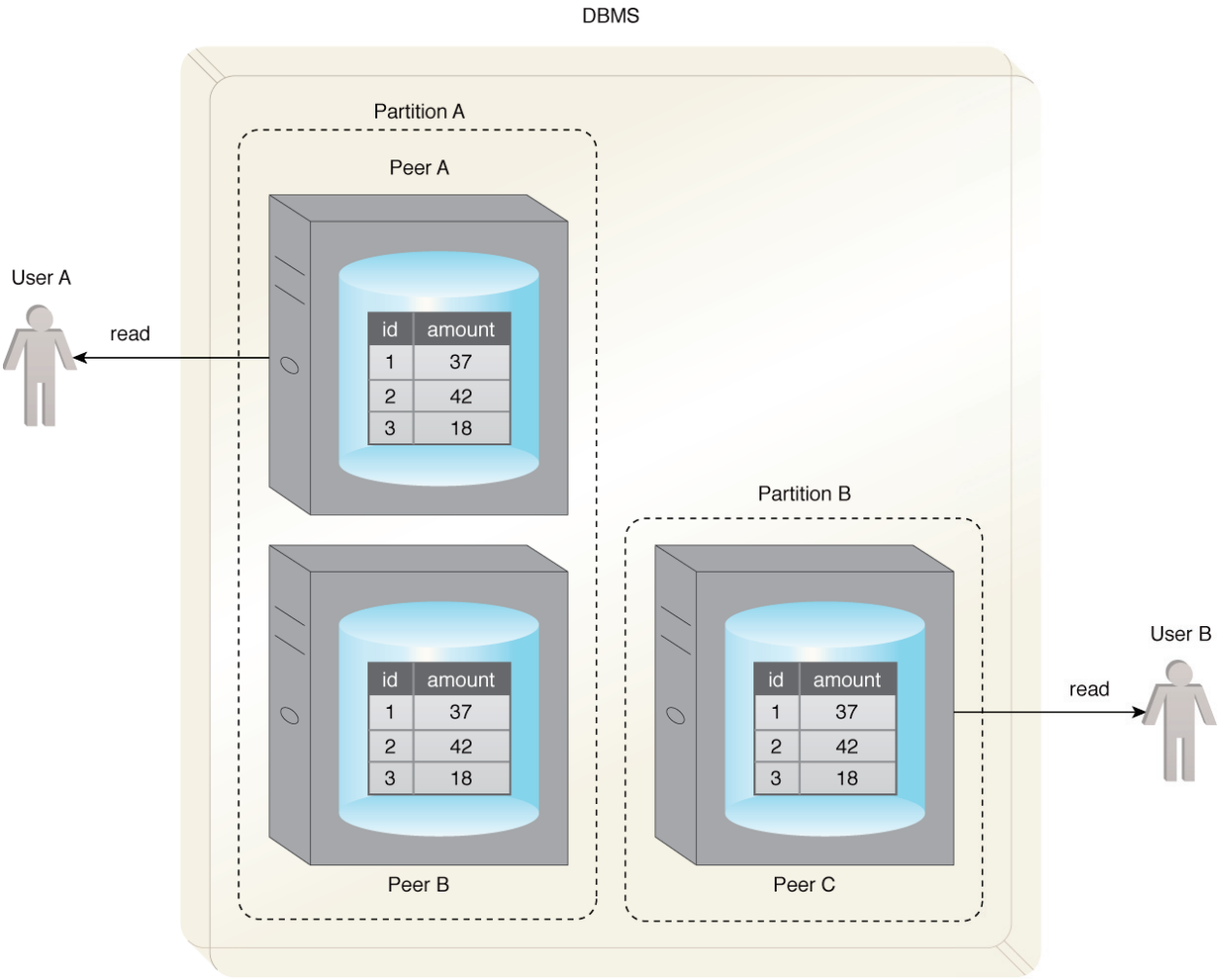


Figure 7.19 – User A and User B are serviced despite the database system partitioning due to network failure.

Soft state entails that the database may not be in a consistent 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 data between the two reads. This property is closely related to eventual consistency.

In Figure 7.20:

1. User A updates a record on Peer A.
2. Before the other peers can be 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.

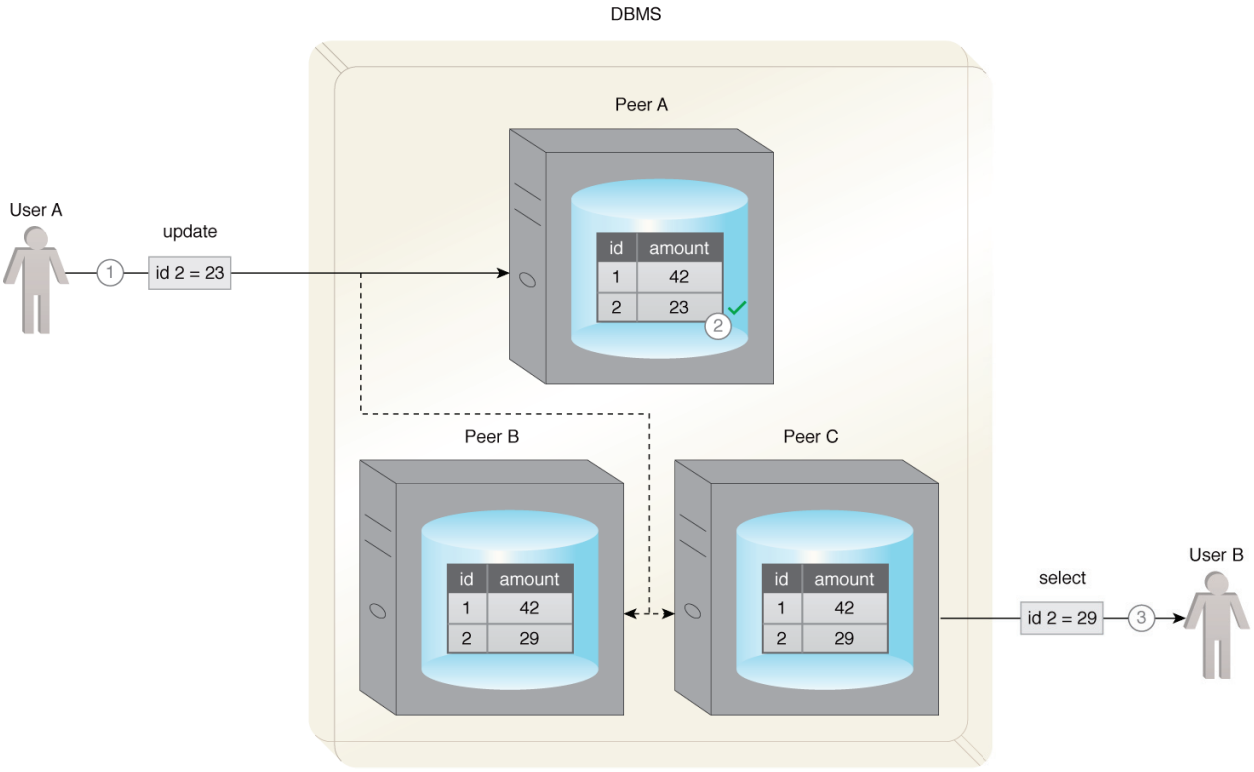


Figure 7.20 – An example of the soft state property of BASE.

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

In Figure 7.21:

1. User A updates a record.
2. The record only gets updated at Peer A. 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.

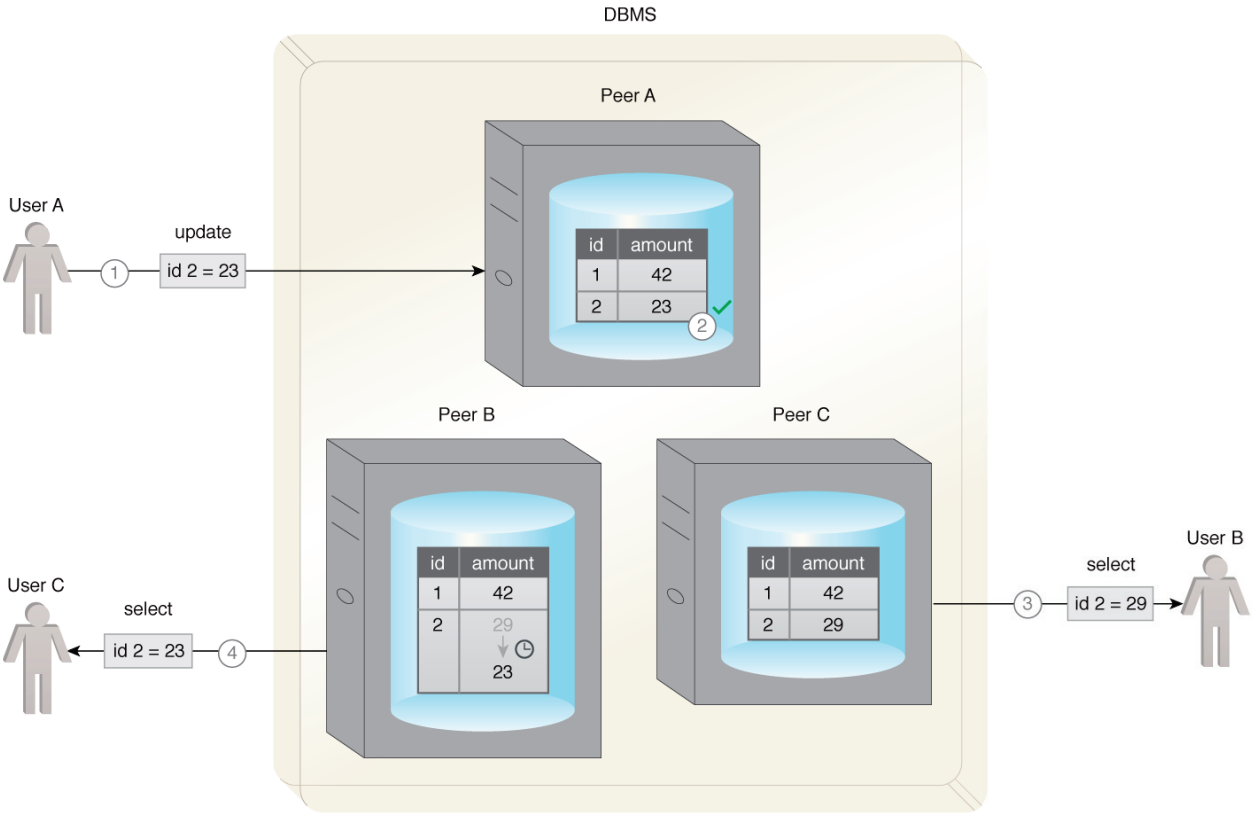


Figure 7.21 – An example of the eventual consistency property of BASE.

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 towards consistency allows BASE compliant databases to serve multiple clients without any latency albeit serving inconsistent results. However, BASE compliant databases are generally not used for transactional systems where lack of consistency can become a concern.

Exercise 7.1: Fill in the Blanks

1. _____ is the process of horizontally partitioning a large dataset into a collection of smaller, more manageable datasets called _____, spread across multiple nodes.

2. In sharding, for data partitioning, _____ need to be taken into account so that shards themselves do not become performance bottlenecks.

3. _____ makes multiple copies of a dataset, called _____, and stores them across multiple nodes.

4. The two methods of implementing replication include _____ and _____ replication.

5. In _____ replication, the _____ node is the single point of contact for all writes, while data can be read from any _____ node.

6. In _____ replication, there are no _____ and _____ nodes, and all nodes, called _____, operate at the same level.

7. Sharding and replication can be combined to improve on the limited _____ offered by sharding, while benefiting from the increased _____ and _____ of replication.

8. The CAP theorem says that a distributed system can only support two out of three properties, which are _____, _____ and _____.

9. In the context of the CAP theorem, relational databases support _____ and _____.

10. In the context of database design, BASE stands for _____, _____ and _____.

Exercise answers are provided at the end of this booklet.

Notes

Notes / Sketches

Big Data Storage Device Characteristics

Big Data consists of datasets that cannot be stored using traditional storage solutions, mainly because of the large volume of data they contain. Velocity is a factor that makes traditional database solutions inappropriate for Big Data storage, mainly because of their centralized design which offers little or no scalability.

The variety characteristic of Big Data datasets requires special attention as it is estimated that 80% of Big Data datasets are unstructured data, and traditional storage solutions do not support storing semi-structured and unstructured data in a scalable and efficient manner.

Due to the unique characteristics of Big Data, Big Data datasets cannot be persisted using traditional storage solutions. However, before choosing an alternative storage mechanism for a Big Data solution environment, the following storage device characteristics need to be considered:

- scalability
- redundancy and availability
- fast access
- long-term storage
- schema-less storage
- inexpensive storage

Scalability

Big Data datasets come in huge volumes at a fast pace and are generally acquired from multiple sources. This results in large amounts of data within a short period of time. With the potential of finding new insights about the way businesses work, enterprises are retaining more and more data, both generated inside the enterprise and obtained from outside sources, as part of their data acquisition activities.

It may not be possible to re-acquire data due to expensive acquisition costs, for example when a dataset is purchased from a data provider. In addition, it may not be possible to re-generate data if the events that generated the data initially were one-off events, for example a smart meter reading for a certain point in time. In either case, we need a scalable storage solution with enough capacity for current and future data capture requirements.

Apart from storing the raw data, additional storage is required to store the data created as a result of the data wrangling activity. The data storage requirement, in the case of joining multiple datasets, will increase in size due to the need to keep both the original datasets and the joined dataset.

Storage is further required in order to persist the analytic results from a data analysis activity. In order to address the increasing demands for data storage, a storage device can either scale up or scale out.

Scalability: Scale Up

Scaling up, also called vertical scaling, is a strategy for increasing resource capacity by replacing an existing low capacity device with a higher capacity device. For example, to double the capacity of a storage device, data from a 500GB disk can be copied over to 1TB disk. Scaling up will cause disruption and system downtime.

Scalability: Scale Out

Scaling out, also called horizontal scaling, is a strategy for increasing resource capacity by adding similar or higher capacity commodity devices alongside the existing device. For example, to double the storage capacity, a 500GB disk can be added alongside an existing 500GB disk. Scaling out does not cause disruption and system downtime.

Scalability

Considering the differences between scaling up and scaling out, a storage device should be able to scale out as this is a lower cost strategy that enables increasing storage capacity without incurring system downtime and interruption. Sharding can help create scalable storage as shards can be stored on the nodes added via scaling out (Figure 7.22).

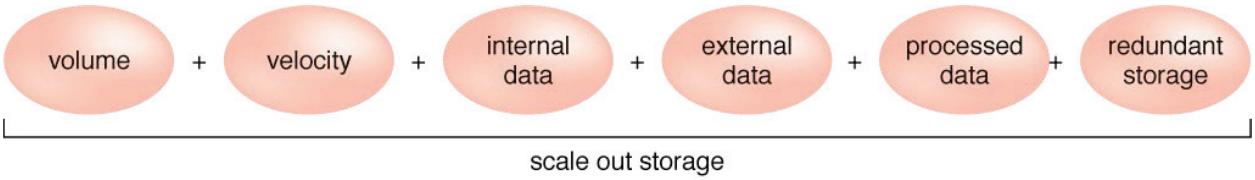


Figure 7.22 – An example of components of scale out storage.

Redundancy & Availability

Big Data datasets, both in raw and processed form, are a business asset and require attention with regards to storage. Multiple business functions may need to glean value out of these datasets, sometimes simultaneously. This dependence on Big Data datasets across an enterprise requires a reliable storage device that is fault-tolerant and is highly available.

A Big Data solution environment is generally composed of clusters that are built using commodity servers connected via a high bandwidth network. With more nodes and network connections, the chances of a node becoming unavailable increases either due to hardware breakdown such as disk failure, or due to network failure. As a result, storage redundancy is required to ensure uninterrupted access to data in the event of a storage device failure, thereby providing high availability and fault tolerance.

To provide such redundancy, a storage device implements automatic sharding and replication with a configuration of either sharding and master-slave replication or sharding and peer-to-peer replication.

Fast Access

Big Data analytics generally involve both realtime as well as batch processing. Realtime data analysis requires fast read/write capabilities that are usually implemented via in-memory storage solutions. On the other hand, batch processing requires stream-based data access with high throughput, implemented via traditional disk-based storage devices or newer solid state drives that provide better performance at the expense of a higher cost. At the same time, data arriving at high velocity needs a storage device that supports fast writes with minimal overhead, for example, schema validation at read time instead of write time.

Long-term Storage

The basic tenet of Big Data is to extract value from large amounts of data. This requires enterprises to retain data for longer periods of time to create larger datasets so that future data analyses are more insightful due to having more historic data available. This characteristic warrants the need for a storage device with increased storage capacity that is reliable and can be brought online without incurring too much time delay.

Traditionally, historic data that is no longer required is generally archived in offline storage. However, this makes the historic data unavailable for instant analysis. Contrary to this, Big Data analytics require historic data to be available online for discovering hidden patterns leading to valuable insights. As a result, historic data is kept online by adding more storage capacity via scaling out.

In traditional data storage environments, once the database reaches its capacity, data is offloaded to tape drives. In Big Data storage environments, instead of offloading data to tape drives, historical data is kept online and the storage capacity is simply increased via scaling out (Figure 7.23).

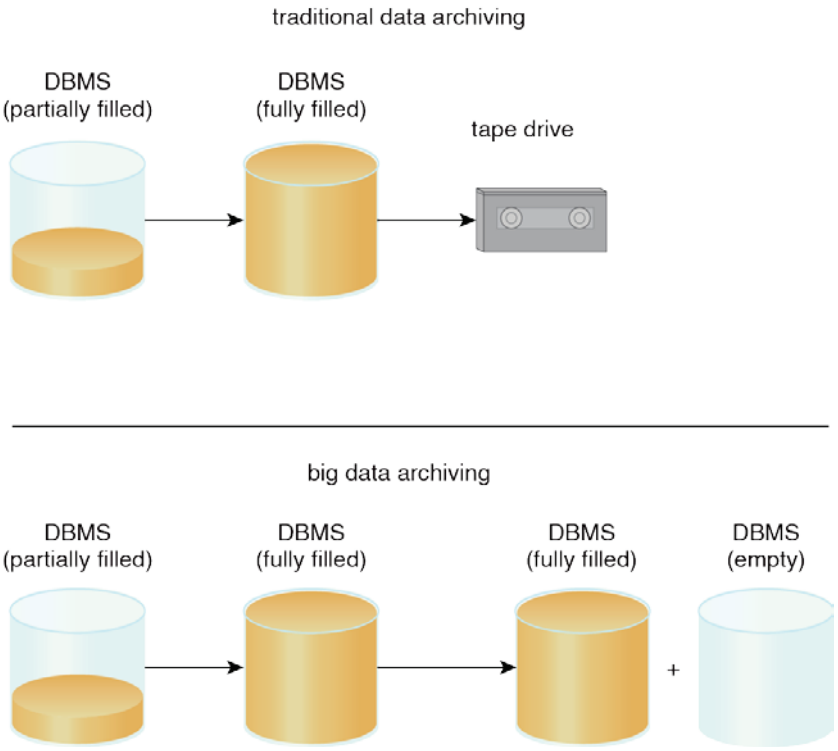


Figure 7.23 – A comparison of traditional data archiving and big data archiving.

Schema-less Storage

Big Data datasets come in multiple formats with limited or no schema, such as semi-structured and unstructured data. Without any prior knowledge about the structure of the data, schema enforcement is not possible. Similarly, even having some knowledge about the structure of the data does not guarantee that the data would conform to the same structure in the future, as is the nature of unstructured data. This requires the storage device to support schema-less data persistence along with the added ability to make schema changes on the fly without breaking existing applications or incurring downtime.

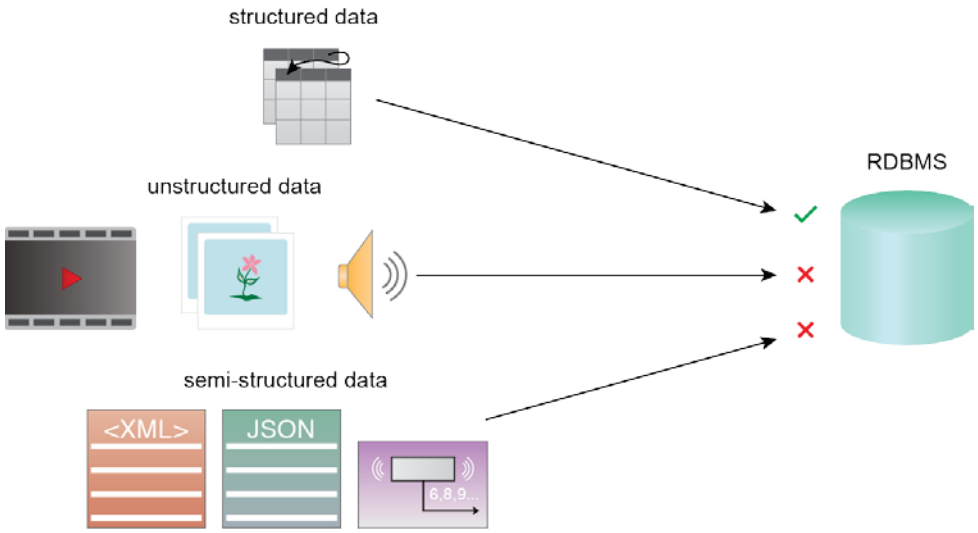


Figure 7.24 – An example of traditional storage.

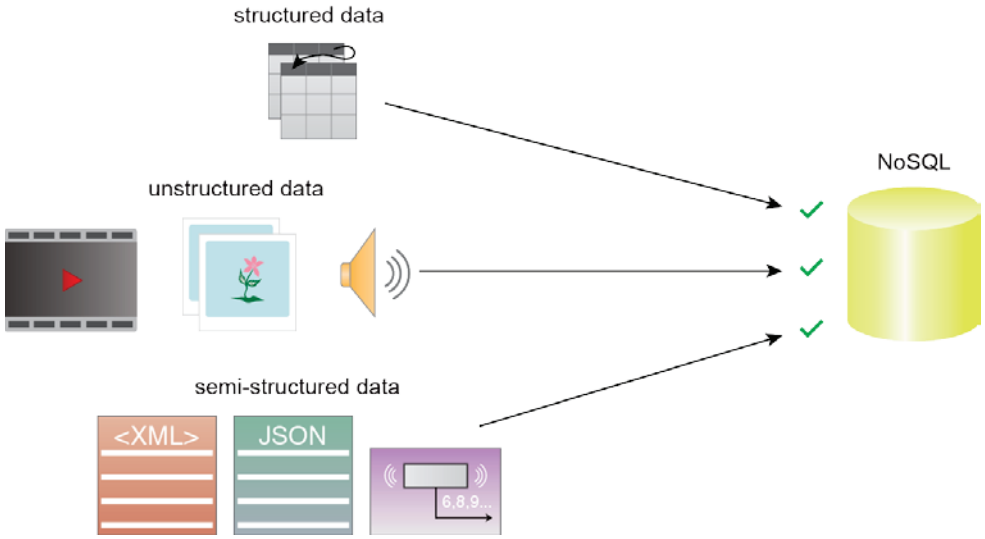


Figure 7.25 – An example of Big Data storage.

Inexpensive Storage

With all the characteristics required to store voluminous data, provide replication, and support long-term storage, the cost of storage devices can become a concern. A storage device must make efficient use of the underlying disk resources in order to minimize storage waste.

Use of proprietary storage devices generally requires replacement of existing nodes with more expensive ones in order to scale up, eventually hitting a limit. A storage device needs to make use of commodity hardware that can scale out so that costs can be kept down as enterprises amass more and more data.

Notes

Notes / Sketches

On-disk Storage Devices

Based on the type of storage medium used, storage devices can be broadly divided into two types:

- on-disk storage
- in-memory storage

The upcoming material describes the types and characteristics of an on-disk storage device, along with its suitability in light of the already established Big Data storage characteristics. In-memory storage devices are covered in Module 8.

On-Disk Storage Device

On-disk storage generally utilizes low cost hard disk drives for long-term storage. On-disk storage can be implemented via a distributed file system or a database, shown in Figure 7.26.

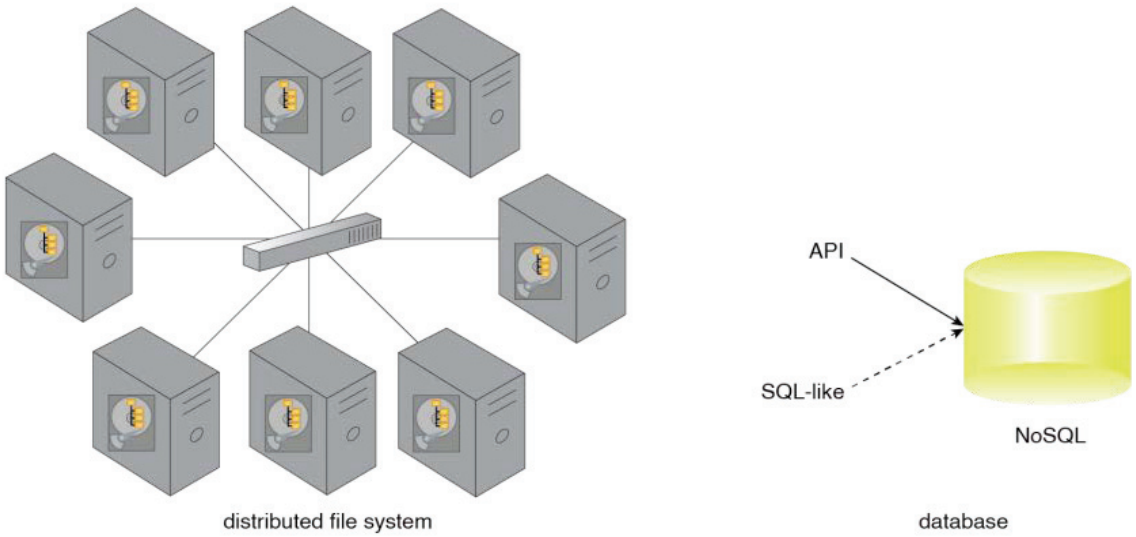


Figure 7.26 – The symbols used to represent a distributed file system and a NoSQL database.

On-Disk Storage Device: Distributed File System

A storage device that is implemented with a distributed file system provides simple, fast access data storage that is capable of storing large datasets that are non-relational in nature, such as semi-structured and unstructured data. Although based on straightforward file locking mechanisms for concurrency control, it provides fast read/write capability, which addresses the velocity characteristic of Big Data.

Distributed file systems are a good fit when data must be accessed in streaming mode with no random reads and writes (Figure 7.27).

A distributed file system is not ideal for datasets comprising a large number of small files as this creates excessive disk-seek activity, slowing down the overall data access. There is also 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.

Due to these limitations, distributed file systems work best with fewer but larger files accessed in a sequential manner. Multiple smaller files are generally combined into a single file to enable optimum storage and processing. Note that distributed file systems do not provide any file searching capability out of box.

When considering file systems within the realm of Big Data, the only true candidate file systems are distributed file systems. This is because non-distributed file systems cannot be employed in clustered environments, which is a requirement for processing large datasets using clusters. (Cluster architecture is introduced in the upcoming *Fundamental Big Data Processing* section.)

A distributed file system storage device is suitable when large datasets of raw data are to be stored or when archiving of datasets is required. In addition, it provides an inexpensive storage option for storing large amounts of data over a long period of time that remains online. This is because more disks can simply be added without needing to offload the data to offline data storage, such as tape drives.

Distributed file systems, like any file system, are agnostic to the data being stored and therefore support schema-less data storage. In general, a distributed file system storage device provides out of box redundancy and high availability by copying data to multiple locations via replication.

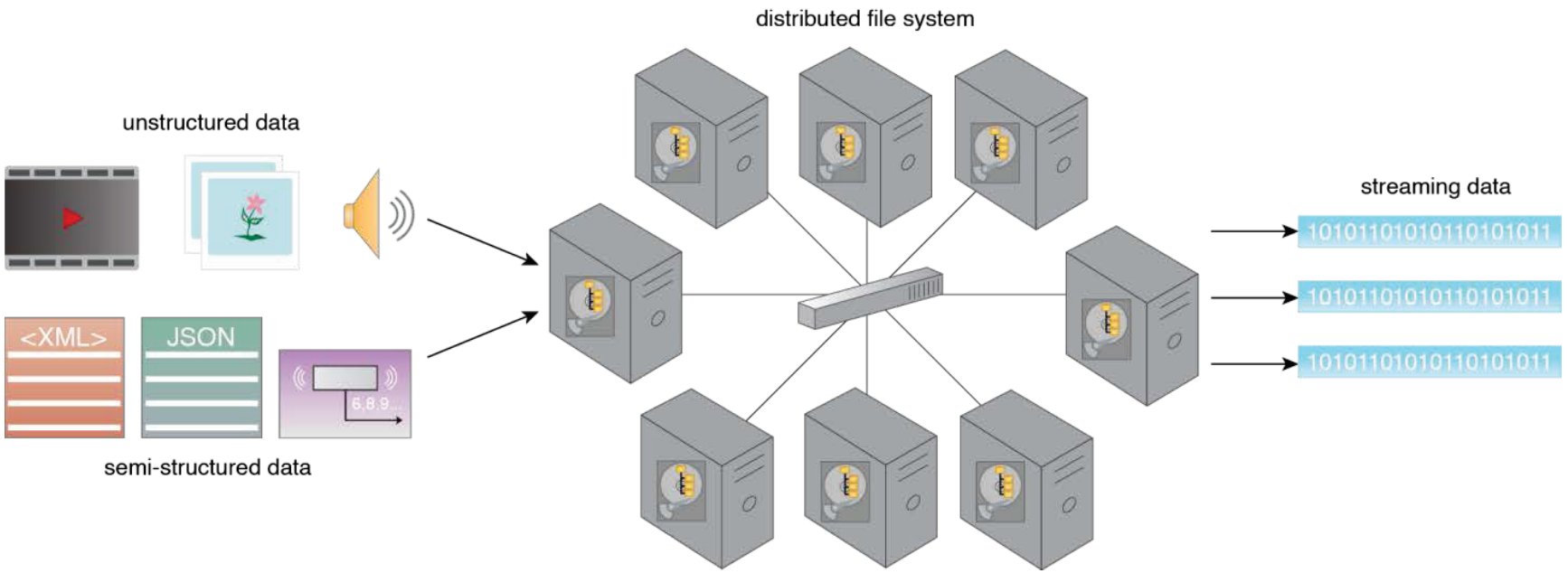


Figure 7.27 – A distributed file system accessing data in streaming mode with no random reads and writes.

On-Disk Storage Device: Database

Three types of database technologies are prevalent in Big Data solution environments for on-disk storage:

- relational databases or relational database management systems (RDBMSs)
- non-relational databases or Not only SQL (NoSQL)
- NewSQL

The upcoming content examines the suitability of these three database categories as storage devices for Big Data.

RDBMS

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

To handle large volumes of data arriving at a fast pace, relational databases generally need to scale. 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.

Note that some relational databases, for example IBM DB2 pureScale, Sybase ASE Cluster Edition, Oracle Real Application Clusters (RAC) and Microsoft Parallel Data Warehouse (PDW), are capable of being run on clusters (Figure 7.28). However, these database clusters still use shared storage that can act as a single point of failure.

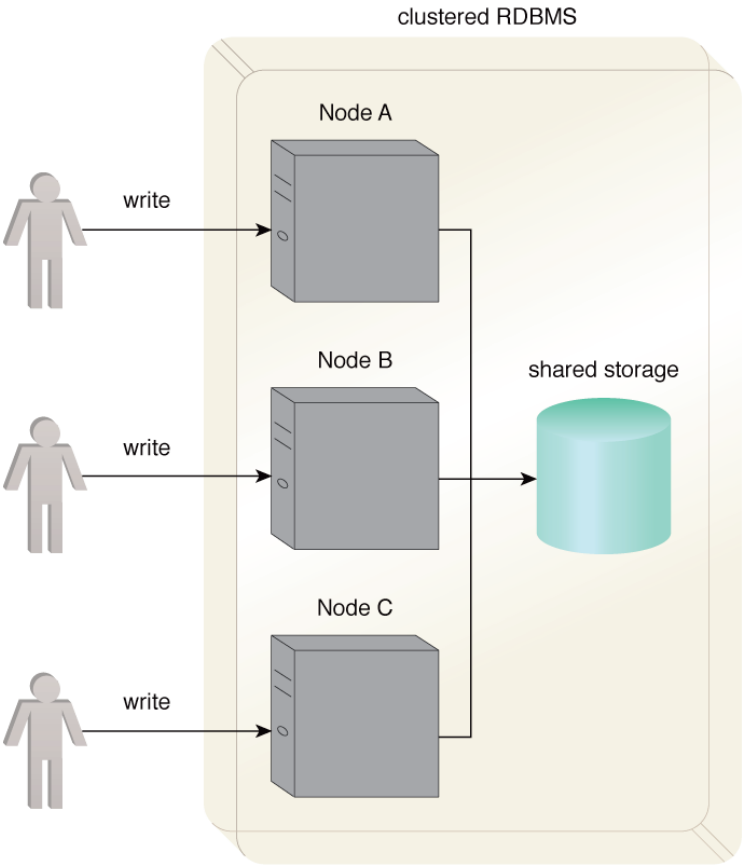


Figure 7.28 – A clustered relational database uses a shared storage architecture which is a potential single point of failure that affects the availability of the database.

With regards to sharding, relational databases need to be manually sharded, mostly using application logic. This means that the client (application logic) needs to know which shard to query in order to get the required data. This further complicates the data processing when data from multiple shards is required.

The following steps are shown in Figure 7.29:

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.

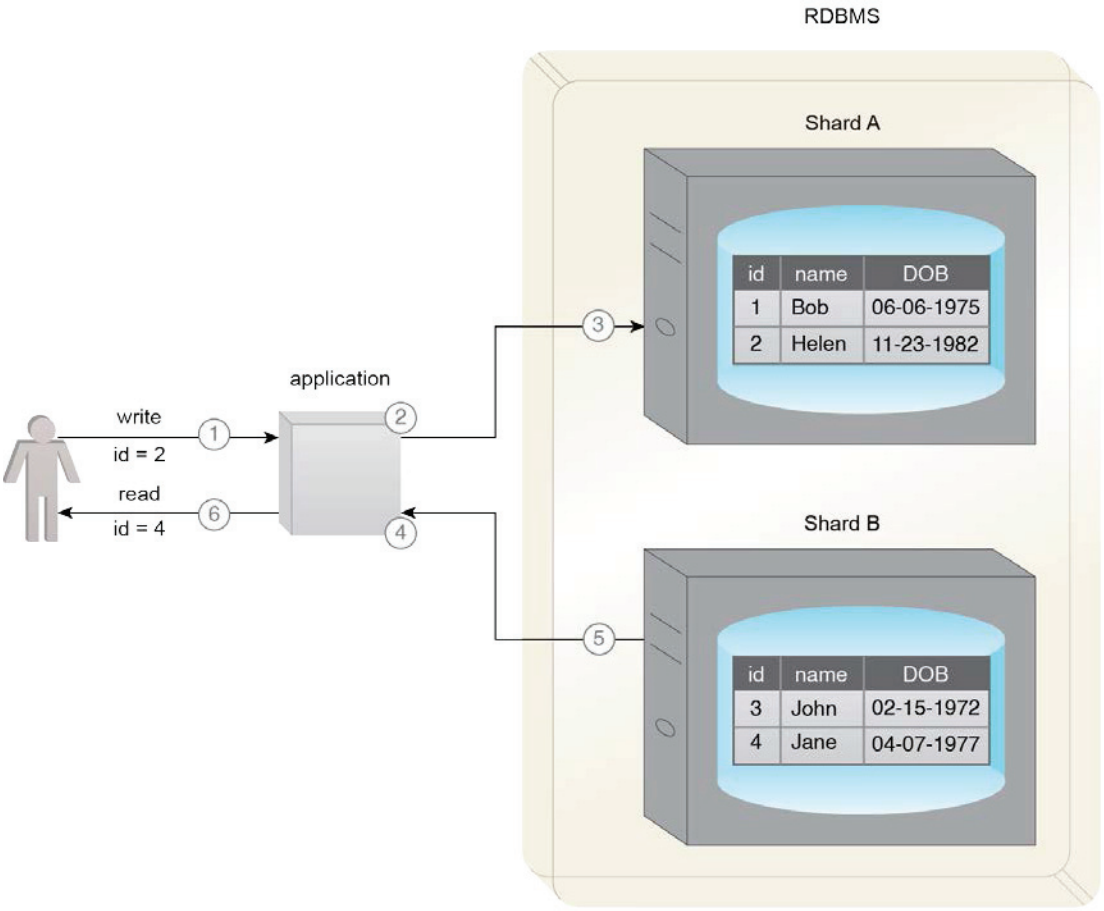


Figure 7.29 – An example of a relational database that is manually sharded using application logic.

The following steps are shown in Figure 7.30:

1. A user requests multiple records (id = 1, 3) and the application logic is used to determine which shards need to be read.
2. It is determined by the application logic that both Shard A and Shard B need to be read.
3. The data is read and joined by the application.
4. Finally, the data is returned to the user.

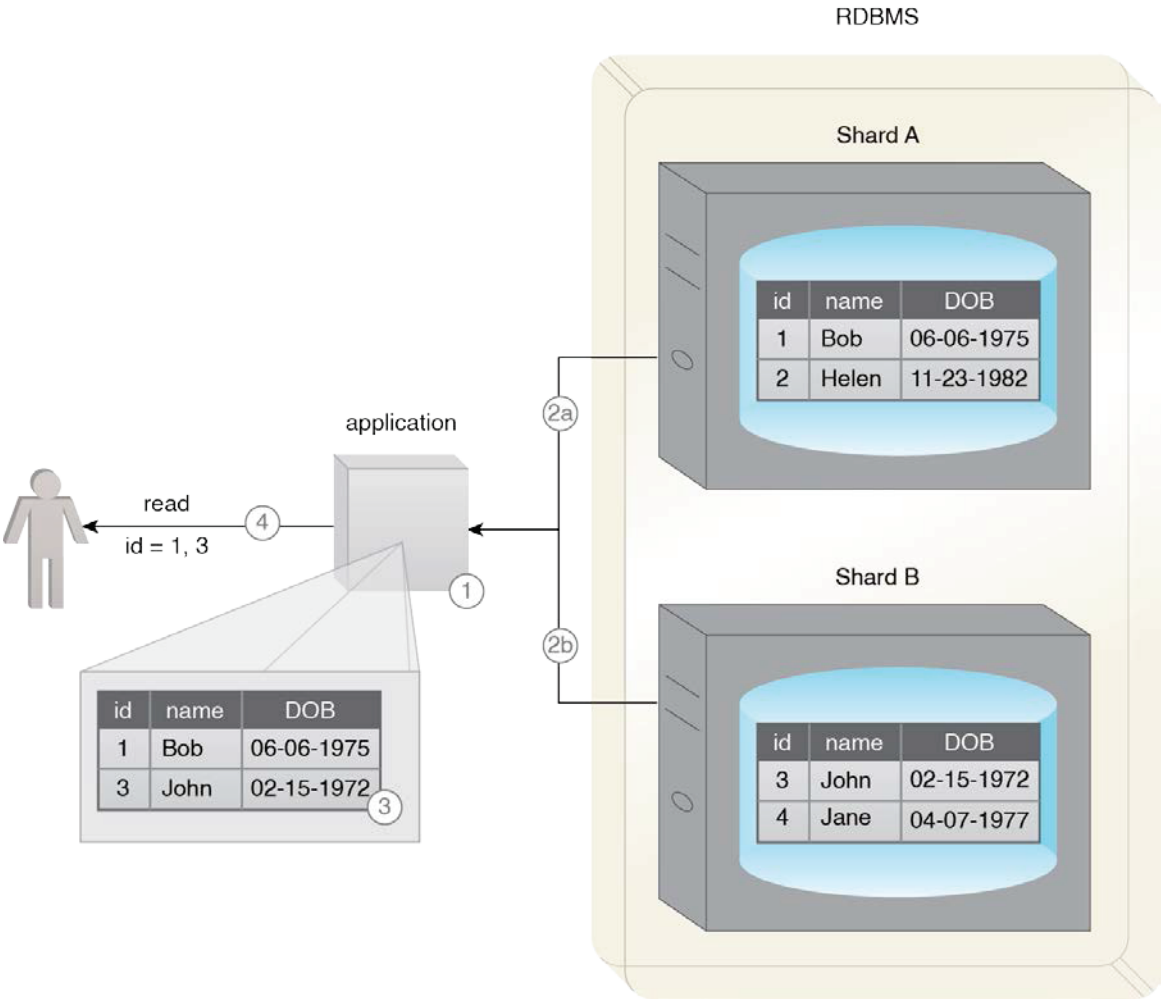


Figure 7.30 – An example of the use of the application logic.

Relational databases generally require data to adhere to a schema. As a result, storage of semi/unstructured data whose schema is not known or keeps changing is not supported. Furthermore, the schema conformance is validated at the time of data insert/update, which introduces overhead and leads to latency.

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. As a result of its shortcomings, a traditional RDBMS is generally not useful as a storage device in a Big Data solution environment.

On-Disk Storage Device: NoSQL

The term *Not only SQL (NoSQL)* refers to technologies used to develop next generation non-relational databases that are highly scalable and fault-tolerant.

Figure 7.31 – The symbol used to represent a NoSQL database.

On-Disk Storage Device: NoSQL Characteristics

Below are some of the principal features of NoSQL storage devices that differentiate them from traditional RDBMSs. This list should only be considered a general guide, as not all NoSQL storage devices exhibit all of these features.

- Schema-less data model – Data can exist in its raw form.
- Scale out rather than scale up – More nodes are added as required rather than replacing the existing one with a better, higher performance node.
- Highly available – Built on cluster-based technologies providing fault tolerance out of box.
- Lower operational costs – Built on Open Source platforms with no licensing costs, and can be deployed on commodity hardware.
- Eventual consistency – Reads across multiple nodes may not be consistent immediately after a write. However, all nodes will eventually be in a consistent state.
- 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. As a result, in consideration of the CAP theorem, NoSQL storage devices are generally AP or CP.
- 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.
- 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 over to multiple nodes.
- Integrated caching – 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.

- Polyglot persistence – The use of NoSQL device storage does not mandate retiring traditional RDBMSs. In fact, both can be used at the same time, thereby supporting polyglot persistence (an approach of persisting data using different types of storage technologies). This is good for developing systems requiring structured as well as semi/unstructured data.
- 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 mapping between application objects and the data stored in the database. Note that graph database storage devices (introduced shortly) are not aggregate-focused.

On-Disk Storage Device: NoSQL Rationale

The emergence of NoSQL storage devices can primarily be attributed to the volume, velocity, and variety characteristics of Big Data datasets.

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 scaling out capability while using inexpensive commodity servers. Furthermore, there may not be licensing costs involved as NoSQL databases generally follow the Open Source development model.

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/network failure.

Variety

A storage device needs to handle different types of 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. At the same time, NoSQL storage devices are able to store schema-less data and incomplete data with the added ability of making schema changes as the data model of the datasets evolve. In other words, NoSQL databases support *schema evolution*.

On-Disk Storage Device: NoSQL Types

NoSQL storage devices can mainly be divided into four types based on the way they store data, as shown in Figures 7.32-7.35:

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

key	value
631	John Smith, 10.0.30.25, Good customer service
365	1001010111011011110111010101101010100111001101O
198	<CustomerId>32195</CustomerId><Total>43.25</Total>

Figure 7.32 – An example of key-value NoSQL storage.

Figure 7.33 – An example of document NoSQL storage.

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

Figure 7.34 – An example of column-family NoSQL storage.

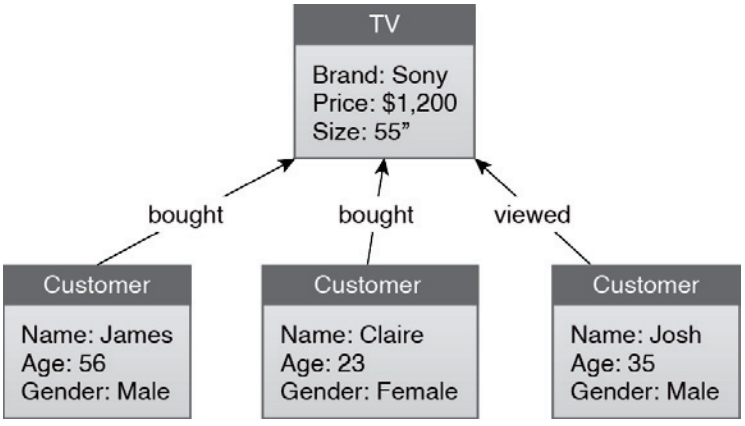


Figure 7.35 – An example of graph NoSQL storage.

NoSQL: 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. The value is opaque to the database, and is essentially stored as a BLOB. The value stored can be any aggregate, ranging from sensor data to videos.

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.

As keys are the only means of retrieving the data, the key is usually appended with the type of the value being saved for easy retrieval. For example, *123_sensor1*.

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, as shown in Figure 7.36. Some implementations support compressing values for reducing the storage footprint. However, this introduces latency at read time, as the data needs to be decompressed first.

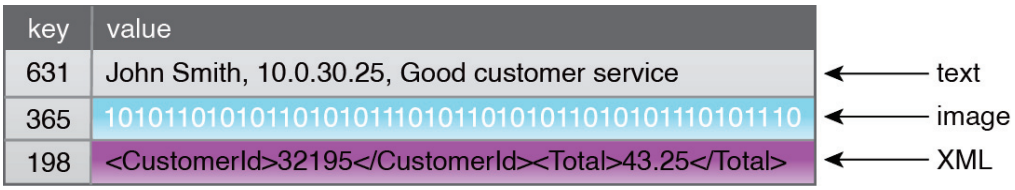


Figure 7.36 – An example of data organized into key-value pairs.

A key-value storage device is appropriate when:

- unstructured data storage is required
- high performance read/writes are required
- the value is fully identifiable via the key alone
- value is a standalone entity that is not dependent on other values
- values generally have a comparatively simple structure or are binary
- query patterns are simple, involving insert, select, and delete operations only
- stored values are manipulated at the application layer

A key-value storage device is inappropriate when:

- applications require searching or filtering data using attributes of the stored value
- relationships exist between different key-value entries
- a group of keys' values need to be updated in a single transaction
- multiple keys require manipulation in a single operation
- schema consistency across different values is required
- update to individual attributes of the value is required

Examples of key-value storage devices include Riak, Redis and Amazon Dynamo DB.

Reading

The *Key-Value Databases* section on pages 81-88 of the *NoSQL Distilled* book that accompanies this module provides further discussion on NoSQL key-value storage devices.

NoSQL: 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 have a complex nested structure, such as an invoice, as shown in Figure 7.37. The document 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. The main differences of document storage devices when compared with key-value storage devices are as follows:

- document storage devices are value-aware
- the stored value is self-describing; the schema can be inferred from the structure of the value
- a select operation can reference a field inside the aggregate value
- a select operation can retrieve a part of the aggregate value
- partial updates are supported; a subset of the aggregate can be updated
- indexes that speed up searches are generally supported

Each document can have a different schema. It is possible to store different types of documents or documents of the same type that have fewer or more fields with respect to each other. Additional fields can be added to a document after the initial insert, thereby manifesting flexible schema support.

It should be noted that document storage devices are not limited to storing data that occurs in the form of actual documents, such as an XML file, but can also be used to store any aggregate that consists of a collection of fields having a flat or a nested schema.

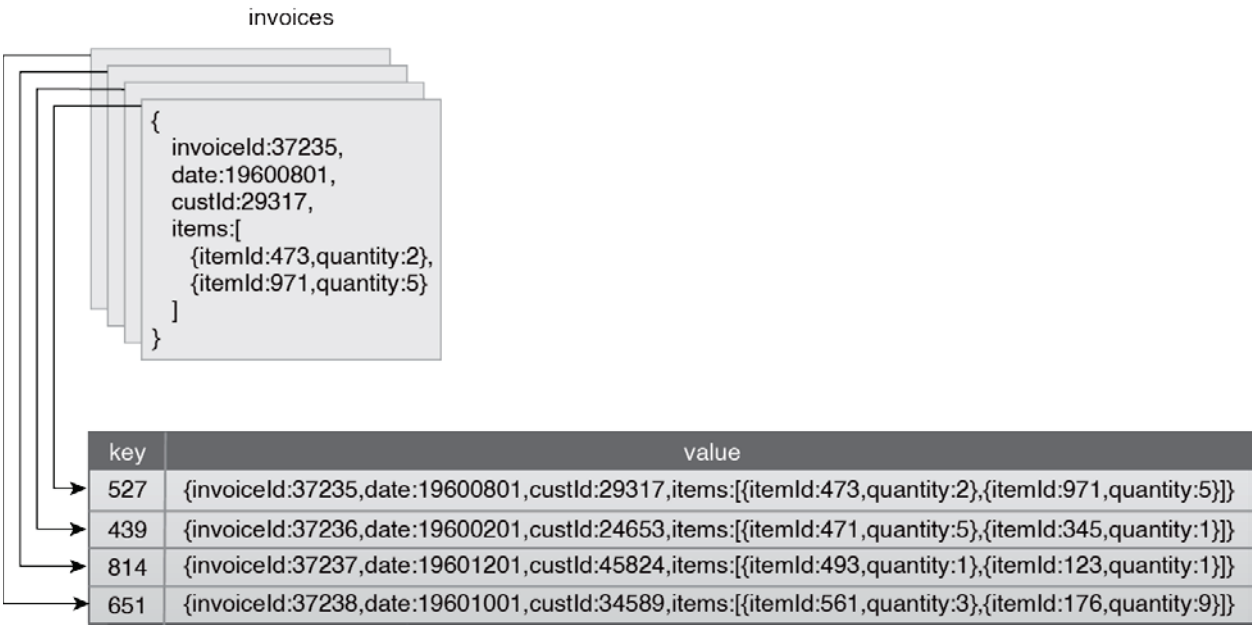
A document storage device is appropriate when:

- 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 document
- searches need to be performed on different fields of the documents
- storing domain objects, such as customers, in 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
- stored value is not self-describing
- binary data needs to be stored

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



key	value
527	{invoiceId:37235,date:19600801,custId:29317,items:[{itemId:473,quantity:2},{itemId:971,quantity:5}]}
439	{invoiceId:37236,date:19600201,custId:24653,items:[{itemId:471,quantity:5},{itemId:345,quantity:1}]}
814	{invoiceId:37237,date:19601201,custId:45824,items:[{itemId:493,quantity:1},{itemId:123,quantity:1}]}
651	{invoiceId:37238,date:19601001,custId:34589,items:[{itemId:561,quantity:3},{itemId:176,quantity:9}]}

Figure 7.37 – An example of a stored value as a document with a complex nested structure.

Reading

The *Document Databases* section on pages 89-98 of the *NoSQL Distilled* book that accompanies this module further discusses NoSQL document storage devices.

NoSQL: 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 (Figure 7.38). 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. 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 storage devices provide fast data access with random read/write capability. They store different column-families in separate physical files, which greatly helps in speeding up queries 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 expiry time for columns after which the configured columns are 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
- efficient use of storage is required when the data consists of sparsely populated rows, as column-family databases only allocate storage space if a column exists for a row (no column, no space)
- 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 that initiate a corresponding restructuring of how column-families are arranged, for example, during proof of concept development

Examples include Cassandra, HBase and Amazon SimpleDB.

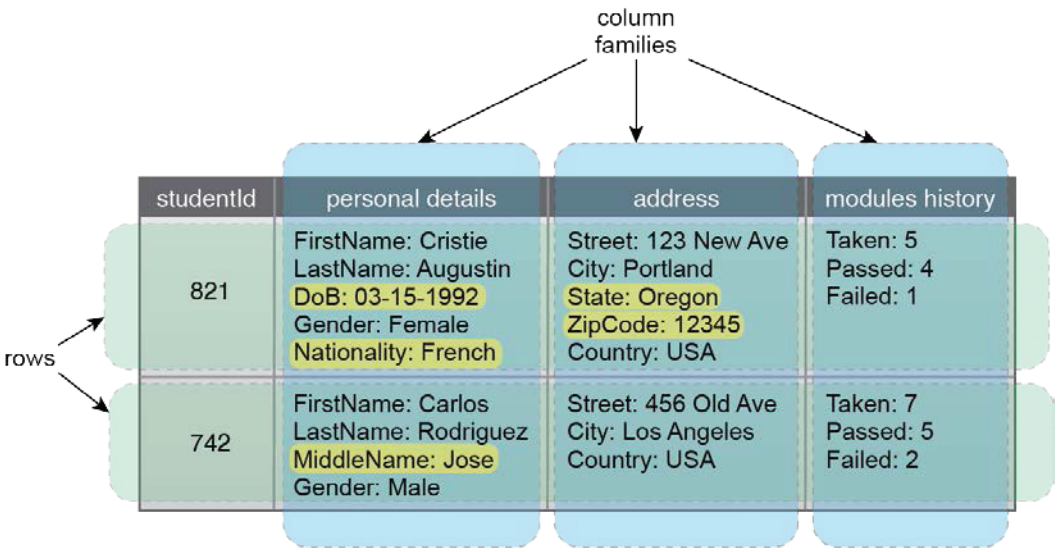


Figure 7.38 – The highlighted columns depict the flexible schema feature supported by the column-family databases, where each row can have a different set of columns.

Reading

The *Column-Family Stores* section on pages 99-109 of the *NoSQL Distilled* book that accompanies this module further discusses NoSQL column-family storage devices.

NoSQL: 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 (Figure 7.40).

Entities are stored as nodes (not to be confused with *cluster nodes*) also called vertices, while the linkages between entities are stored as edges. In RDBMS parlance, each node can be thought of a single row while the edge denotes a join.

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. Multiple edges are similar to defining multiple foreign keys in an RDBMS. 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.

The degree of usefulness of a graph storage device depends on the number and types of edges defined between the nodes. The higher the number and more diverse the edges are, the more diverse types of queries it can handle. As a result, it is important to comprehensively capture the

types of relations that exist between the nodes. This is not only true for existing usage scenarios, but also for exploratory analysis of data.

Graph storage devices generally allow adding new types of nodes without making changes to the database. It 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, as this involves searching for nodes or edges 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 or else queries based on selection of node/edge attributes dominate node traversal queries

Examples include Neo4J, Infinite Graph and OrientDB.

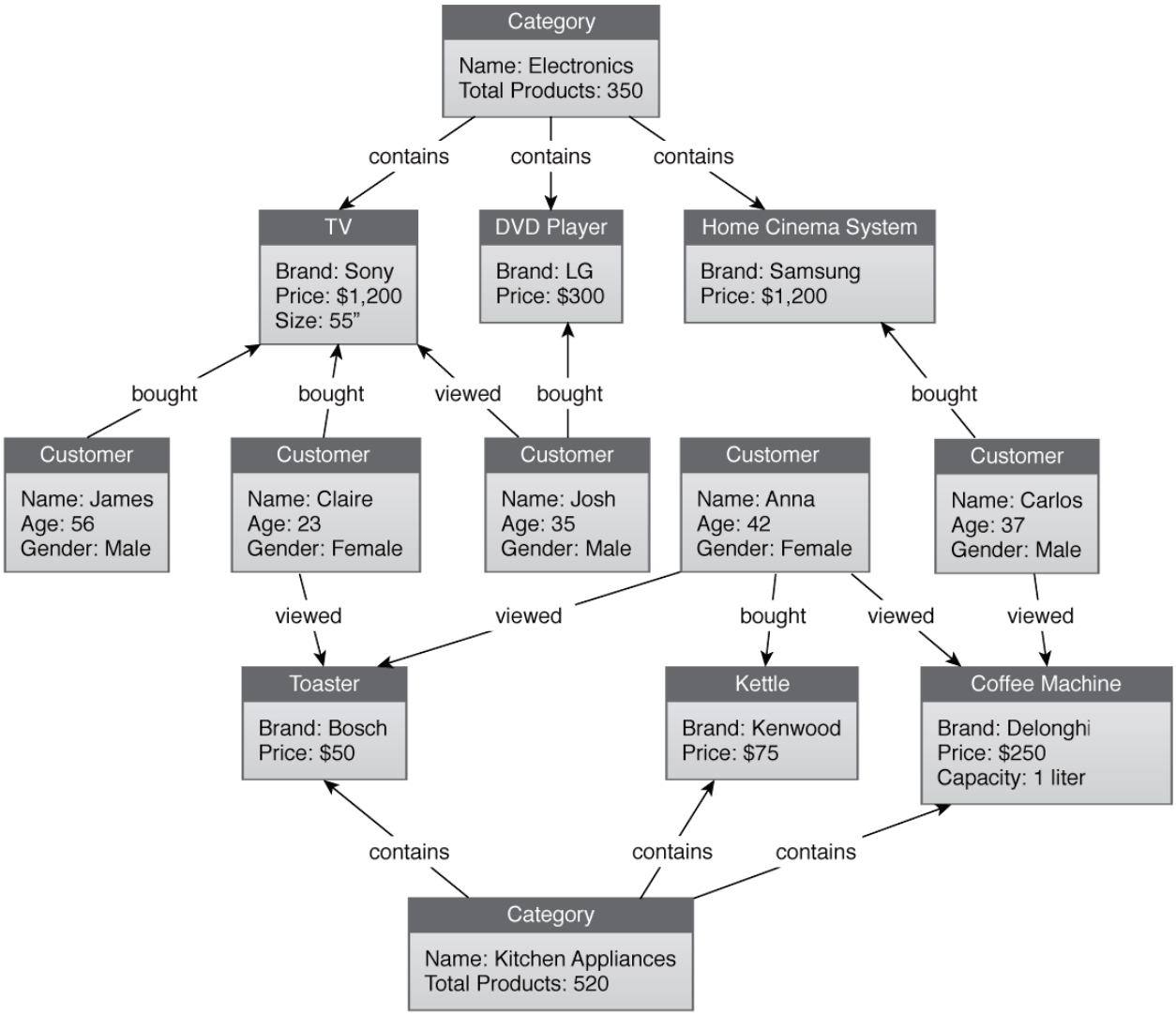


Figure 7.39 – An example of a graph storage device displaying relationships between different entities.

Reading

For further discussion on NoSQL graph storage devices, refer to the *Graph Databases* section on pages 111-121 of the *NoSQL Distilled* book that accompanies this module.

NewSQL

NoSQL storage devices are highly scalable, available, fault-tolerant, and very fast for read/write operations. However, they do not provide the same transaction and consistency support as exhibited by ACID compliant RDBMSs. Following the BASE model, NoSQL storage devices provide eventual consistency rather than immediate, and could therefore be in an inconsistent stage while reaching the state of consistency. As a result, they cannot be used for implementing large scale transactional systems.

NewSQL storage devices combine the ACID properties of RDBMS with the scalability and fault tolerance offered by NoSQL storage devices. They databases generally support SQL compliant syntax for data definition and data manipulation operations, and they often use a relational data model for data storage.

NewSQL databases can be used for developing OLTP systems with very high volumes of transactions, for example a bank. They can also be used for realtime analytics, for example operational analytics, as some implementations are memory based.

As compared to a NoSQL storage device, a NewSQL storage device provides an easier transition from a traditional RDBMS to a highly scalable database due to its support for SQL. Examples include VoltDB, FoundationDB, NuoDB and InnoDB.

Exercise 7.2: Choose the Correct Storage Device

Correctly identify which storage device is best used in each of the following problem statements (note that not all storage devices are used):

- on-disk
- in-memory
- distributed file system
- RDBMS
- key-value
- document
- column-family
- graph

1. Jane needs to store a large number of CCTV video files. Due to their poor quality, all of these files will be processed one after the other using a video enhancing software library. Which type of storage device can Jane use to ensure maximum read throughput to enable fast processing of video files?

   _____

2. Kerry is designing a Big Data application that needs to store large quantities of XML files. Each XML file represents a separate entity consisting of multiple sections where each section has sub-fields. Different sections of the XML file need to be retrieved and updated as part of the processing workflow. Which NoSQL storage device best fulfills Kerry's data storage requirements?

   _____

3. Roger is planning to replace the current relational database with a NoSQL database to store user session data for a popular online store. A user session is identified via the user ID and a timestamp, and stores application-specific data that is append-only. Analysis is performed on user session data that requires grouping each user's session data. Which NoSQL storage device can be used in this scenario?

   _____

4. Mike has been tasked with designing a database for storing physical assets that are geographically spread across the country. Each asset has a basic set of attributes, such as ID, type, and date of manufacture. Each asset is physically connected with a number of other assets. Mike learns that the database will be used heavily by the engineers to find assets that are connected with each other, as well as the distance between two assets. Which storage device can Mike use to address the query requirements of the engineers?

   _____

5. John is designing a web application that stores various pieces of information regarding each customer, such as customer's personal information including address and credit card information, customer's buying history, and comments left on the website for different products. John wants to be able to search customers using customer name in order to update customer records. Various types of customer analyses are expected to be performed. One of the analyses that determine customers' sentiments requires interactive access to text in comments, as well as searching comments left by each customer. Which NoSQL database can John use to enable access to individual fields and provide fast retrieval of comment data?

   _____

Exercise answers are provided at the end of this booklet.

Notes

Notes / Sketches

Big Data Processing Engine Characteristics

Processing Big Data datasets is not the same as processing small data, mainly due to the introduction of semi-structured and unstructured data formats and the large quantity of data involved. Small data refers to predominantly, if not wholly, structured data that is processed by enterprise applications and stored in relational databases.

Owing to the unique Big Data characteristics, processing of Big Data datasets establishes certain prerequisites for processing engines. This section covers some fundamental characteristics that are key to formulating a technology set and choosing the correct software frameworks for Big Data processing.

The following processing engine characteristics are covered:

- distributed/parallel data processing
- schema-less data processing
- multi-workload support
- linear scalability
- redundancy & fault tolerance
- low cost

Distributed/Parallel Data Processing

Owing to the volume characteristic of Big Data and as established in the *On-disk Storage Devices* section, Big Data datasets are generally stored utilizing distributed technologies (distributed file system or NoSQL database).

The very nature of a distributed storage device requires a processing engine that can process data without needing to transfer the data from storage to a computing resource, as with distributed data processing. In support of maximizing the value characteristic of Big Data, it is imperative to employ a processing model based on the divide-and-conquer principle, as with parallel data processing.

Schema-less Data Processing

Big Data datasets come in multiple formats (variety characteristic) and may not conform to any schema, especially unstructured data. Schemas may change over time in an effort to accommodate changing business requirements, or simply because of an application software upgrade. Similarly, more data sources with unknown schemas may need to be added in the future.

The lack of adherence to any particular data model requires flexible processing of Big Data datasets so that they can be processed in raw form without the need to be stored in a particular data model.

Multi-Workload Support

Big Data datasets may arrive thick (volume characteristic) and/or fast (velocity characteristic). Often, it may be feasible to process data offline in batches, such as with overnight report generation. In other cases, the results may be required in realtime, such as with GPS signal processing. These scenarios generate opposite processing workloads: transactional and batch.

In order to achieve maximum value from Big Data datasets, the underlying processing platform may need to support both transactional and batch workloads. A single processing engine may fulfill this requirement, or multiple processing engines may need to be used. Although having the ability to process data both in realtime (transactional) and batch mode is ideal for extracting maximum value out of Big Data datasets, support for both modes may not be required or may not be feasible.

Generally, assembling a batch processing Big Data solution environment is simpler and cheaper when compared to a realtime processing Big Data solution. Consequently, adding support for multi-workload processing should be business driven, involving careful cost-benefit analysis.

Linear Scalability

Owing to the volume and velocity characteristics of Big Data, the processing demand can grow quite sharply with increasing volumes of data arriving at a fast pace. Supporting a distributed processing environment with parallel processing capabilities requires a processing engine that can provide a steady throughput as the data volumes grow.

The processing of Big Data datasets requires a highly scalable processing engine that can be linearly scaled. In the context of processing, linear scalability means that one receives a proportional increase in performance with the addition of more processing nodes.

Realtime business intelligence and analytics can leverage such a linearly scalable processing environment to deliver quicker responses involving complex operations on an entire dataset. Linear scalability is generally achieved by employing a scaling out strategy as it provides a simple, non-disruptive, and cost effective method for increasing processing capacity.

Redundancy & Fault Tolerance

A highly distributed data processing environment with parallel data processing capabilities generally involves complicated architecture. With a horizontally scaled processing architecture, which by design involves a large number of nodes and networking components, the chances of partial system failure increase.

As a system failure in the middle of a long running distributed task would be detrimental to achieving the analytic goals, a processing engine needs to provide fault tolerance so that a partial failure does not render the entire system unavailable and data processing does not need to be started from scratch.

Fault tolerance is generally provided through redundancy. With redundant processing resources, the system can still be available in the event of a partial system failure. Horizontal

scaling lends itself quite useful in this case, as redundancy can generally be increased by simply adding more processing resources.

Low Cost

At the start of a Big Data initiative, the cost for a highly scalable distributed data processing environment that involves a few processing nodes and networking equipment may not be that high. However, over time, as the data volume grows and the types and frequency of analytics being run increase, the requirement for an increased number of processing resources can translate into soaring IT costs involving both software and hardware. This could prove counterproductive to the very reason the Big Data initiative was undertaken, often being to help the business deliver increased value, drive down costs, find new sources of revenue, or establish new service offerings.

Consequently, it is quite important to employ a low cost processing engine whose costs can be kept to a minimum as the demand for more processing resources increases. Use of Open Source software deployed over commodity hardware helps to keep the costs down.

Another aspect of keeping costs down is the ability of the processing engine to take advantage of cloud computing. The on-demand and elastic nature of the cloud helps avoid any up-front capital investment, coupled with faster setup of the data processing solution environment.

Exercise 7.3: Match Terms to Statements

Answer the questions below by filling in each blank with one of the following terms:

- schema-less data processing
- multi-workload support
- low cost
- redundancy and fault tolerance
- distributed/parallel data processing
- linear scalability

1. Which processing engine characteristic demands steady throughput in the face of voluminous data arriving at a fast speed?

   _____

2. Which processing engine characteristic provides support for evolving data models and enables the processing of data in its original form without performing any data model transformations?

   _____

3. Which processing engine characteristic encourages use of Open Source software and cloud computing?

   _____

4. Which processing engine characteristic enables both batch and realtime (transactional) data processing?

   _____

5. Which processing engine characteristic provides availability in the face of system failures?

   _____

6. Which processing engine characteristic enables processing of large amounts of data at the source without the need to transfer data from storage to the computing resource?

   _____

Exercise answers are provided at the end of this booklet.

Notes

Notes / Sketches

Fundamental Big Data Processing

A quick look at the processing engine characteristics reveals that most of them are a function of the underlying computing architecture on which the resulting Big Data platform is based. Processing large amounts of data is not a new phenomenon, and different large scale data processing architectures exist. Big Data processing requires a distributed environment that is capable of processing data in parallel, a characteristic supported by the cluster architecture.

Big Data Processing: Cluster

As introduced in Module 2, a cluster is a group of nodes connected together via a network to process tasks in parallel. A cluster is a centrally managed network of nodes where each node is responsible for a sub-task of a larger problem (Figure 7.40). Clusters enable distributed data processing. Ideally, a cluster comprises low-cost commodity nodes that collectively provide increased processing capacity with inherent redundancy and fault tolerance, as it consists of physically separate nodes.

Clusters are highly scalable, supporting horizontal scaling with linear performance gains. They provide an ideal deployment environment for a processing engine as large datasets can be divided into smaller datasets and then processed in parallel in a distributed manner.

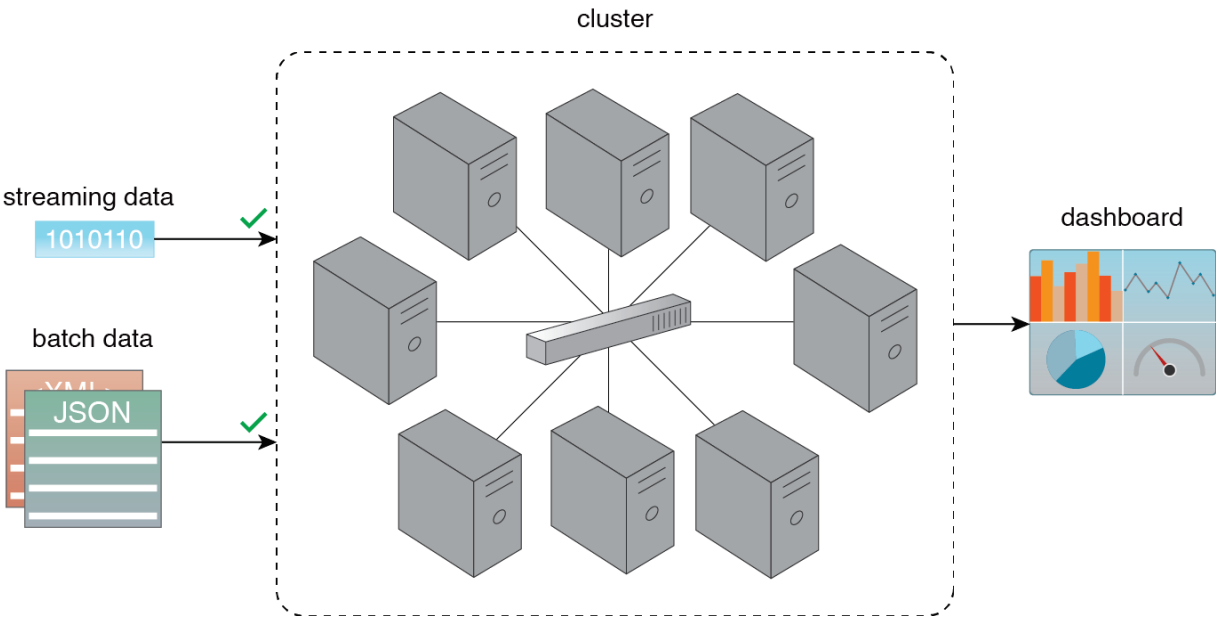


Figure 7.40 – A cluster can be utilized by both a realtime processing engine and a batch processing engine, such as Spark and MapReduce respectively.

In the preceding section, a set of processing engine characteristics were introduced, and earlier in this section, it was established that a Big Data processing environment needs to be based on a cluster architecture that supports distributed/parallel data processing with linear scalability. The upcoming content explores how to make use of the cluster architecture for Big Data

processing. In a cluster architecture, Big Data datasets can either be processed in batch mode or realtime mode using a batch processing engine or a realtime processing engine, respectively.

Big Data Processing: Batch Mode

In batch mode, data is processed offline in batches where the response time could vary from minutes to hours. Data is first persisted to the disk, and only then is it processed. Batch mode generally involves processing a range of large datasets, either on their own or joined together, essentially addressing the volume and variety characteristics of Big Data datasets.

The majority of Big Data processing occurs in batch mode. It is relatively simple, easy to set up, and low in cost compared to realtime mode. Strategic BI, predictive/prescriptive analytics and ETL operations are commonly batch-oriented.

Big Data Processing: Realtime Mode

In realtime mode, data is processed in-memory as it is captured before being persisted to the disk. Response time generally ranges from a few seconds to under a minute. Realtime mode addresses the velocity characteristic of Big Data datasets.

Within Big Data processing, realtime processing is also called event or stream processing as the data either arrives continuously (stream) or at intervals (event). The individual event/stream datum is generally small in size, but its continuous nature results in very large datasets.

Another related term, interactive mode, falls within the category of realtime. Interactive mode generally refers to query processing in realtime. Operational BI/analytics are generally conducted in realtime mode.

NOTE

It is important to understand that although the two processing modes introduced here are discussed in the context of whether the data first gets persisted to the disk or not, the mere fact that the data resides on disk does not mean that data cannot be processed in realtime or near realtime.

Data processing, once the data is saved to disk, can also refer to *query processing* where the processing mode is characterized by the response time.

Batch mode query processing refers to queries that do not instantly provide a result. On the other hand realtime mode query processing refers to queries that instantly provide a result.

Notes

Notes / Sketches

Introducing the MapReduce Processing Engine

MapReduce is a widely used implementation of the batch processing engine mechanism. It is a highly scalable and reliable processing engine based on the principle of divide-and-conquer that provides built-in fault tolerance and redundancy. It divides a bigger problem into a set of smaller problems that are easier and quicker to solve, and has its roots both in distributed computing as well as in parallel computing. MapReduce is a batch-oriented processing engine used to process large datasets using parallel processing deployed over clusters of commodity hardware.

Figure 7.41 – The symbol used to represent a processing engine.

MapReduce does not require the input data to conform to any particular data model. Therefore, it can be used to process schema-less datasets. A dataset is broken down into multiple smaller parts, and operations are performed on each part independently and in parallel. The results from all operations are then summarized to arrive at the answer. Because of the involved coordination overhead, the MapReduce processing engine generally supports batch workloads only. MapReduce is based on Google's research paper on the subject, published in early 2000.

The MapReduce processing engine works differently compared to the traditional distributed processing paradigm. Traditionally, data processing requires moving data from the storage node to the processing node that runs the actual data processing algorithm, which works fine for smaller datasets. However, with large datasets, moving data generally incurs more overhead than the actual data processing.

With MapReduce, the data processing algorithm is instead moved to the nodes that store the data. The data processing algorithm executes in parallel on these nodes, thereby eliminating the need to first move the data. This not only saves network bandwidth but also results in a large reduction in processing time for large datasets, as processing smaller chunks of data in parallel is much faster.

MapReduce Concepts

A single processing run of the MapReduce processing engine is known as a MapReduce job. Each MapReduce job is composed of a map task and a reduce task. Each task consists of multiple stages:

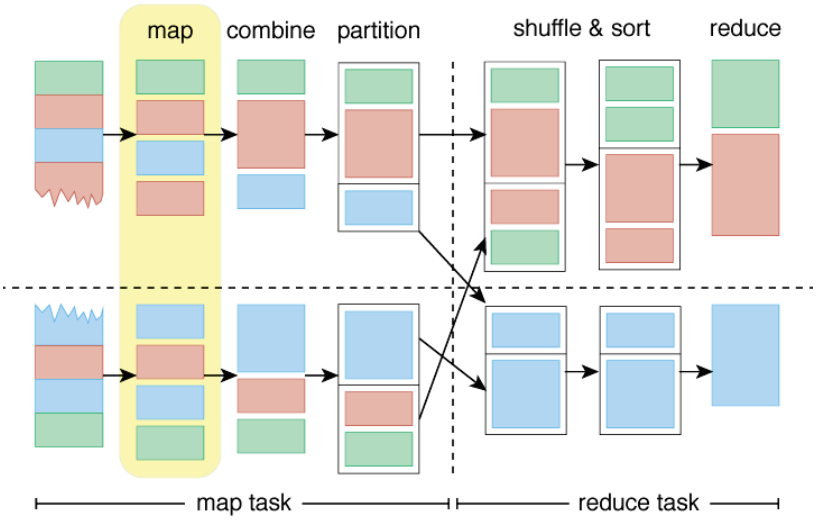
Map task

- Map
- Combine (optional)
- Partition

Reduce task

- Shuffle & Sort
- Reduce

MapReduce: Map



The first stage of MapReduce is known as map, during which the dataset file is divided into multiple smaller splits. Each split is parsed into its constituent records as a key-value pair. The key is usually the ordinal position of the record, and the value is the actual record. For example, *(234, sky is blue)*.

The parsed key-value pairs for each split are then sent to a map function or mapper, with one mapper function per split. The map function executes user-defined logic. Each split generally contains multiple key-value pairs and the mapper is run once for each key-value pair in the split.

The mapper processes each key-value pair as per the user-defined logic and further generates a key-value pair as its output. The output key can either be the same as the input key or a substring value from the input value, or another serializable user-defined object. Similarly, the output value can either be the same as the input value or a substring value from the input value, or another serializable user-defined object.

When all records of the split have been processed, the output is a list of key-value pairs where multiple key-value pairs can exist for the same key. It should be noted that for an input key-value pair, a mapper may not produce any output key-value pair (filtering) or can generate

multiple key-value pairs (demultiplexing). The map stage can be summarized by the equation shown in Figure 7.42.

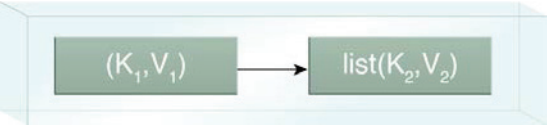
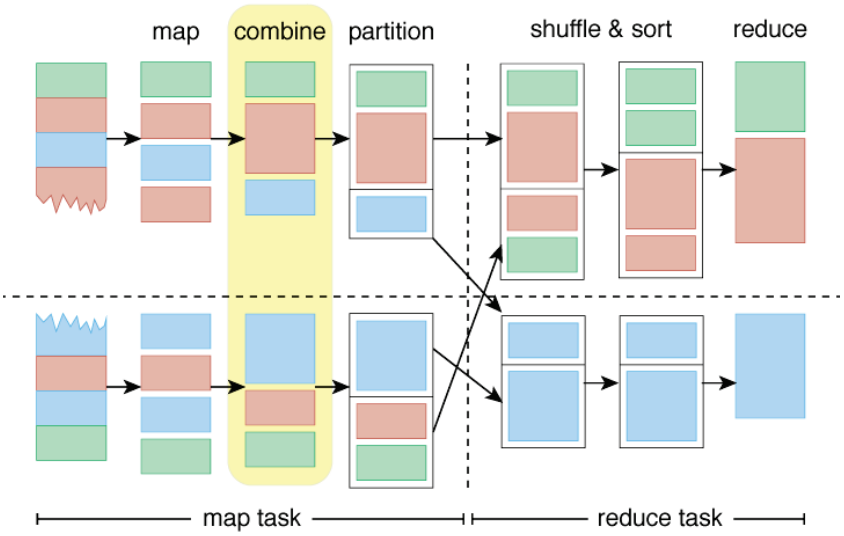


Figure 7.42 – A summary of the map stage.

MapReduce: Combine



Generally, the output of the map function is handled directly by the reduce function. However, map tasks and reduce tasks are mostly run over different nodes. This requires moving data between mappers and reducers that can consume a lot of valuable bandwidth, and directly contributes to processing latency.

With larger datasets, the time taken to move the data between map and reduce stages can exceed the actual processing undertaken by the map and reduce tasks. For this reason, the MapReduce engine provides an optional combine function (combiner) that summarizes a mapper's output before it gets processed by the reducer.

A combiner is essentially a reducer function that locally groups a mapper's output on the same node as the mapper. A reducer function can be used as a combiner function, or a custom user-defined function can be used.

The MapReduce engine combines all values for a given key from the mapper output, creating multiple key-value pairs as input to the combiner where the key is not repeated and the value exists as a list of all corresponding values for that key. The combiner stage is only an optimization stage, and may therefore not even be called by the MapReduce engine.

Note that a combiner should only be specified when its introduction does not affect the net result, as performing an action on a part of the group may not yield the same results when the same action is performed on the whole group. For example, a combiner function will work for finding the largest or the smallest number, but will not work for finding the average of all numbers. The combine stage can be summarized by the equation shown in Figure 7.43.

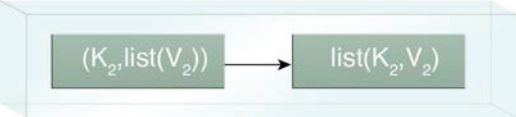
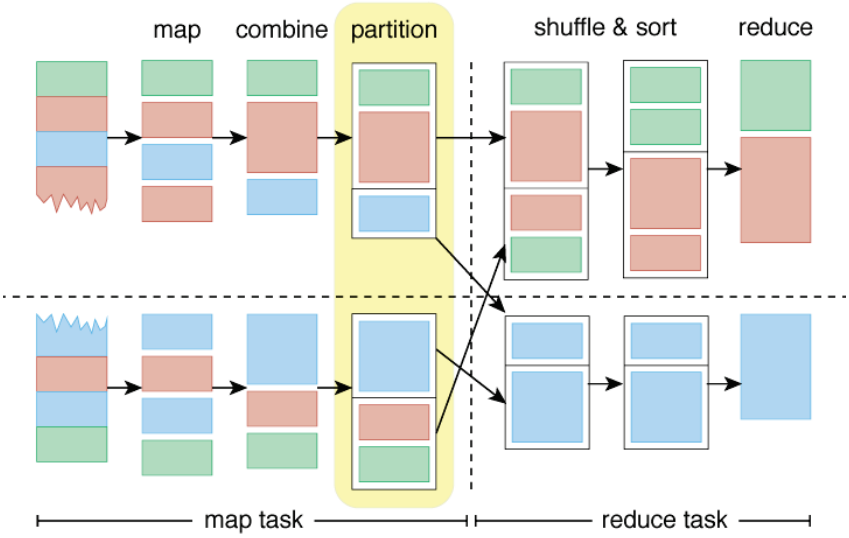


Figure 7.43 – A summary of the combine stage.

MapReduce: Partition



During this stage, if more than one reducer is involved, a partitioner divides the output from the mapper or combiner (if specified and called by the MapReduce engine) into partitions between reducer instances. The number of partitions equals the number of reducers.

Although each partition contains multiple key-value pairs, all records for a particular key are within the same partition. The MapReduce engine guarantees a random and fair distribution between reducers while making sure that all of the same keys across multiple mappers end up with the same reducer instance.

Depending on the nature of the job, certain reducers can sometimes receive a large number of key-value pairs compared to others. As a result of this uneven workload, some reducers will finish earlier than others. Overall, this is less efficient and leads to longer job execution times than if the work was evenly split across reducers. This can be rectified by customizing the partitioning logic in order to guarantee a fair distribution of key-value pairs.

The partition function is the last stage of the map task. It returns the index of the reducer to which a particular partition should be sent. The partition stage can be summarized by the equation in Figure 7.44.

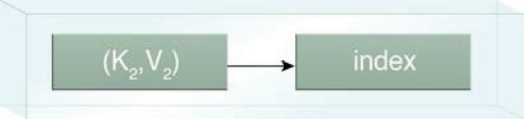
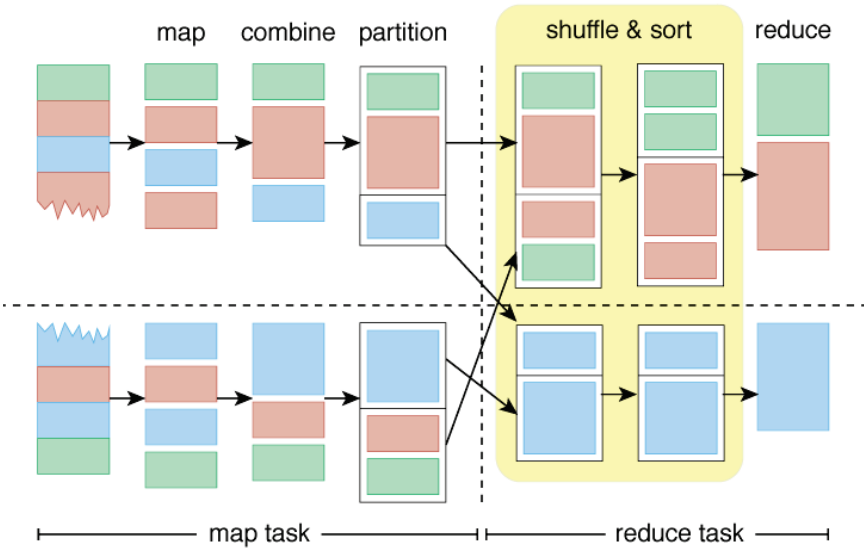


Figure 7.44 – A summary of the partition stage.

MapReduce: Shuffle & Sort



During the first stage of the reduce task, output from all partitioners is copied across the network to the nodes running the reduce task. This is known as shuffling. The list based key-value output from each partitioner can contain the same key multiple times.

Next, the MapReduce engine automatically groups and sorts the key-value pairs according to the keys so that the output contains a sorted list of all input keys and their values with the same keys appearing together. The way in which keys are grouped and sorted can be customized.

The MapReduce engine then merges each group of keys together before the shuffle and sort output is processed by the reducer function. This merge creates a single key-value pair per group, where key is the group key and the value is the list of all group values. This stage can be summarized by the equation in Figure 7.45.

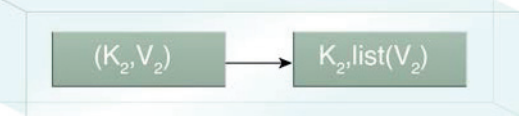
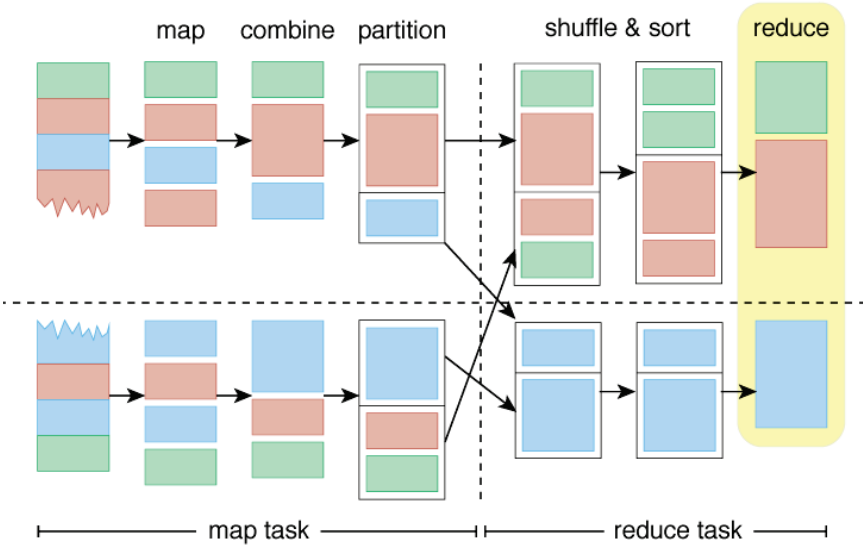


Figure 7.45 – A summary of the shuffle and sort stage.

MapReduce: Reduce



Reduce is the final stage of the reduce task. Depending on the user-defined logic specified in the reduce function (reducer), the reducer will either further summarize its input or will emit the output without making any changes. Therefore, for each key-value pair that a reducer receives, the list of values stored in the value part of the pair is processed and another key-value pair is written out.

The output key can either be the same as the input key or a substring value from the input value, or another serializable user-defined object. The output value can either be the same as the input value or a substring value from the input value, or another serializable user-defined object.

Note that just like the mapper, for the input key-value pair, a reducer may not produce any output key-value pair (filtering) or can generate multiple key-value pairs (demultiplexing). The output of the reducer, that is the key-value pairs, is then written out as a separate file – one file per reducer.

To view the full output from the MapReduce job, all the file parts must be combined. The number of reducers can be customized. It is also possible to have a MapReduce job without a reducer, for example when performing filtering.

Note that the output signature (key-value types) of the map function should match that of the input signature (key-value types) of the reduce/combine function. The reduce stage can be summarized by the equation in Figure 7.46.

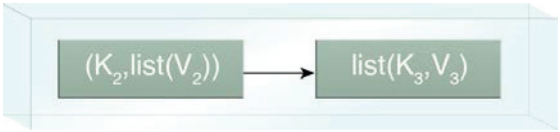


Figure 7.46 – A summary of the reduce stage.

MapReduce: Example

The following steps are shown in Figure 7.47:

1. a, b. The input (sales.txt) is divided into two splits.
2. a, b. Two map tasks running on two different nodes, Node A and Node B, extract product and quantity from the respective split's records in parallel. The output from each map function is a key-value pair where product is the key while quantity is the value.
3. a, b. The combiner then performs local summation of product quantities.
4. a, b. As there is only one reduce task, no partitioning is performed.
5. The output from the two map tasks is then copied to a third node, Node C, that runs the shuffle stage as part of the reduce task.
6. The sort stage then groups all quantities of the same product together as list.
7. Like the combiner, the reduce function then sums up the quantities of each unique product in order to create the output.

NOTE
Note that some NoSQL storage devices provide MapReduce support for batch processing out of box. Making use of clustered deployment of NoSQL storage devices, the MapReduce processing engine distributes the query processing across multiple nodes. This is generally achieved via the provision of map and reduce constructs that the user then needs to implement via the respective API of the NoSQL storage devices.

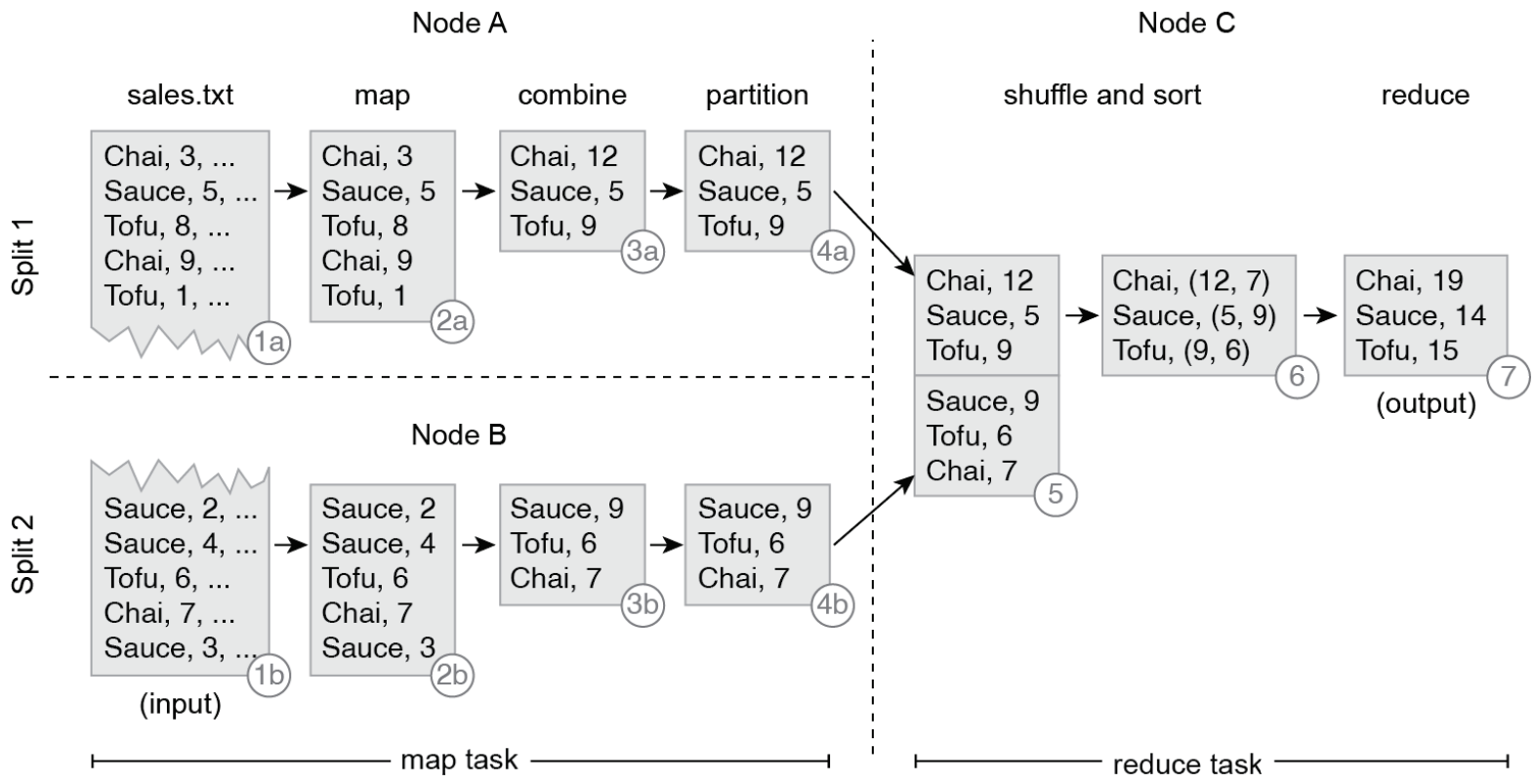


Figure 7.47 – An example of MapReduce.

Exercise 7.4: Organize the MapReduce Stages

Organize the following MapReduce stages into the correct order:

shuffle and sort	1. _____
partition	2. _____
map	3. _____
reduce	4. _____
combine	5. _____

Exercise answers are provided at the end of this booklet.

Notes

Notes / Sketches

Fundamental MapReduce Algorithm Design

Unlike traditional programming models, MapReduce follows a distinct programming model. In order to understand how algorithms can be designed or adapted to the MapReduce programming model, its design principle first needs to be explored.

As described earlier, MapReduce works on the principle of divide-and-conquer. However, it is important to understand the semantics of this principle in the context of MapReduce. The divide-and-conquer principle can generally be achieved using one of the following approaches:

- task parallelism
- data parallelism

Task Parallelism

Task parallelism refers to the parallelization of data processing by dividing a task into sub-tasks and running each sub-task on a separate processor, generally on a separate node in a cluster (Figure 7.48). Each sub-task generally executes a different algorithm, with its own copy of the same data or different data as its input, in parallel. Generally, the output from multiple sub-tasks is joined together to obtain the final set of results.

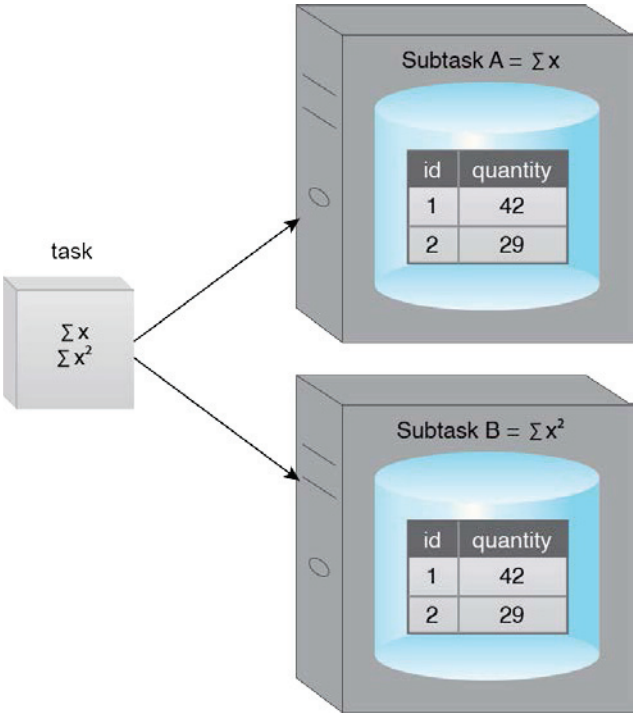


Figure 7.48 – A task is split into two subtasks, Subtask A and Subtask B, which are then run on two different nodes on the same dataset.

Data Parallelism

Data parallelism refers to the parallelization of data processing by dividing a dataset into multiple sub-datasets and processing each sub-dataset in parallel (Figure 7.49). Different sub-datasets are spread across multiple nodes and are processed using the same algorithm. Generally, the output from each processed sub-dataset is joined together to obtain the final set of results.

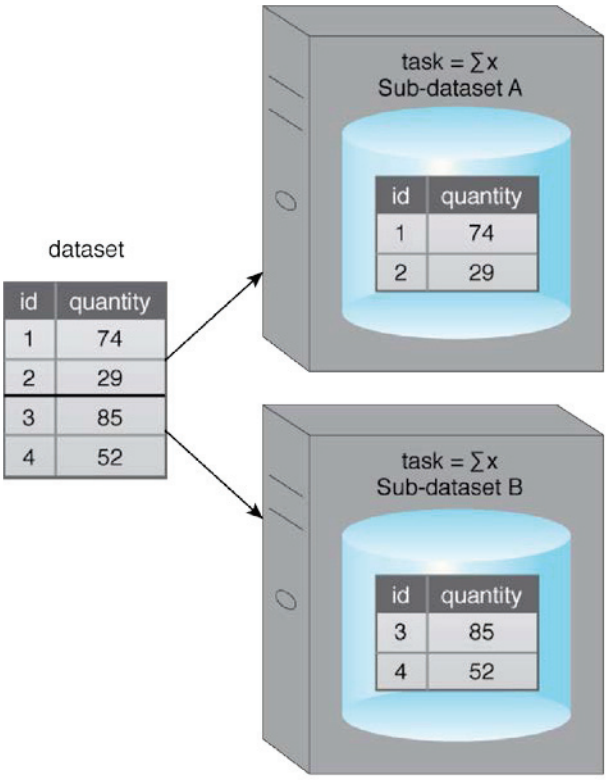


Figure 7.49 – A dataset is divided into two sub-datasets, Sub-dataset A and Sub-dataset B, which are then processed on two different nodes using the same function.

MapReduce Algorithm Design

Within Big Data environments, the same task generally needs to be performed repeatedly on a data unit, such as a record, where the complete dataset is distributed across multiple locations due to its large size. MapReduce addresses this requirement by employing the data parallelism approach, where the data is divided into splits. Each split is then processed by its own instance of the map function, which contains the same processing logic as the other map functions.

The majority of traditional algorithmic development follows a sequential approach where operations on data are performed one after the other in such a way that subsequent operation is dependent on its preceding operation.

In MapReduce, operations are divided among the map and reduce functions. Map and reduce tasks are independent, and in turn, run isolated from each other. Furthermore, each instance of a map or reduce function runs independently of other instances.

Function signatures in traditional algorithmic development are generally not constrained. In MapReduce, the map and reduce function signatures are constrained to a set of key-value pairs that are the only way through which a map function communicates with a reduce function. Apart from this, the logic in the map function is dependent on how records are parsed, which further depends on what constitutes a logical data unit within the dataset.

For example, each line in a text file generally represents a single record. However, it may be that a set of two or more lines actually constitute a single record (Figure 7.50). Furthermore, the logic within the reduce function is dependent on the output of the map function, particularly which keys were emitted from the map function as the reduce function receives a unique key with a consolidated list of all of its values. It should be noted that in some scenarios, such as with text extraction, a reduce function may not be required.

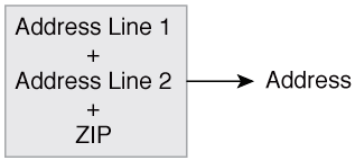


Figure 7.50 – An example where three lines constitute a single record.

MapReduce Algorithm Design: Considerations

The key considerations when developing a MapReduce algorithm can be summarized as follows:

- Relatively simplistic algorithmic logic, such that the required result can be obtained by applying the same logic to different portions of a dataset in parallel, and then aggregating the results in some manner.
- Availability of the dataset in a distributed manner partitioned across a cluster so that multiple map functions can process different subsets of datasets in parallel.
- Understanding of the data structure within the dataset so that a meaningful data unit (a single record) can be chosen.
- Dividing algorithmic logic into map and reduce functions so that the logic in the map function is not dependent on the complete dataset, as only data within a single split is available.
- Emitting the correct key from the map function along with all the required data as value because the reduce function's logic can only process those values that were emitted as part of the key-value pairs from the map function.
- Emitting the correct key from the reduce function along with the required data as value because the output from each reduce function becomes the final output of the MapReduce algorithm.

Exercise 7.5: Fill in the Blanks

1. The MapReduce framework is based on the principle of _____.

2. _____ and _____ are the two approaches generally used to achieve the divide-and-conquer principle.

3. _____ refers to the parallelization of data processing by dividing a task into sub-tasks and running each sub-task on a separate processor, generally on a separate node in a cluster.

4. _____ refers to the parallelization of data processing by dividing a dataset into multiple sub-datasets and processing each sub-dataset in parallel.

5. The MapReduce framework addresses the requirement of repeated execution of the same task on distributed data by employing the _____ approach.

6. In MapReduce, the logic in the reduce function is dependent on the output of the _____ function.

7. The MapReduce framework requires the dataset to be _____ across the cluster so that multiple _____ functions can process sub-datasets in parallel.

8. With the MapReduce algorithm, logic in the map function should not be dependent on the complete dataset as only data within a _____ is available.

Exercise answers are provided at the end of this booklet.

Notes

Notes / Sketches

Exercise Answers

Exercise 7.1 Answers

1. **Sharding** is the process of horizontally partitioning a large dataset into a collection of smaller, more manageable datasets called **shards**, spread across multiple nodes.

2. In sharding, for data partitioning, **query patterns** need to be taken into account so that shards themselves do not become performance bottlenecks.

3. **Replication** makes multiple copies of a dataset, called **replicas**, and stores them across multiple nodes.

4. The two methods for implementing replication include **master-slave** and **peer-to-peer** replication.

5. In **master-slave** replication, the **master** node is the single point of contact for all writes while data can be read from any **slave** node.

6. In **peer-to-peer** replication, there are no **master** and **slave** nodes, and all nodes, called **peers**, operate at the same level.

7. Sharding and replication can be combined to improve on the limited **fault tolerance** offered by sharding, while benefiting from the increased **availability** and **scalability** of replication.

8. The CAP theorem says that a distributed system can only support two out of three properties, which are **consistency**, **availability** and **partition tolerance**.

9. In the context of the CAP theorem, relational databases support **consistency** and **availability**.

10. In the context of database design, BASE stands for **basically available**, **soft state** and **eventual consistency**.

Exercise 7.2 Answers

1. distributed file system
2. document
3. key-value
4. graph
5. column-family

Exercise 7.3 Answers

1. linear scalability
2. schema-less data processing
3. low cost
4. multi-workload support
5. redundancy and fault tolerance
6. distributed/parallel data processing

Exercise 7.4 Answers

1. map
2. combine
3. partition
4. shuffle and sort
5. reduce

Exercise 7.5 Answers

1. The MapReduce framework is based on the principle of **divide-and-conquer**.

2. **Task parallelism** and **data parallelism** are the two approaches generally used to achieve the divide-and-conquer principle.

3. **Task parallelism** refers to the parallelization of data processing by dividing a task into sub-tasks and running each sub-task on a separate processor, generally on a separate node in a cluster.

4. **Data parallelism** refers to the parallelization of data processing by dividing a dataset into multiple sub-datasets and processing each sub-dataset in parallel.

5. The MapReduce framework addresses the requirement of repeated execution of the same task on distributed data by employing the **data parallelism** approach.

6. In MapReduce, the logic in the reduce function is dependent on the output of the **map** function.

7. The MapReduce framework requires the dataset to be **partitioned** across the cluster so that multiple **map** functions can process sub-datasets in parallel.

8. With the MapReduce algorithm, logic in the map function should not be dependent on the complete dataset as only data within a **single split** is available.

Exam B90.07

The course you just completed corresponds to Exam B90.07, which is an official exam that is part of the Big Data Science Certified Professional (BDSCP) program.

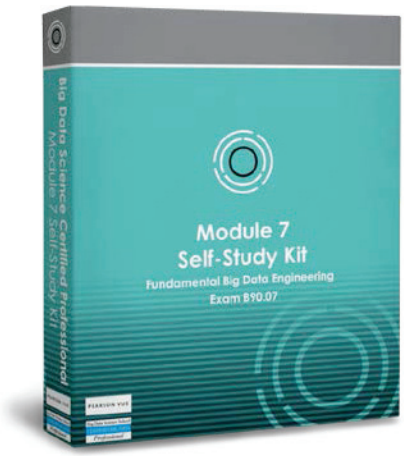
This exam can be taken at Pearson VUE testing centers worldwide or via Pearson VUE Online Proctoring, which enables you to take exams from your home or office workstation with a live proctor. For more information, visit:

www.bigdatascienceschool.com/exams/

www.pearsonvue.com/arcitura/

www.pearsonvue.com/arcitura/op/ (Online Proctoring)

Module 7 Self-Study Kit



An official BDSCP Self-Study Kit is available for this module, providing additional study aids and resources, including a separate self-study guide, Audio Tutor CDs and flash cards.

Note that versions of this self-study kit are available with and without a Pearson VUE exam voucher for Exam B90.07.

For more information, visit:
www.bigdataselfstudy.com

Contact Information and Resources

AITCP Community

Join the growing international Arcitura IT Certified Professional (AITCP) community by connecting on official social media platforms: LinkedIn, Twitter, Facebook, and YouTube.

Social media and community links are accessible at:

- www.arcitura.com/community
- www.servicetechbooks.com/community

General Program Information

For general information about the BDSCP program and Certification requirements, visit:

www.bigdatascienceschool.com and www.bigdatascienceschool.com/matrix/

General Information about Course Modules and Self-Study Kits

For general information about BDSCP Course Modules and Self-Study Kits, visit:

www.bigdatascienceschool.com and www.bigdataselfstudy.com

Pearson VUE Exam Inquiries

For general information about taking BDSCP Exams at Pearson VUE testing centers or via Pearson VUE Online Proctoring, visit:

www.pearsonvue.com/arcitura/
www.pearsonvue.com/arcitura/op/ (Online Proctoring)

Public Instructor-Led Workshop Schedule

For the latest schedule of instructor-led BDSCP workshops open for public registration, visit:

www.bigdatascienceschool.com/workshops

Private Instructor-Led Workshops

Certified trainers can deliver workshops on-site at your location with optional on-site proctored exams. To learn about options and pricing, contact:

info@arcitura.com

or

1-800-579-6582

Becoming a Certified Trainer

If you are interested in attaining the Certified Trainer status for this or any other Arcitura courses or programs, learn more by visiting:

www.arcitura.com/trainerdevelopment/

General BDSCP Inquiries

For any other questions relating to this Course or any Module, Exam, or Certification that is part of the BDSCP program, contact:

info@arcitura.com

or

1-800-579-6582

Automatic Notification

To be automatically notified of changes or updates to the BDSCP program and related resource sites, send a blank message to:

notify@arcitura.com

Feedback and Comments

Help us improve this course. Send your feedback or comments to:

info@arcitura.com