Big Data Science Certified Professional (BDSCP) Curriculum

Module 1 - Fundamental Big Data	(Exam B90.01)
Module 2 - Big Data Analysis & Technology Concepts	(Exam B90.02)
Module 3 - Big Data Analysis & Technology Lab	(Exam B90.03)
Module 4 - Fundamental Big Data Analysis & Science	(Exam B90.04)
Module 5 - Advanced Big Data Analysis & Science	(Exam B90.05)
Module 6 - Big Data Analysis & Science Lab	(Exam B90.06)
Module 7 - Fundamental Big Data Engineering	(Exam B90.07)
Module 8 - Advanced Big Data Engineering	(Exam B90.08)
Module 9 - Big Data Engineering Lab	(Exam B90.09)
Module 10 - Fundamental Big Data Architecture	(Exam B90.10)
Module 11 - Advanced Big Data Architecture	(Exam B90.11)
Module 12 - Big Data Architecture Lab	(Exam B90.12)
Module 13 - Fundamental Big Data Governance	(Exam B90.13)
Module 14 - Advanced Big Data Governance	(Exam B90.14)
Module 15 - Big Data Governance Lab	(Exam B90.15)

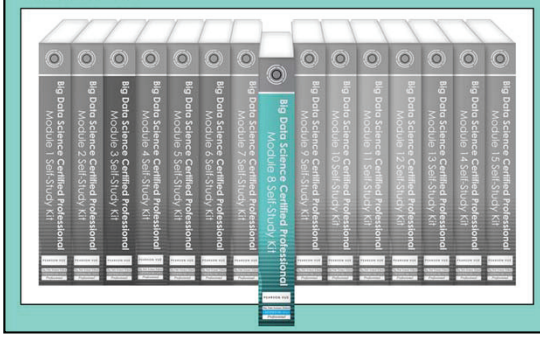
BDSCP Certification Requirements

Big Data Science School
CERTIFIED BIG DATA
Professional

Certified Big Data Science Professional
1, 2, 3

Certified Big Data Scientist
1, 2, 4, 5, 6

Certified Big Data Consultant
1, 2, 3, 4, 7



Certified Big Data Engineer
1, 2, 7, 8, 9

Certified Big Data Architect
1, 2, 10, 11, 12

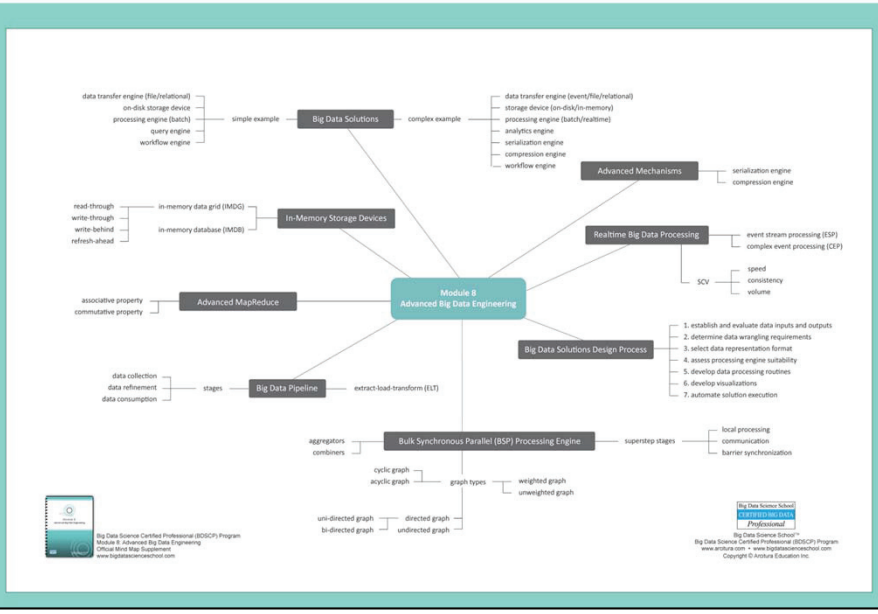
Certified Big Data Governance Specialist
1, 2, 13, 14, 15

For complete descriptions of certifications, visit:
www.bigdatascienceschool.com

Module 8 Overview

- Advanced Big Data Engineering Mechanisms
- In-Memory Storage Devices
- Polyglot Persistence
- Realtime Big Data Processing
- Advanced MapReduce Algorithm Design
- The Bulk Synchronous Parallel Processing Engine
- Big Data Pipeline & ELT
- Big Data Solution Design

Mind Map Poster



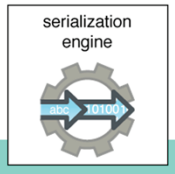
Advanced Big Data Engineering Mechanisms

Overview

Module 2 introduced the fundamental Big Data mechanisms. Module 8 introduces two additional advanced mechanisms that are part of a typical Big Data solution environment. They are:

- Serialization Engine
- Compression Engine

Serialization Engine



- A serialization engine provides the ability to serialize and deserialize data in a Big Data platform.
- *Serialization* is the process of transforming objects or data entities into bytes for persistence (in-memory or on disk) or transportation from one machine to another over a network. The opposite transformation process from bytes to objects or data entities is called *deserialization*.
- Serialization provides means for structuring raw bytes into a format that can easily be manipulated.

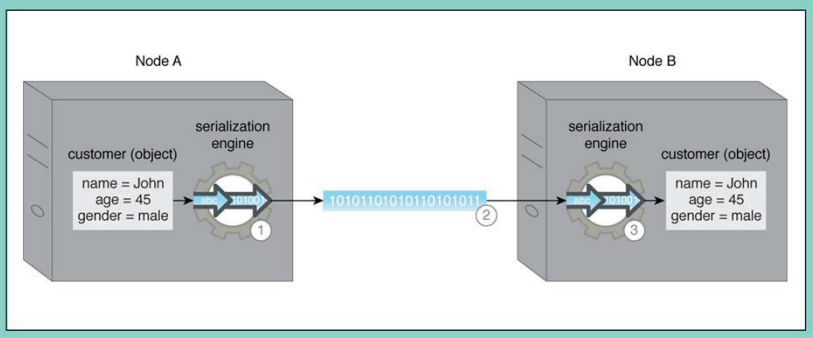
Serialization Engine

- The serialized bytes can either be encoded using a *binary* format or a *plain-text* format.
- Different serialization engines provide different levels of speed, size reduction, extensibility and interoperability.
- Ideally, a serialization engine should serialize/deserialize data at a fast speed with maximum size reduction, be amenable to future changes, and work with a variety of data producers and consumers.
- However, depending on data processing requirements, a choice may need to be made between a fast serialization engine and one that provides data in a more compressed form.

Serialization Engine

- A fast serialization engine requires fewer processing resources but outputs a higher volume of data. This choice is good when fewer processing resources are available and the network is fast.
- A serialization engine that supports a high level of compression requires greater processing resources but outputs a lower volume of data and is generally slow. This choice is good when more processing resources are available and the network is slow.

Serialization Engine Example



Node A needs to send a customer object to Node B. A serialization engine at Node A (1) serializes the object using a binary format that is then sent to Node B over the network (2). At Node B, another serialization engine (3) deserializes the customer object into its original form.

Serialization Engine

- Just as on-wire data needs to be transmitted quickly with a small footprint, the same considerations apply to on-disk/in-memory data storage.
- When serializing data for a data storage scenario, an appropriate serialization engine needs to be chosen.
- The choice depends on the data persistence requirements, as some serialization engines may be more extensible and interoperable than others.
- Furthermore, some serialization engines may be more suitable for serializing binary data, such as images, than serializing textual data, such as log files.

Serialization Engine

- Some examples of serialization engines in the context of Hadoop include Writables, Thrift, SequenceFile, Protocol Buffers and Avro.
- Storage devices often provide raw byte storage without enforcing any structure via a schema. Data is stored as key-value pairs, in key-value NoSQL storage devices and in-memory grids (IMDGs). Distributed file systems also enable storing values as file-based raw byte storage where the file name acts as the key.
- Traditional relational databases and relational in-memory databases (IMDBs) require a schema.
- IMDG and IMDB are introduced in the *In-memory Storage Devices* section.

Serialization Engine

- Consequently, data needs to be serialized using a suitable encoding as working with raw bytes is not only inconvenient but can also introduce interoperability issues. For example, different clients developed using different technologies may be unable to interpret data stored by the other.
- For distributed file system storage devices, serialization needs to be applied to the raw data using a serialization engine.
- To achieve high throughput, chunks of data are generally stored in the same physical file where they are addressed as individual key-value pairs.
- Size, speed, interoperability and seek time of individual key-value pairs inside the file need to be considered when choosing between different serialization engines.

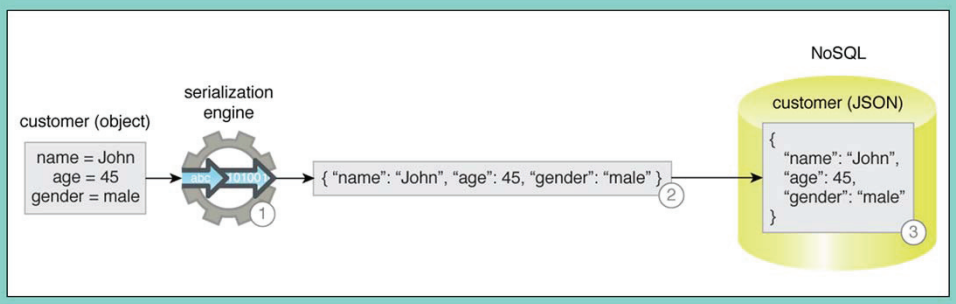
Serialization Engine

- With key-value storage devices, although they provide raw binary storage, some serialization is generally applied for more structured data storage.
- Serialization may be applied automatically by key-value storage devices with support for applying custom serialization.

Serialization Engine

- Document storage devices generally auto-serialize data using a format like JSON or BSON, which allows searching to be performed on individual fields of the document.
- Column-family storage devices store data as a byte array, and consequently, a supported serialization engine is required for serializing data.
- In a graph storage device, the nodes and edges are mainly lightweight objects generally consisting of string-based attributes that are automatically serialized. If using binary types as attributes, manual serialization needs to be performed.

Serialization Engine Example



A customer object needs to be persisted to the disk. A serialization engine (1) serializes the customer object using a JSON plain-text format (2). The customer object is then persisted to a NoSQL document database (3).

Compression Engine

- A compression engine provides the ability to compress and decompress data in a Big Data platform.
- Compression is the process of compacting data in order to reduce its size, whereas decompression is the process of uncompacting data in order to bring the data back to its original size.
- In Big Data environments, due to the volume, velocity and variety characteristics of Big Data datasets, there is a requirement to store ever increasing amounts of data.
- For both on-disk and in-memory storage devices, although the cost has been plummeting and the storage capacity has been increasing, the reduction in cost and increase in storage capacity is disproportionate to the corresponding increase in Big Data datasets' volume.

Compression Engine

- Another factor that makes storage space more valuable and scarce is the way storage devices remain highly available and fault-tolerant through data redundancy.
- Furthermore, unlike traditional data processing environments where historical data is archived to offline storage, in Big Data environments historic data is kept online and can provide more value through the application of deep analytics for finding hidden insights.
- As a result, data compression is required in Big Data environments so that more data can be persisted to a storage device in order to save valuable disk/memory space and to keep costs to a minimum.

Compression Engine

- Compression also helps to achieve faster data transmission rates across the network, as discussed under the serialization engine.
- Different compression engines may provide different levels of data compression.
- When choosing a compression engine, it is important to understand the relationship between compression speed, compactness level and processing resources.

Compression Engine

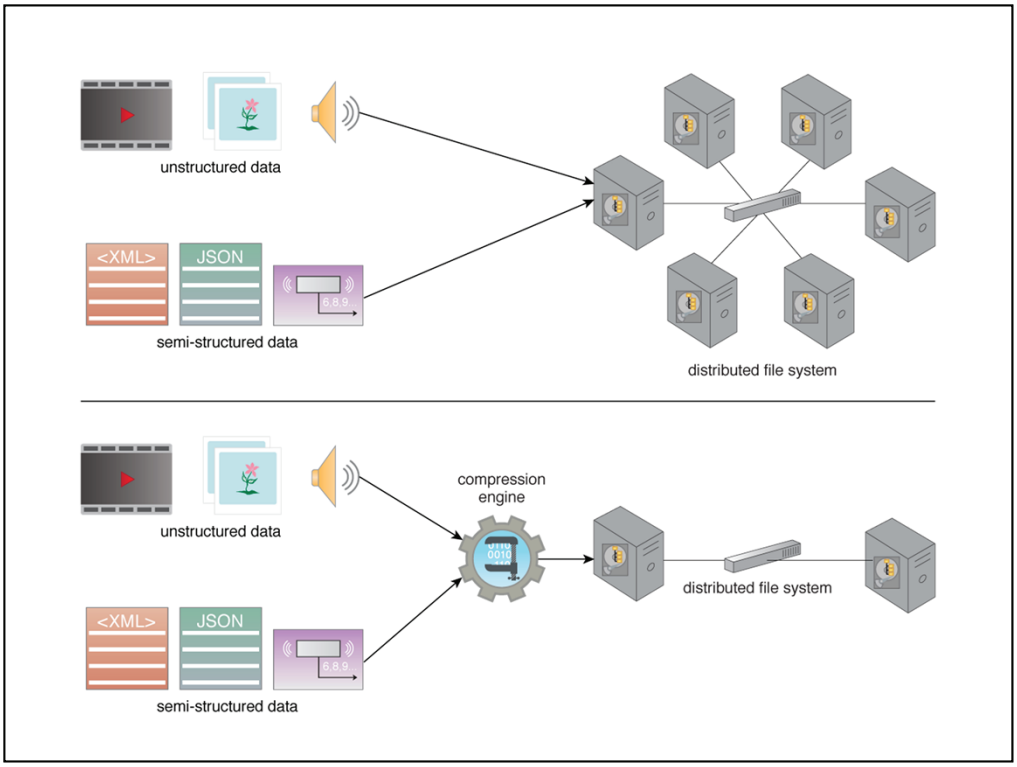
- A fast compression engine will generally provide less compact data and require fewer processing resources. A slow compression engine will generally provide more compact data, but require more processing resources.
- Some compression engines may provide faster decompression than compression.
- Another factor worth considering is whether the compression engine provides the ability to create splits from a file stored on a distributed file system.

Compression Engine

- Recall from Module 7 that, as part of the first stage of MapReduce, the file is divided into multiple smaller splits where each split is sent to a separate mapper.
- If data has been compressed using a compression engine that does not support splitting, the file cannot be split and a single mapper will process the entire file, thereby forgoing the benefits of distributed/parallel data processing.
- Generally, a reduction of 70% to 80% in data size is possible when a compression engine is employed.
- Examples of compression engines include gzip, bzip2, LZO and LZ4.

Compression Engine: Example

- A large amount of semi/unstructured data needs to be stored in a distributed file system.
- In the top diagram on the following page, storing data in its uncompressed form requires six disks.
- In the bottom diagram, a compression engine is used to compress the data. As a result, only two disks are required to store the same amount of data in compressed form.


unstructured data
<XML>
JSON
semi-structured data
distributed file system
unstructured data
compression engine
<XML>
JSON
semi-structured data
distributed file system

Exercise

Exercise 8.1

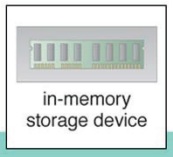
Fill in the Blanks

In-Memory Storage Devices

Note

Any reference to in-memory storage devices in this module implicitly refers to cluster-based in-memory storage devices, spanning over multiple machines as opposed to a single machine.

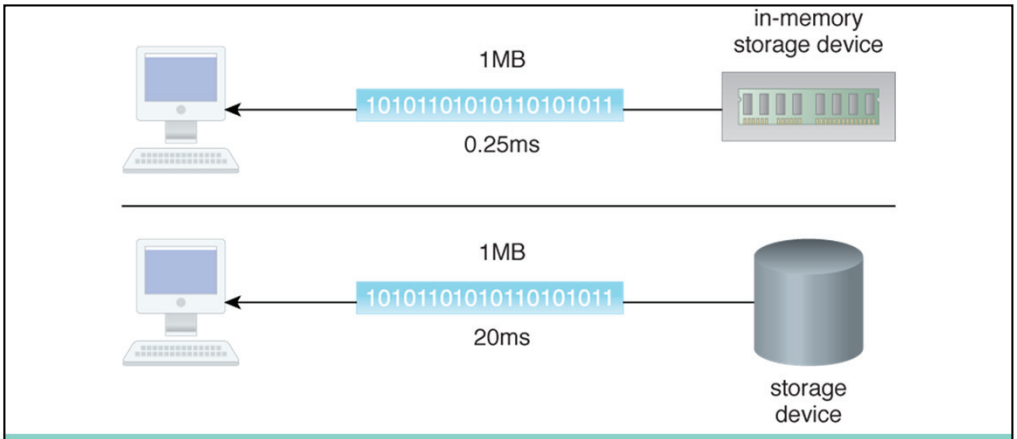
In-Memory Storage Devices



- Module 7 introduced the on-disk storage device and its various types as a fundamental means of data storage. Module 8 builds upon this knowledge by presenting in-memory storage as a means of advanced data storage.
- An in-memory storage device generally utilizes RAM, the main memory of a computer, as its storage medium to provide fast data access.
- The growing capacity and decreasing cost of main memory (RAM) coupled with the increasing read/write speed of solid state hard drives has made it possible to develop in-memory data storage solutions.

In-Memory Storage Devices

- Storage of data in memory eliminates the latency linked with the disk read/write and the data transfer between the main memory and the hard drive. This overall reduction in data read/write latency makes data processing much faster.
- In-memory storage device capacity can be increased massively by creating clusters consisting of multiple nodes connected together.
- Cluster-based memory enables storage of large amounts of data (Big Data datasets) which can be accessed considerably faster when compared with an on-disk storage device. This significantly reduces the overall execution time of Big Data analytics, thus enabling realtime Big Data analytics.



In the top diagram, a sequential read of 1 MB of data from an in-memory storage device takes around 0.25 ms. In the bottom diagram, reading the same amount of data from an on-disk storage device takes around 20 ms, which shows that reading data from in-memory storage is approximately eighty times faster than on-disk storage. Note that it is assumed that the network data transfer time is the same across the two scenarios and has therefore been excluded from the read time.

In-Memory Storage Devices

- An in-memory storage device enables *in-memory analytics*, which refers to in-memory analysis of data, such as generating certain statistics by executing queries on data that is stored in memory instead of on disk.
- In-memory analytics enable operational analytics and operational BI through fast execution of queries and algorithms.
- Primarily, in-memory storage enables making sense of the fast influx of data in a Big Data environment (velocity characteristic) by providing a storage medium that facilitates realtime insight generation. This supports making quick business decisions for mitigating a threat or taking advantage of an opportunity.

In-Memory Storage Devices

- A Big Data in-memory storage device is implemented over a cluster, providing high availability and redundancy. Therefore, horizontal scalability can be achieved by simply adding more nodes or memory.
- When compared with an on-disk storage device, an in-memory storage device is expensive because of the higher cost of memory as compared to a disk-based storage device.
- Although a 64-bit machine can make use of 16 exabytes of memory, due to the physical limitations of the machine, such as the number of memory bays, the installed memory is considerably less.
- For scaling out, it is not just the addition of more memory, but also the addition of nodes that are required once the per node memory limit is reached. This increases the data storage cost.

In-Memory Storage Devices

- Apart from being expensive, in-memory storage devices do not provide the same level of support for durable data storage. The price factor further affects the achievable capacity of an in-memory device when compared with an on-disk storage device.
- Consequently, only up-to-date and fresh data or data that has the most value is kept in memory, whereas stale data gets replaced with newer, fresher data.
- Depending on how it is implemented, an in-memory storage device can support schema-less or schema-aware storage. Schema-less storage support is provided through key-value based data persistence (explained shortly).

In-Memory Storage Devices

An in-memory storage device is appropriate when:

- data arrives at a fast pace and requires realtime analytics or event stream processing
- continuous or always-on analytics is required, such as operational BI and operational analytics
- interactive query processing and realtime data visualization needs to be performed, including what-if analysis and drill-down operations
- the same dataset is required by multiple data processing jobs

In-Memory Storage Devices

An in-memory storage device is appropriate when (continued):

- performing exploratory data analysis, as the same dataset does not need to be reloaded from disk if the algorithm changes
- data processing involves iterative access to the same dataset, such as executing graph-based algorithms
- developing low latency Big Data solutions with ACID transaction support

In-Memory Storage Devices

An in-memory storage device is inappropriate when:

- data processing consists of batch processing
- very large amounts of data need to be persisted for a longer term for performing in-depth data analysis
- performing strategic BI or strategic analytics that involves access to very large amounts of data and involves batch data processing
- datasets are extremely large and do not fit into the available memory

In-Memory Storage Devices

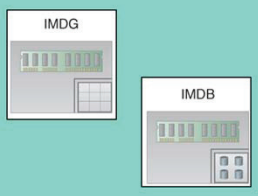
An in-memory storage device is inappropriate when (continued):

- making the transition from traditional data analysis towards Big Data analysis, as incorporating an in-memory storage device may require additional skills and involves a complex setup
- an enterprise has a limited budget, as setting up an in-memory storage device may require upgrading nodes, which could either be done by node replacement or by adding more RAM

In-Memory Storage Devices: Types

In-memory storage devices can be implemented as:

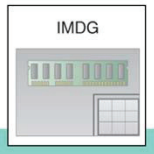
- In-Memory Data Grid (IMDG)
- In-Memory Database (IMDB)



Although both of these technologies use memory as their underlying data storage medium, what makes them distinct is the way data is stored in the memory.

Key features of each of these technologies are discussed over the next few pages.

In-Memory Storage Devices: IMDG



- IMDGs store data in memory as key-value pairs across multiple nodes where the keys and values can be any business object or application data in serialized form.
- This supports schema-less data storage through storage of semi/unstructured data.
- Data access is provided generally via APIs.

In-Memory Storage Devices: IMDG

In the diagram on the following page:

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

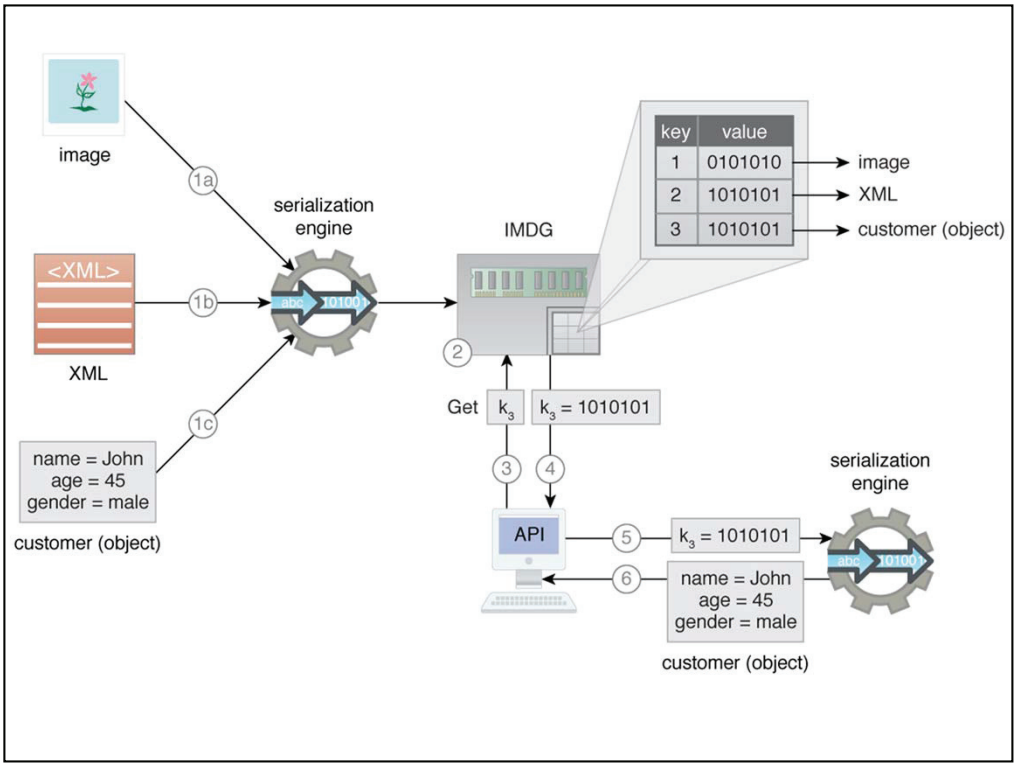


image
1a
serialization engine
IMDG
key
value
1
0101010
image
2
1010101
XML
3
1010101
customer (object)
<XML>
1b
XML
2
1c
Get
k_3
k_3 = 1010101
name = John
age = 45
gender = male
customer (object)
3
4
serialization engine
API
5
k_3 = 1010101
6
name = John
age = 45
gender = male
customer (object)

In-Memory Storage Devices: IMDG

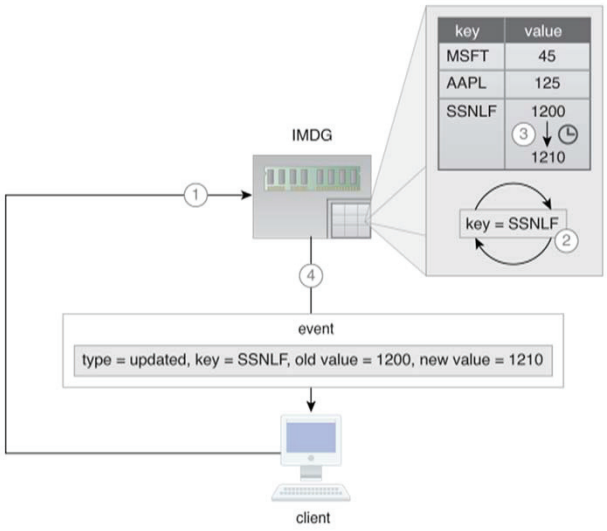
- Nodes in IMDGs keep themselves synchronized and collectively provide high availability, fault tolerance and consistency. In comparison to NoSQL's eventual consistency approach, IMDGs support immediate consistency.
- As compared to relational IMDBs (discussed under IMDB), IMDGs provide faster data access as IMDGs store non-relational data as objects. Hence, unlike relational IMDBs, *object-to-relational* mapping is not required and clients can work directly with the domain specific objects.

In-Memory Storage Devices: IMDG

- IMDGs scale horizontally by implementing data partitioning and data replication and further support reliability by replicating data to at least one extra node. In case of a machine failure, IMDGs automatically re-create lost copies of data from replicas as part of the recovery process.
- They are heavily used for realtime analytics as IMDGs support Complex Event Processing (CEP) via the publish-subscribe messaging model.
- This is achieved through a feature called *continuous querying*, also known as *active querying*, where a filter for event(s) of interest is registered once with the IMDG.
- CEP is introduced in the upcoming *Realtime Big Data Processing* section.

In-Memory Storage Devices: IMDG

- The IMDG then continuously evaluates the filter and whenever the filter is satisfied as a result of insert/update/delete operations, subscribing clients are informed.
- Notifications are sent asynchronously as change events, such as *added*, *removed* and *updated* events, with information about key-value pairs, such as *old* and new *values*.



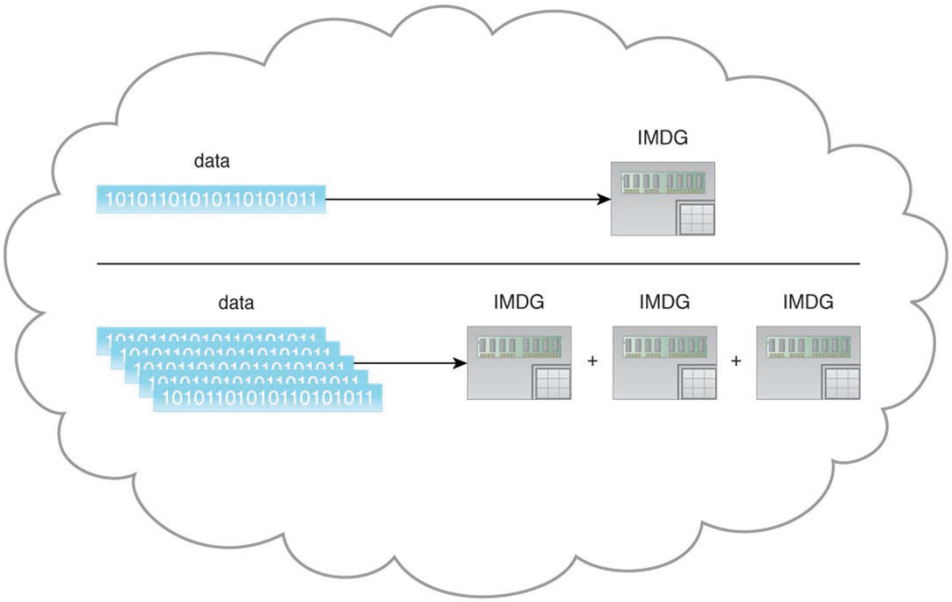
An IMDG stores stock prices where key is the stock symbol while value is the stock value (shown as text for readability). A client issues a continuous query (key=SSNLF) (1) which is registered in the IMDG (2). When the stock price for SSNLF stock changes (3), an *updated* event is sent to the subscribing client that contains various details about the event (4).

In-Memory Storage Devices: IMDG

- From a functionality point of view, an IMDG is akin to a distributed cache as both provide memory-based access to frequently accessed data. However, unlike a distributed cache, an IMDG provides built in support for replication and high availability.
- Realtime processing engines can make use of IMDG where high velocity data is stored in the IMDG as it arrives and is processed there before being saved to the on-disk storage device, or data from the on-disk storage device is copied to the IMDG. This makes data processing orders of large magnitude faster, and further enables data-reuse in case multiple jobs or iterative algorithms are run against the same data.

In-Memory Storage Devices: IMDG

- IMDGs may also support in-memory MapReduce that helps to reduce the latency of disk based MapReduce processing, especially when the same job needs to be executed multiple times.
- An IMDG can also be deployed within a cloud based environment where it provides a flexible storage medium that can scale out or scale in automatically as the storage demand increases or decreases.



An IMDG deployed in a cloud scales out automatically as the demand for data storage increases.

In-Memory Storage Devices: IMDG

- IMDGs can be added to existing Big Data solutions by introducing them between the existing on-disk storage device and the data processing application.
- However, this introduction generally requires changing application code in the form of API implementation.
- Note that some IMDG implementations may also provide limited or full SQL support.
- Examples include In-Memory Data Fabric, Hazelcast, and Oracle Coherence.

In-Memory Storage Devices: IMDG

In a Big Data solution environment, IMDGs are often deployed together with on-disk storage devices that act as the backend storage.

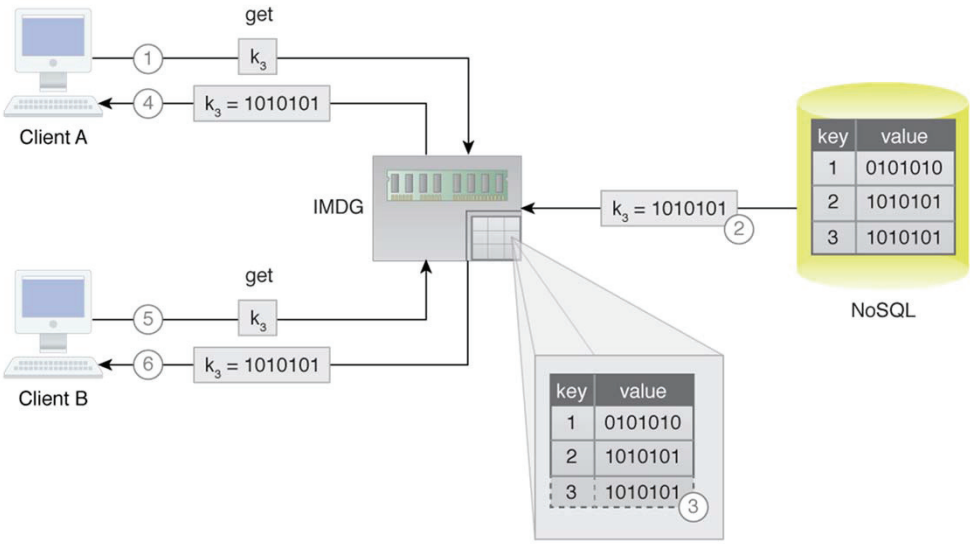
This is achieved via the following approaches that can be combined as per read/write performance, consistency and simplicity requirements:

- Read-through
- Write-through
- Write-behind
- Refresh-ahead

In-Memory Storage Devices: IMDG

Read-through

- If a requested value for a key is not found in the IMDG then it is synchronously read from the backend on-disk storage device, such as a database.
- Upon successful read from the backend on-disk storage device, the key-value pair is inserted into the IMDG and the requested value is returned to the client.
- Any subsequent requests for the same key are then served by the IMDG directly, instead of the backend storage.
- Although a simple approach, its synchronous nature may introduce read latency.

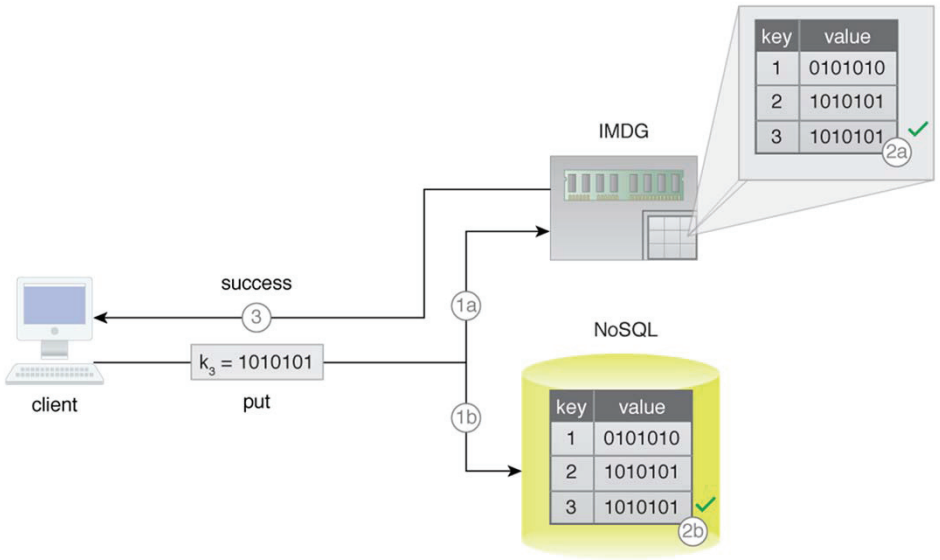


Client A tries to read key K_3 (1), which does not currently exist in the IMDG. Consequently, it is read from the backend storage (2) and is inserted into the IMDG (3) and then sent to Client A (4). A subsequent request for the same key by Client B (5) is then served directly by the IMDG (6).

In-Memory Storage Devices: IMDG

Write-through

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



A client inserts a new key-value pair (K_3,V_3), which is inserted both in the IMDG (1a) and the backend storage (1b) in a transactional manner. Upon successful insertion of data both in the IMDG (2a) and the backend storage (2b), the client is informed that data has been successfully inserted (3).

In-Memory Storage Devices: IMDG

Write-behind

- Any write to the IMDG is written asynchronously in a batch manner to the backend on-disk storage device, such as a database.
- A queue is generally placed between the IMDG and the backend storage for keeping track of the required changes to the backend storage. This queue can be configured to write data to the backend storage at different intervals.
- The asynchronous nature increases both write performance (write operation is considered completed as soon as it is written to the IMDG) and read performance (data can be read from the IMDG as soon as it is written to the IMDG) and scalability/availability in general.

In-Memory Storage Devices: IMDG

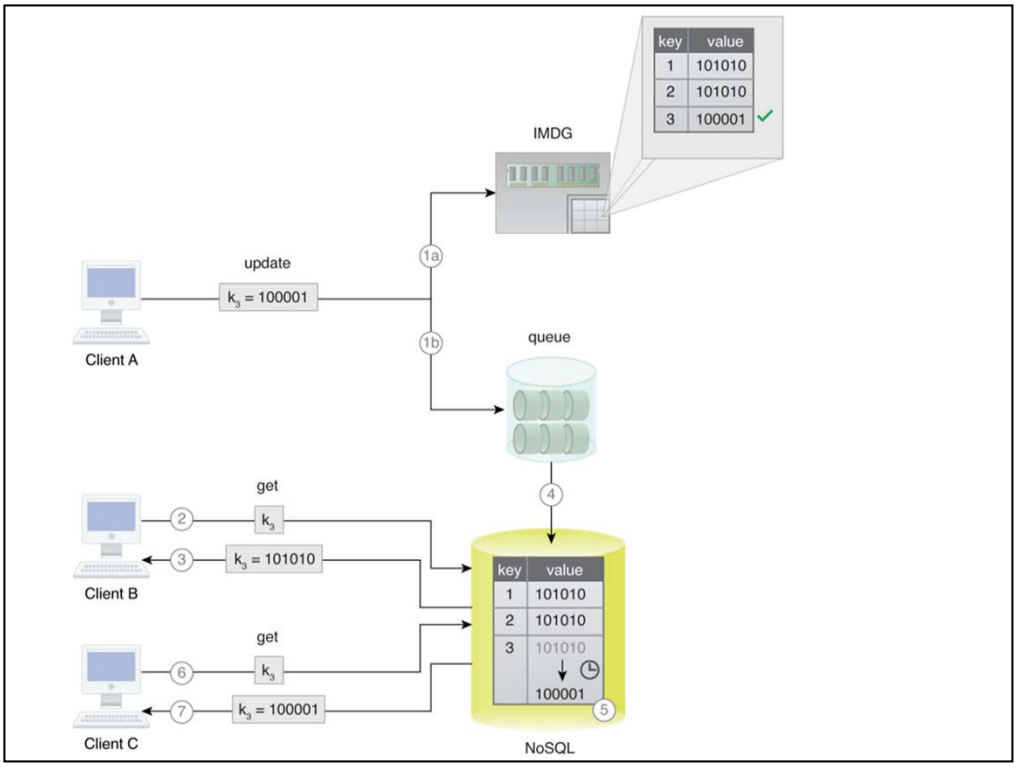
Write-behind (continued)

- However, the asynchronous nature introduces inconsistency until the backend storage is updated at the specified interval.

In-Memory Storage Devices: IMDG

In the diagram on the following page:

1. Client A updates value of K_3, which is updated in the IMDG (a) and is also sent to a queue (b).
2. However, before the backend storage is updated, Client B makes a request for the same key.
3. The old value is sent.
4. After the configured interval …
5. … the backend storage is eventually updated.
6. Client C makes a request for the same key.
7. This time, the updated value is sent.


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

In-Memory Storage Devices: IMDG

Refresh-ahead

- This is a proactive approach where any frequently accessed values are automatically, asynchronously refreshed in the IMDG, provided that the value is accessed before its expiry time as configured in the IMDG.
- If a value is accessed after its expiry time, the value is synchronously read from the backend storage, as in the case of read-through approach, and is updated in the IMDG before being returned to the client.

In-Memory Storage Devices: IMDG

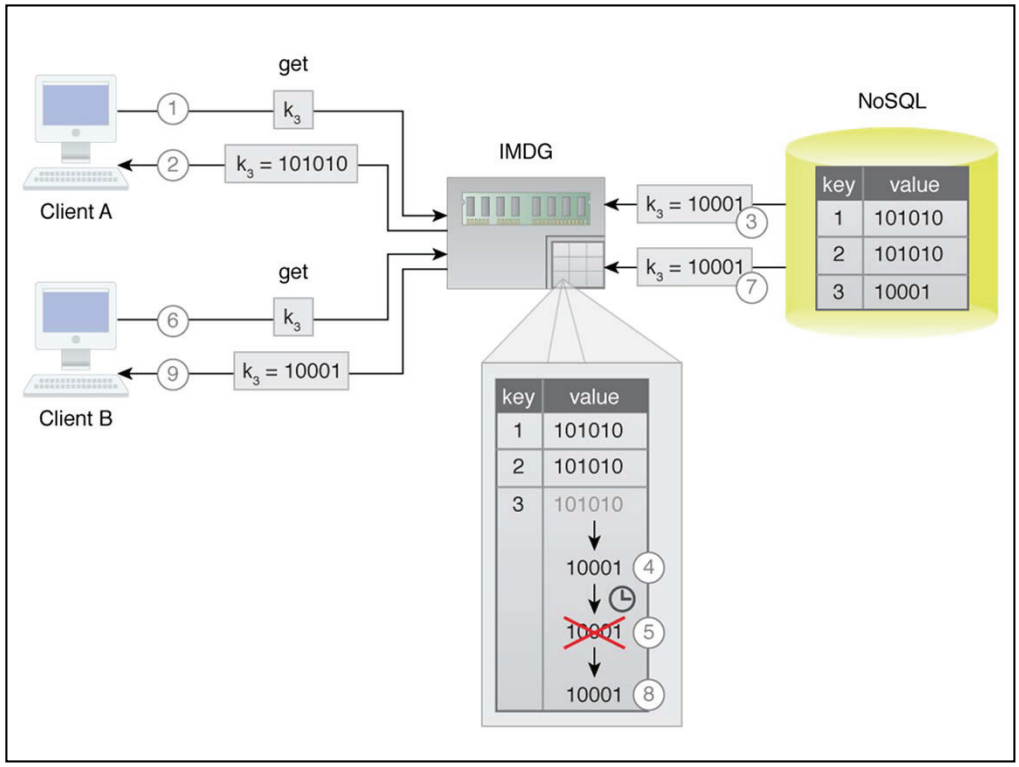
Refresh-ahead (continued)

- Due to its asynchronous and forward looking nature, this approach helps to achieve better read-performance and is especially useful when the same values are accessed frequently or accessed by a number of clients.
- As compared to the read-through approach where a value is served from the IMDG until its expiry, data inconsistency between the IMDG and the backend storage is minimized as values are refreshed before their expiry time.

In-Memory Storage Devices: IMDG

In the diagram on the following page:

1. Client A requests K_3 before its expiry time.
2. The current value is returned from the IMDG.
3. The value is refreshed from the backend storage.
4. The value is then updated in the IMDG asynchronously.
5. After the configured expiry time, the key-value pair is evicted from the IMDG.
6. Now Client B makes a request for K_3.
7. As the key does not exist in the IMDG, it is synchronously requested from the backend storage …
8. … and updated.
9. The value is then returned to Client B.


get
1
k_3
2
k_3 = 101010
Client A
get
6
k_3
9
k_3 = 10001
Client B
IMDG
NoSQL
k_3 = 10001
3
k_3 = 10001
7
key
value
1
101010
2
101010
3
10001
key
value
1
101010
2
101010
3
101010
10001
4
10001
5
10001
8

In-Memory Storage Devices: IMDG

An IMDG storage device is appropriate when:

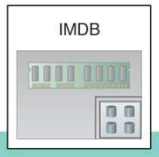
- data needs to be readily accessible in object form with minimal latency
- data being stored is non-relational in nature such as semi-structured and unstructured data
- adding realtime support to an existing Big Data solution currently using on-disk storage

In-Memory Storage Devices: IMDG

An IMDG storage device is appropriate when (continued):

- the existing storage device cannot be replaced but the data access layer can be modified
- scalability is more important than relational storage; although IMDGs are more scalable than IMDBs (IMDBs are functionally complete databases), they do not support relational storage

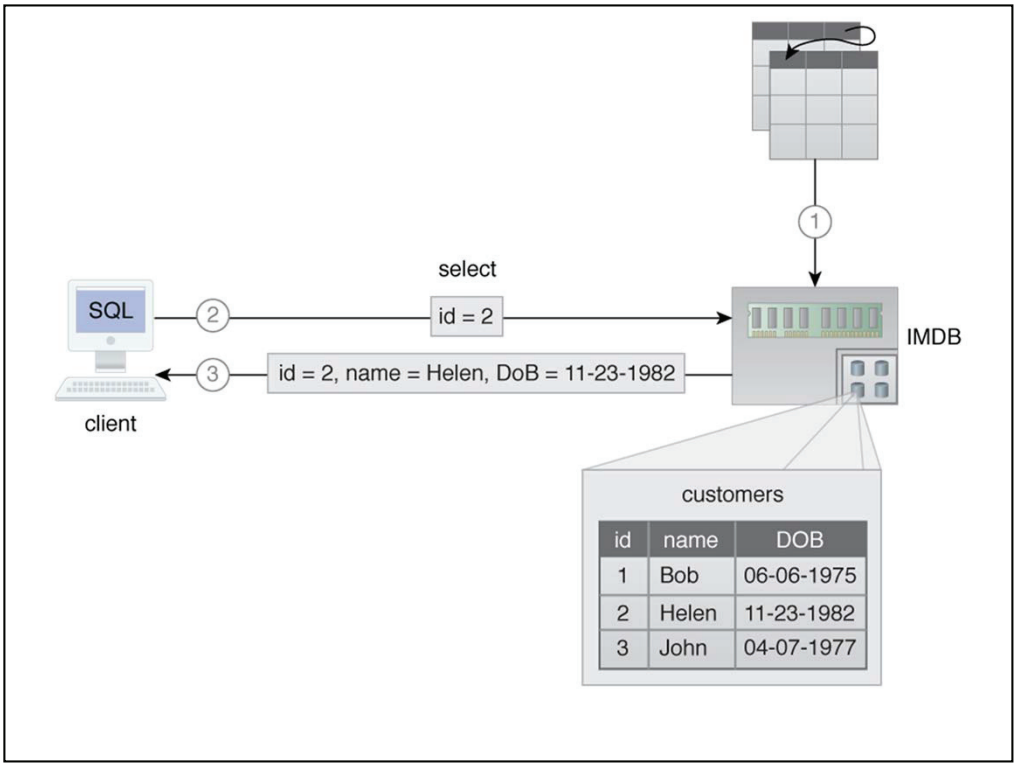
In-Memory Storage Devices: IMDB



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

In the diagram on the following page:

1. A relational dataset is stored into an IMDB.
2. A client requests a customer record (id = 2) via SQL.
3. The relevant customer record is then returned by the IMDB, which is directly manipulated by the client without the need for any deserialization.


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

In-Memory Storage Devices: IMDB

- An IMDB can be relational in nature (relational IMDB) for the storage of structured data, or may leverage NoSQL technology (non-relational IMDB) for the storage of semi-structured and unstructured data.
- Unlike IMDGs, which generally provide data access via APIs, relational IMDBs make use of the more familiar SQL language, which helps data analysts or data scientists that do not have advanced programming skills.
- NoSQL-based IMDBs generally provide API-based access, which may be as simple as put, get and delete operations.
- Depending on the underlying implementation, some IMDBs scale-out, while others scale-up, to achieve scalability.

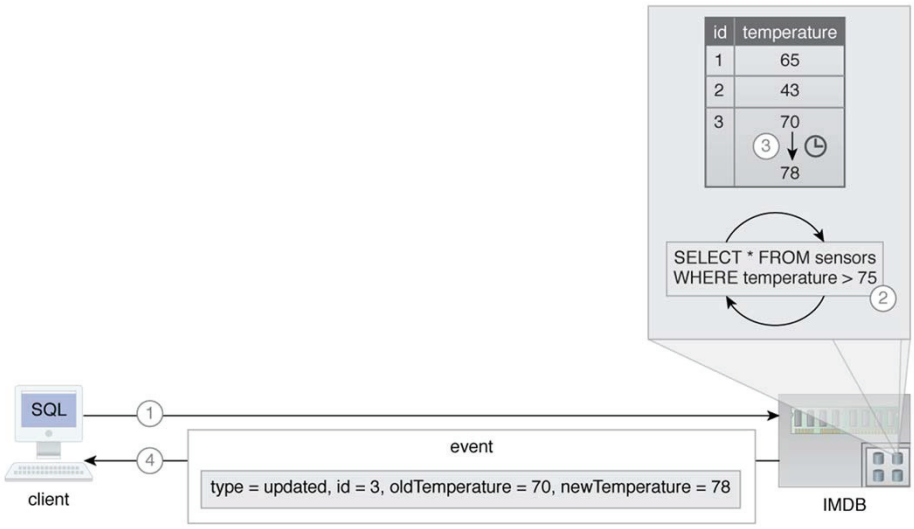
In-Memory Storage Devices: IMDB

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

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

In-Memory Storage Devices: IMDB

- Like an IMDG, an IMDB may also support the continuous query feature, where a filter in the form of a query for data of interest is registered once with the IMDB.
- The IMDB then continuously executes the query in an iterative manner. Whenever the query result is modified as a result of insert/update/delete operations, subscribing clients are asynchronously informed by sending out changes as events, such as *added*, *removed* and *updated* events, with information about record values, such as *old* and new *values*.



An IMDB stores temperature values for various sensors. A client issues a continuous query (select * from sensors where temperature > 75) (1) which is registered in the IMDB (2). When the temperature for any sensor exceeds 75^0C (3), an *updated* event is sent to the subscribing client that contains various details about the event (4).

In-Memory Storage Devices: IMDB

- IMDBs are heavily used in realtime analytics and can further be used for developing low latency applications requiring full ACID transaction support (relational IMDB).
- In comparison with IMDGs, IMDBs provide an easy to set up in-memory data storage option, as IMDBs do not generally require on-disk backend storage devices.
- Introduction of IMDBs into the existing Big Data solutions generally requires replacement of existing on-disk storage devices, including any RDBMSs if used.

In-Memory Storage Devices: IMDB

- In the case of replacing an RDBMS with a relational IMDB, little or no application code change is required due to SQL support provided by the relational IMDB. However, when replacing with a NoSQL IMDB, code change may be required due to the implementation of APIs.
- In the case of replacing an on-disk NoSQL database with a relational IMDB, code change will often be required to establish SQL-based access. However, when replacing with a NoSQL IMDB, code change may still be required due to the implementation of new APIs.

In-Memory Storage Devices: IMDB

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

In-Memory Storage Devices: IMDB

An IMDB storage device is appropriate when:

- relational data needs to be stored in memory with ACID support
- adding realtime support to an existing Big Data solution currently using on-disk storage
- the existing on-disk storage device can be replaced with an in-memory equivalent technology

In-Memory Storage Devices: IMDB

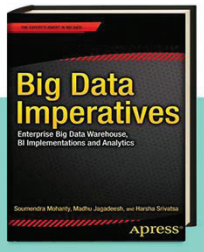
An IMDB storage device is appropriate when (continued):

- it is required to minimize changes to the data access layer of the application code, such as when the application consists of an SQL-based data access layer.
- relational storage is more important than scalability; although many IMDBs support relational storage, they are less scalable than IMDGs.

Note

Some vendor implementations may provide features that support, partially or fully, both IMDG and IMDB. Consequently, such implementations cannot be distinguished as either IMDG or IMDB, rather these can generally be referred as in-memory storage devices.

Reading



"Big Data Imperatives"

IMDG & IMDB
Pages 222 – 224, 229 - 230

Exercise

Exercise 8.2

Match Terms to Statements

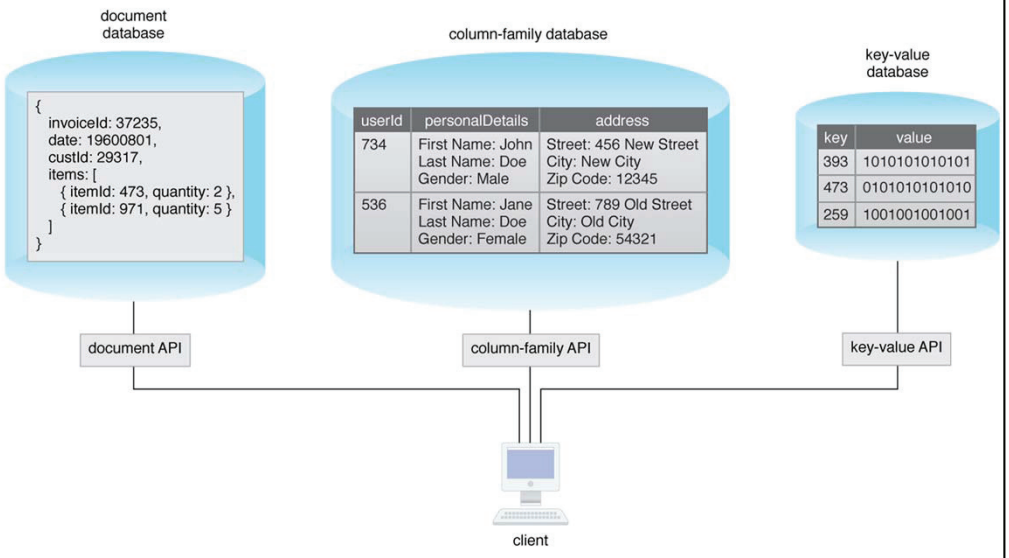
Polyglot Persistence

Polyglot Persistence

- On-disk (Module 7) and in-memory storage devices, along with their subtypes, serve different purposes.
- Each is built for storing a specific type of data (structured, unstructured and semi-structured) and data access patterns (streaming and batch), and supporting different types of workloads (realtime and batch).
- For example, graph databases are optimized for storing relationship-oriented data while key-value databases are better suited for storing binary data.

Polyglot Persistence

- To address realtime and batch processing of Big Data datasets while supporting structured, semi-structured and unstructured data storage requirements, a heterogeneous storage environment is required where specialized, fit-for-purpose storage devices are used for persisting data.
- Such an environment is known as *polyglot persistence*, which is a hybrid approach that involves the use of different types of storage devices based on individual data storage requirements.
- This helps to develop an optimal storage architecture where each subtype of the storage device specializes in storing data for particular use cases.



The diagram depicts a polyglot persistence environment where invoices are stored in a document database, customer information is stored in a column-family database and product images are stored in a key-value database. A client retrieves the required data by using the API provided by each NoSQL storage device.

Polyglot Persistence

- An in-memory storage device can be used to perform realtime analysis on event-based data whereas an on-disk storage device can be used for detailed batch data analysis on unstructured data.
- Similarly, storage devices like key-value, document, column-family and graph NoSQL databases, can each be used depending upon the type of data to be stored, how the data is accessed (individual fields or as a BLOB) and whether the data includes any relationships.

Polyglot Persistence

- Apart from the type of the data that a storage device supports, ACID and BASE requirements further specify which storage devices can be used in a polyglot storage environment.
- Note that implementing polyglot persistence can also include the use of relational databases. Within Big Data environments, this refers to the use of OLTP systems or data warehouses for structured data storage, while semi-structured or unstructured data is stored in NoSQL databases.
- Use of a relational database may also be appropriate when a subset of a dataset is only updated/accessed sparingly and does not warrant the need for a scalable, cluster-based NoSQL approach.

Polyglot Persistence: Issues

- Although polyglot persistence helps address multiple storage requirements, it complicates the storage architecture and the Big Data solution design.
- For example, creating a report that involves products (stored in a column-family database), their images (stored in a key-value database) and invoices (stored in a document database) would require consolidating data from three different storage devices.
- Secondly, this exposes multiple potential failure points where the reliability and availability of the data processing system is no more than the storage device with the lowest level of reliability and availability.

Polyglot Persistence: Issues

- With polyglot persistence, the management of different types of storage devices can become a concern and may require individuals with specialized skillsets, such as a separate database administrator (DBA) per database or a single DBA with multiple skills.
- Similarly, with little or no SQL-like support across many storage devices, integrating each storage device may require custom adapters or in-application data joining and filtering.
- Different storage devices have different security and access controls. This makes consistent authentication and authorization difficult.

Polyglot Persistence: Issues

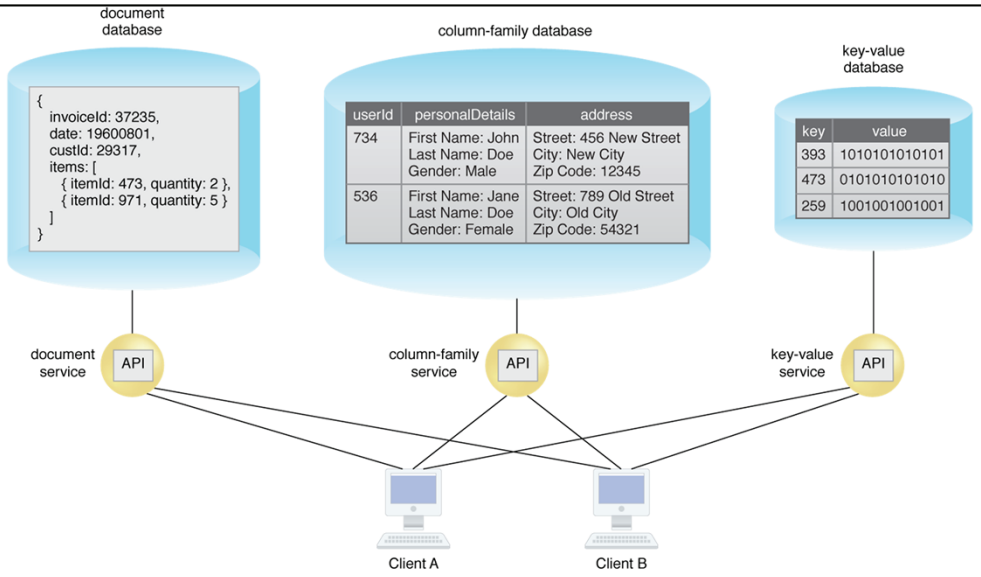
- For performing detailed data analyses in Big Data environments, due to the variety characteristic, it may be necessary to join data from disparate storage devices, each with its own individual storage model, in a staging storage device. This introduces data access latency and further complicates how data is accessed.
- On the other hand, multiple applications or data products may interact with the same storage device, each using its own implementation of the storage device API. This creates duplicate code bases, making maintenance difficult.
- A *data product* is an instantiation of a statistical/machine learning model built during data analysis. It exists in the form of an application and generates value from data to fulfill a business goal.

Polyglot Persistence: Recommendations

- In a polyglot persistence environment, communication with multiple storage devices can be standardized by encapsulating the API implementation logic into a service.
- This not only helps to achieve a standardized communication interface but also protects applications from any changes to the underlying storage device, such as replacement of the storage device.

Polyglot Persistence: Recommendations

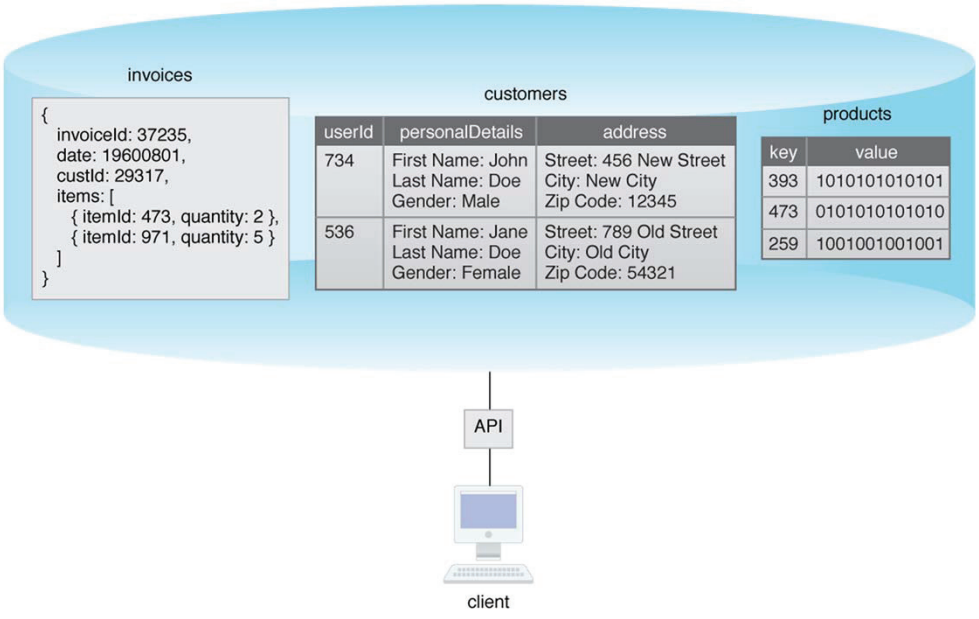
- Security is another concern when using polyglot persistence as unlike RDBMSs, Big Data storage devices, such as NoSQL databases, generally do not provide security out of box or may implement individual, heterogeneous security schemes.
- Encapsulating access to the storage devices in a service can further add a security layer to storage devices that do not have any inbuilt security features, or can homogenize heterogeneous security interactions for storage devices with individual security mechanisms.
- This can achieve a standardized security endpoint that provides authentication and authorization features for multiple storage devices in a polyglot storage environment.



The diagram depicts a polyglot persistence environment. To standardize data access and to secure each storage device, the API access to each storage device is encapsulated in a service. Instead of individual data access, all clients use the same endpoint for accessing a particular storage device.

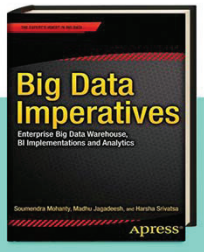
Polyglot Persistence: Recommendations

- Some storage devices may support the storage of multiple data types, for example both key-value and document-oriented data.
- Their introduction in a polyglot persistence environment can eliminate the use of multiple storage devices, which can simplify the storage architecture.



A single NoSQL storage device provides storage for unstructured, semi-structured and structured data.

Reading



"Big Data Imperatives"

Polyglot Persistence (use cases)
Pages 98 - 102

Realtime Big Data Processing

Introduction

Modules 2 and 7 established several fundamental concepts and terminology related to Big Data processing. These included clusters, distributed file systems, distributed data processing, parallel data processing, processing workloads, data parallelism and task parallelism. These established a firm foundation for understanding batch data processing in Big Data environments.

This section introduces realtime Big Data processing, how it works, and discusses whether or not it can use MapReduce.

Before discussing realtime Big Data processing, a fundamental principle related to Big Data processing called speed, consistency and volume (SCV) is covered.

Speed Consistency Volume (SCV)

Whereas the CAP theorem discusses properties of data storage, SCV is a theorem regarding distributed data processing. It states that a distributed data processing system can be designed to support only two of the following three requirements:

- Speed – refers to how quickly the data can be processed once it is generated. In the case of realtime analytics, data is processed comparatively faster than batch analytics. This generally excludes the time taken to capture data and focuses only on the actual data processing, such as generating statistics or executing an algorithm.

Speed Consistency Volume (SCV)

- Consistency – refers to the accuracy and the precision of the results. Results are deemed accurate if they are close to the correct value and precise if close to each other. A more consistent system will make use of all available data, resulting in high accuracy and precision as compared to a less consistent system that makes use of sampling techniques, which can result in lower accuracy with an acceptable level of precision.
- Volume – the amount of data that can be processed. Big Data's velocity characteristic results in fast growing datasets leading to huge volumes of data that need to be processed in a distributed manner. Processing such voluminous data in its entirety while ensuring speed and consistency becomes an issue.

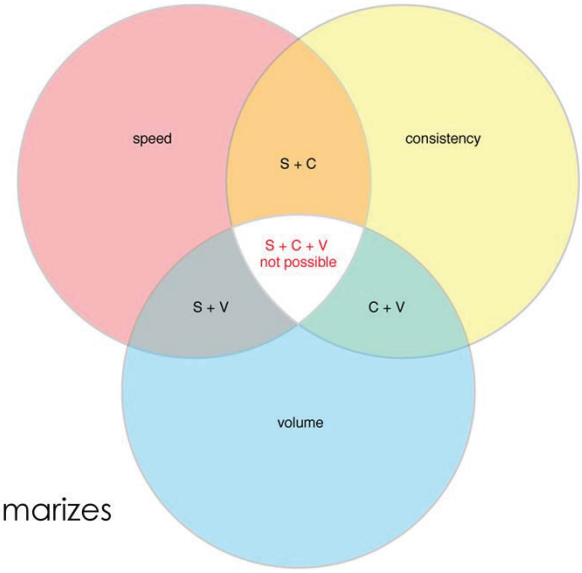
Speed Consistency Volume (SCV)

- If speed (S) and consistency (C) are required then it may not be possible to process high volumes of data (V) as processing large amounts of data slows down data processing.
- If consistency (C) and processing of high volumes of data (V) are required then it may not be possible to process the data at high speed (S) as achieving high speed data processing requires smaller data volumes.
- If high volume (V) data processing coupled with high speed (S) data processing are required then the processed results may not be consistent (C) as high speed processing of large amounts of data involves sampling the data which may reduce consistency.

Speed Consistency Volume (SCV)

- It should be noted that the choice of which two out of the three dimensions to support is totally dependent upon the system requirements.
- In Big Data environments, availability of the maximum amount of data is mandatory for performing in-depth analysis, such as pattern identification. Hence, forgoing volume (V) over speed (S) and consistency (C) needs to be considered carefully as data may still be required for batch processing in order to glean further insights.
- In the case of Big Data processing, assuming that data (V) loss is unacceptable, generally a realtime data analysis system will either be SV or CV, depending upon if speed (S) or consistent results (C) are required.

Speed Consistency Volume (SCV)



The Venn diagram summarizes the SCV principle.

Realtime Big Data Processing

- Processing Big Data in realtime generally refers to realtime or near realtime analytics.
- Data is processed as it arrives at the enterprise boundary without unreasonable delay.
- Instead of initially persisting the data to the disk, for example to a database, the data is first processed in memory and then persisted to the disk for future use/archival purposes.
- This is opposite of batch processing mode, where data is persisted to the disk first and then subsequently processed, which can cause significant delays.

Realtime Big Data Processing

- Analyzing Big Data in realtime employs the use of in-memory storage devices (IMDGs or IMDBs).
- Once in memory, the data can then be processed in realtime without incurring any disk I/O latency.
- The realtime processing may involve calculating simple statistics, executing complex algorithms or updating the state of the in-memory data as a result of change in some data metric.
- For enhanced data analysis, in-memory data can be combined with previously batch processed data or denormalized data loaded from on-disk storage devices. This helps to achieve realtime data processing as datasets can be joined in memory.

Realtime Big Data Processing

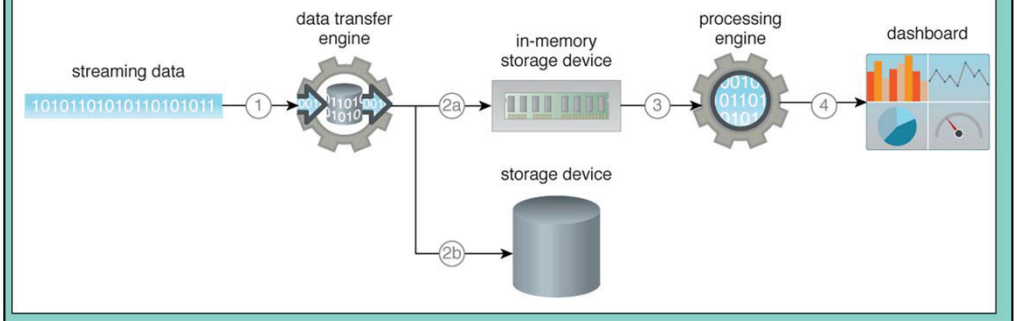
- Although realtime Big Data processing generally refers to incoming new data, it can also include queries performed on already persisted old data that requires interactive response.
- Once the data has been processed, the processing results can be pushed out to the interested consumers, for example via a realtime dashboard application or a Web application that delivers realtime updates to the user.
- Depending on system requirements, the processed data along with the raw input data can then be offloaded to the on-disk storage device for complex batch data analyses using the synchronous write-through approach or the asynchronous write-behind approach.

Realtime Big Data Processing

In the diagram on the following page:

1. Streaming data is captured via a data transfer engine.
2. It is then synchronously saved to an in-memory storage device (a) and an on-disk storage device (b).
3. A processing engine is then used to process data in realtime.
4. Finally, the results are fed to a dashboard for operational analysis.

Realtime Big Data Processing



The diagram shows realtime data processing in a Big Data environment.

Realtime Big Data Processing

Two important concepts within realtime Big Data processing are:

Event stream processing (ESP)

- The incoming stream of events, generally from a single source and ordered by time, are continuously analyzed via the application of algorithms which are mostly formula-based or via fast execution of queries before storing to an on-disk storage device.
- Other (memory resident) data sources can also be incorporated for performing richer analytics, whereas the processing results can be fed to a dashboard or can act as a trigger for another application that performs a preconfigured action or further analyses.
- It focuses more on speed than complexity; the operation to be executed is comparatively simple to aid faster execution.

Realtime Big Data Processing

Complex event processing (CEP)

- A number of realtime events, often from disparate sources and spread over different time intervals, are analyzed simultaneously for unearthing patterns and initiating some action.
- Algorithms, which are mostly rule-based, and statistical techniques are applied, and business logic and processes are also taken into account for discovering cross-cutting complex event patterns.
- CEP focuses more on complexity, providing rich analytics. However, as a result, speed of execution may be adversely affected.
- In general, CEP is considered to be a superset of ESP and often times the output of ESP results in the generation of synthetic events that can be fed into CEP.

Realtime Big Data Processing & SCV

- While designing a realtime Big Data processing system, the SCV principle needs to be kept in mind.
- In light of this principle, consider a hard-realtime and a near-realtime Big Data processing system. For both hard-realtime and near-realtime scenarios, we assume that data loss is unacceptable, in other words high data volume (V) processing is required for both the systems.
- Note that the requirement that the data loss should not occur does not mean that all data will actually be processed in realtime (explained shortly). Rather, it means that the system captures all input data and that the data is always persisted to the disk either indirectly from the in-memory storage or directly to the on-disk storage device(s).

- In the case of a hard-realtime system, a fast response (S) is required, hence consistency (C) will be compromised if high volume data (V) needs to be processed in memory because of the use of some sampling or approximation techniques (less accurate results with tolerable precision).
- In the case of a near-realtime system, a reasonably fast response (S) is required, hence consistency (C) can be guaranteed if high volume data (V) needs to be processed in memory. Results will be more accurate when compared to hard-realtime system as the algorithm can be applied to the complete dataset instead of taking samples or employing any approximation techniques.
- This shows that, in the context of Big Data processing, a hard-realtime system requires a compromise on consistency (C) to guarantee a fast response (S) while a near-realtime system requires a compromise on speed (S) to guarantee consistent results (C).

Realtime Big Data Processing & MapReduce

- MapReduce is a batch-oriented processing engine that is good at processing large amounts of data at rest. However, it cannot process data incrementally and can only process complete datasets.
- It requires all input data to be available in its entirety before the execution of the data processing job, in order words the data needs to be frozen, whereas realtime data processing involves data that is often incomplete and continuous such as streams.
- In MapReduce, a reduce task generally cannot start before the completion of all map tasks. First, the map output is persisted locally on each node that runs the map function. Next, the map output is copied over the network to the nodes that run the reduce function, introducing processing latency.

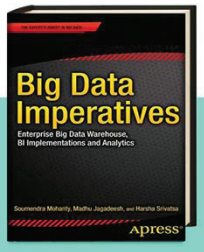
Realtime Big Data Processing & MapReduce

- Similarly, the results of one reducer cannot be directly fed into another reducer, rather the results have to be passed to a mapper first.
- The aforementioned shortcomings, when coupled with the overhead associated with job creation and coordination, serve to make MapReduce generally unsuitable for realtime Big Data processing.
- However, there are some strategies that can enable the use of MapReduce in near-realtime Big Data processing scenarios.

Realtime Big Data Processing & MapReduce

- One strategy is to use in-memory storage device(s) to keep input data for running interactive queries that consist of MapReduce jobs. Alternatively, micro-batch MapReduce jobs can be deployed that are configured to run on comparatively smaller datasets, stored in memory or on disk, with frequent intervals, such as every fifteen minutes.
- Another approach is to continuously run MapReduce jobs on complete datasets stored in on-disk storage devices to create *materialized views* which can then be combined with small volume results obtained from newly arriving in-memory streaming data for interactive query processing.
- It should be noted some Open Source projects, for example, Apache Spark, Apache Storm and Apache Tez, do provide true realtime Big Data processing capabilities.

Reading



"Big Data Imperatives"

Realtime Analytics
Pages 227 – 228

Exercise

Exercise 8.3

Identify True/False Statements

Advanced MapReduce Algorithm Design

MapReduce Algorithm Design

- Module 7 introduced fundamental design considerations for creating algorithms for use with the MapReduce framework. Module 8 builds upon this foundation and further explores the characteristics of problems for which a MapReduce-based solution is appropriate.
- In general, the problem that needs to be solved via MapReduce should be an *embarrassingly parallel* problem. Such a problem is one that can easily be divided into sub-problems so that the sub-problems can be solved on their own without requiring any communication between them.

MapReduce Algorithm Design

- Tasks run as part of the algorithm should not require a shared state nor should they require the passing of messages between them for calculation of the result.
- Algorithms that require access to intermediate results or where access to previous set of values is required, such as iterative algorithms, are generally not amenable to the MapReduce framework.
- Multistep algorithms need to be run as a series of multiple jobs, which can be achieved using a workflow engine or chaining multiple jobs together through a driver program.

MapReduce Algorithm Design

- Problems to be solved are generally aggregation-based such that results from individual tasks are merged together in some manner, although some may not involve aggregation (a map-only job without any reducer), for example, facial recognition where an image recognition algorithm is applied to each image in the mapper.
- Problems to be solved should not be iterative in nature and should not require updating states and performing conditional processing based on the current state, for example, calculating distance between entities in a graph-based data where each vertex can be in a different state (*not processed, processing, processed*).

MapReduce Algorithm Design

- Iterative machine learning algorithms such as clustering and classification are implemented as multiple MapReduce jobs, since a single MapReduce job can accommodate only one iteration at a time.
- For example, classification via the Apriori algorithm involves finding frequently occurring single items and then pairs of items, and so on before finally combining the results from each iteration to find generate rules. The iteration required by this algorithm would require multiple MapReduce jobs.
- The data type of the keys and values specified as the input to the reduce function should match the data type of the keys and values written out from the map function.

MapReduce Algorithm Design

- A mapper is mandatory for a MapReduce job. In other words, output from a reducer cannot directly become the input of another reducer, which requires creating a dummy mapper so that data can be passed to the reducer.
- MapReduce works well for data processing tasks that involve performing operations on data contained within the same aggregate (the record read by the mapper), such as unit price multiplied by number of units sold.
- On the other hand, operations that require access to values contained in other records are not suitable for MapReduce. For example, finding correlation between two different key values (each key in a separate record), as in the case of finding correlation between stock tickers.

MapReduce Algorithm Design

- Making algorithms amenable to MapReduce may require changing the data structure or some pre-processing so that all required values are in the same record, which could itself be achieved through a MapReduce job.
- When designing the algorithm, it needs to be verified if the operation performed on the data obeys the *distributive law*, which states that an operation is distributive if the result obtained from performing an operation on a number is the same as performing the operation on parts of that number and then adding the result. For example, 2x7 has the same result as 2x(3+4) or (2x3)+(2x4).

$$2 \times 7 = 2 \times (3+4) = (2 \times 3) + (2 \times 4)$$

MapReduce Algorithm Design

- Based on distributive property, instead of applying a function to a field across all records, the function can be applied to subsets of records first and then applied to the individual results obtained from each subset.
- For example, consider a file distributed among two nodes consisting of hypothetical smart meter readings taken every fifteen minutes, comprising the following fields:

node	meter id	timestamp	load (watts)	min voltage	max voltage
1	A	20120515:0600	500	107.80	112.50
1	B	20120515:0600	800	108.75	111.15
2	A	20120515:0615	650	109.25	113.50
2	B	20120515:0615	750	107.15	112.35

MapReduce Algorithm Design

To calculate the *maximum load* across all households, if all the readings are in a single file, maximum load can easily be found by processing all load values simultaneously.

Max(500, 800, 650, 750)=800

However, in the case of MapReduce, maximum load for readings within a single node can be found first in the map function before finding the actual maximum load based on the intermediate maximum load results from the two nodes in the reduce function.

	node 1	node 2
map	Max(500, 800)=800	Max(650, 750)=750
reduce	Max(800, 750)=800	

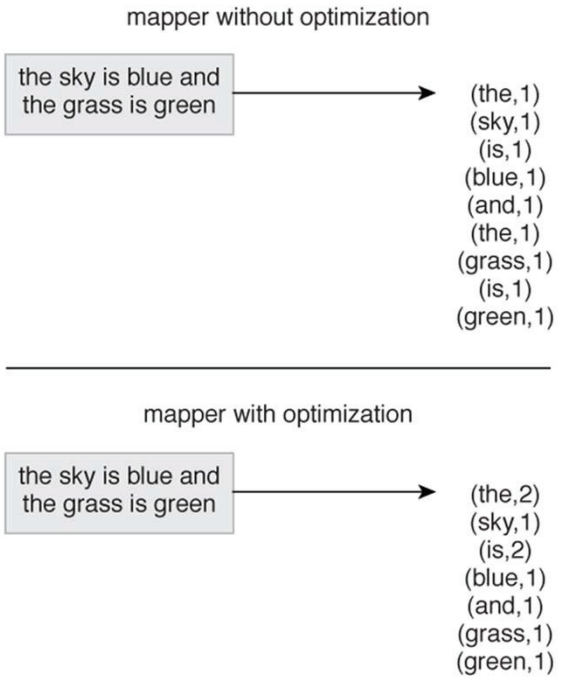
MapReduce Algorithm Design

- Although finding the *maximum load* works in this scenario, it is not possible to distribute calculation of the *average load* in a similar fashion as taking the average of the averages does not bear the same result as taking the average of all numbers simultaneously.
- The algorithm design in the case of finding average load needs to be modified so that the reduce function has access to not only the *sum of all numbers* but also the *quantity of numbers* that constitute that sum.
- It is important to decrease the amount of data sent from mappers to reducers as much as possible because communication within the cluster is one of the major overheads that contributes to the overall job execution latency.

MapReduce Algorithm Design

- The amount of data sent from mappers to reducers can be decreased through *local aggregation* by performing the operation locally within the mapper and only sending the result to the reducer where all results can be aggregated in some manner to arrive at the intended result.
- For example, for a word count algorithm, instead of just extracting each word and writing out *one* (a constant variable with a value of one) as the count, the mapper can be optimized to write the total count for each unique word across all input records for the mapper (like a combiner).

A word count MapReduce algorithm with optimized and unoptimized mapper versions. In the top diagram, an unoptimized mapper generates nine key-value pairs, compared to the bottom diagram with an optimized mapper that generates seven key-value pairs.



MapReduce Algorithm Design

- Local aggregation can also be applied through the use of a combiner that intercepts the mapper's output, provides access to all the values for each unique key in the mapper's output so that the values can be aggregated in some manner via an operation, before sending the aggregate result as a single value for each key to the reducer.
- The data type of the keys and values specified as the input and output for the combine function should match the data type of the keys and values written from the map function.

MapReduce Algorithm Design

When a combiner is used, the operation applied to the dataset should support *associative* and *commutative* properties.

An operation that fulfils the associative property is one where the result remains the same irrespective of how the numbers are grouped together. For example, (2x3)x4 has the same result as 2x(3x4).

$$(2 \times 3) \times 4 = 2 \times (3 \times 4)$$

An operation that fulfils the commutative property is one where the result remains the same irrespective of the order in which the numbers appear. For example, 3x4 has the same result as 4x3.

$$3 \times 4 = 4 \times 3$$

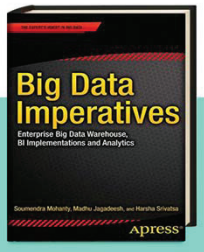
MapReduce Algorithm Design

- Recall from Module 7 that the combiner is not always guaranteed to be executed by the MapReduce framework. If the combiner executes, the groupings and order of values seen by the reducer could be different.
- However, the output from the reducer needs to be the same, irrespective of whether the combiner gets executed or not. This is accomplished by performing operations that support both the associative and commutative properties.

MapReduce Algorithm Design

- Each combiner will apply the operation to the locally available group of values (single mapper output), which is only a subset of the group of values processed by a reducer. If the combiner is executed, the groupings seen by the reducer may be different. However, if the operation supports the associative property, its final output is not impacted by variations in the groupings of the data.
- The order in which the values are processed by the combiner may be different than the order of the values in the reducer. This is due to the localized view of the combiner (single mapper output), as opposed to the global view of the reducer. However, if the operation supports the commutative property, its final output is not affected by the order of the values in the data.

Reading



"Big Data Imperatives"

MapReduce Patterns
Pages 162 - 165

The Bulk Synchronous Parallel Processing Engine

Introduction

- Bulk Synchronous Parallel (BSP) is a highly scalable, parallel/distributed processing engine that enables efficient execution of algorithms that are iterative in nature and require message passing to process large amounts of data.
- BSP was proposed by Leslie Valiant in 1990, executes over a cluster of machines, and is generally used for graph data processing and executing machine learning algorithms.
- BSP is generally used as a batch processing engine but can also be used for realtime or near-realtime processing, thereby providing multi-workload support.
- It also supports redundancy and fault-tolerance through regular checkpoints and restarting failed processes automatically. Being a pure message passing processing engine, it further supports schema-less data processing.

Bulk Synchronous Parallel: Concepts

A single data processing run in BSP consists of a series of *supersteps* where each superstep is composed of three stages:

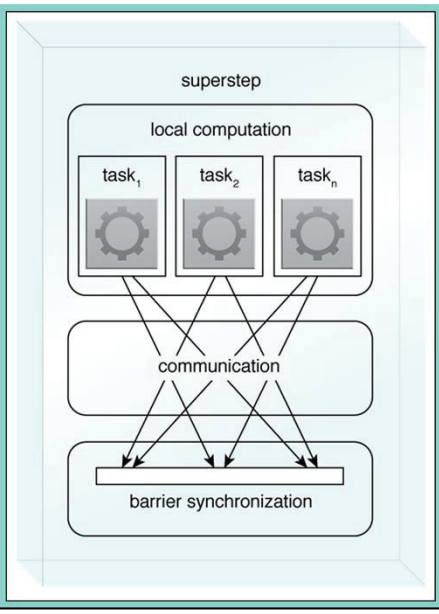
- local processing: multiple tasks execute a user-defined function on data items in parallel and asynchronously on multiple nodes in order to process input data that is local to each node. Messages sent by other tasks in the previous superstep are also retrieved and processed in this stage.
- communication: processing output is transmitted as messages to specific tasks or all tasks that may be running on the same node or different nodes.
- barrier synchronization: any task that has finished data processing and sending out messages waits until all other tasks have finished data processing and sending out messages.

Bulk Synchronous Parallel: Concepts

- The superstep is repeated for a fixed number of times or until a condition becomes true.
- The messages sent in the same superstep are not available for consumption until the next superstep. For example, messages sent in superstep s-1 are available in superstep s while messages transmitted in superstep s can only be retrieved in superstep s+1.
- Note that a task is generally executed by a process hosted on a logical processor. A single physical processor hosts multiple logical processors. Hence, a single cluster node generally hosts multiple instances of processes where each process executes a task in parallel and asynchronously to other processes.

Bulk Synchronous Parallel: Concepts

The diagram summarizes the execution of a single superstep in BSP. Multiple tasks perform computations in parallel and distributed manner in the local computation stage. Messages are sent in the communication stage once each task has completed its computation. Finished tasks wait for other unfinished tasks during the barrier synchronization stage.



Bulk Synchronous Parallel: Concepts

The following two mechanisms help to increase the usability and efficiency of the BSP processing engine:

- Aggregators
- Combiners

Aggregators

Aggregators are tasks that have a global visibility to which any other task can send and retrieve messages from. Basically, these are used for creating summaries of data such as min, max, sum. They also act as a means of achieving selective processing, such as conditional execution of code based on the current value of the aggregator, and global coordination among tasks.

Bulk Synchronous Parallel: Concepts

Combiners

Multiple messages sent to a task can be merged into a single message (depending upon the algorithm, such as sum) to reduce network communication and memory utilization. Combiners combine messages into a single message. Like MapReduce, a combiner function may not always be executed, as a result combiners should only be used when the operation being applied supports associative and commutative properties.

BSP vs MapReduce

- An iterative algorithm in MapReduce needs to be executed by rerunning the same MapReduce job recursively. This incurs substantial overhead as not only does the mappers' output need to be saved to a disk before being sent to the reducers, but the intermediate output from reducers also needs to be saved to a disk in preparation for the next iteration of the MapReduce job.
- In MapReduce, synchronization between mappers and reducers is implicitly achieved by requiring all mappers to finish their processing before the reducers can read their output.
- Synchronization refers to achieving system uniformity by making sure that tasks are run in the correct order and that a task finishes completely before the same task is rerun or a different task is run.

BSP vs MapReduce

- MapReduce does not allow messages to be passed between mappers or reducers. To obtain output from other mappers, intermediate output from each mapper is written out during the map stage which is only available to other mappers in the next iteration of the MapReduce job.
- Although messages can be sent, only mappers can communicate with reducers indirectly. This is done implicitly by the MapReduce engine through grouping the same keys emitted by the mappers so that these can be forwarded to the relevant reducers as messages.

BSP vs MapReduce

- Reverse communication from reducers to mappers is not supported. As a workaround, the output from reducers is saved first before it can be read by the mappers in the next iteration of MapReduce.
- Sending a message in MapReduce is somewhat similar to specifying the same key in the map function as output so that it gets transmitted to the same reducer. However, between different runs of a MapReduce job, the same reduce function may get executed on a different node due to the stateless nature of MapReduce.

BSP vs MapReduce

- MapReduce is stateless; whenever map or reduce tasks finish, the output is saved to the disk and the local state, such as the allocated input of the map and reduce tasks, is cleared.
- For running the same MapReduce job multiple times, the output from each job run needs to be saved to a storage device and then explicitly read back in as an input into the map task in the next run of the MapReduce job.

BSP vs MapReduce

- In MapReduce, transferring data from mappers to reducers and the requirement for saving output to the disk between iterations and reading data back in at the start of each job run contribute to inefficient data processing.
- On the other hand, BSP is stateful: input splits containing the data items are allocated to each task once at the start, and are maintained locally across multiple supersteps.
- Across multiple supersteps, the same tasks continue processing the same data items on the same nodes until the stopping condition is reached. The tasks also locally maintain any state data in between the supersteps.

BSP vs MapReduce

- A superstep in BSP is equivalent to the single execution of a MapReduce job. However, unlike a single job, the superstep does not finish and can be repeated multiple times.
- The efficiency of the BSP engine is, however, dependent on the ability to persist the input split in its entirety within the memory of the processing nodes.
- If the input split cannot be completely retained in the memory, excessive data is offloaded to the disk, which is an overhead, and results in less efficient data processing.

Bulk Synchronous Parallel

The BSP processing engine can be used when:

- processing relationship-oriented datasets
- data processing requires extensive message passing
- processing data in an iterative fashion
- fast, near-realtime data processing is required

Bulk Synchronous Parallel

The BSP processing engine should not be used when:

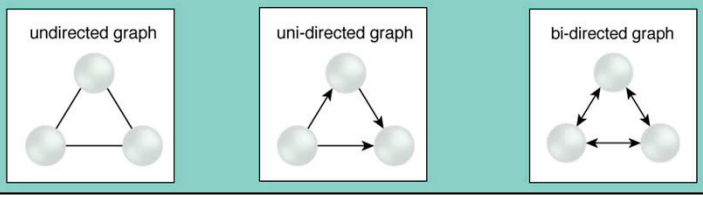
- processing datasets that consist of aggregate-focused records that are not linked with other records
- data processing involves reading the input once and the algorithm can be completed as a single job run
- the state of the input remains the same during the processing
- the data assigned to a task cannot be fully persisted in the main memory of the cluster nodes across multiple supersteps

Graph Data

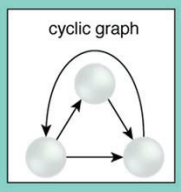
- Graph data, as briefly discussed in Module 7, refers to a set of entities that are connected together via some form of relationship.
- Each entity is generally known as a *vertex* or *node* while each connection is known as an *edge*.
- In the context of graph data, a node represents a data item or a record and is not the same as a computer node in a cluster.
- Each vertex generally has an id and may consist of a single value or multiple key-value pairs. Similarly, each edge may consist of a single value or multiple key-value pairs.

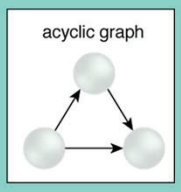
Graph Data Types

- An edge can either be directed or undirected. In case of a directed edge, there can be one-way traversal (uni-directional edge) or two-way traversal (bi-directional edge) between the vertices. Undirected edges are treated the same way as bi-directional edges.
- A graph that consists of directed edges only is called a directed graph, while a graph that consists of undirected edges only is called an undirected graph. A graph may contain a combination of both directed and undirected edges.

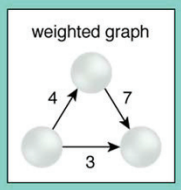


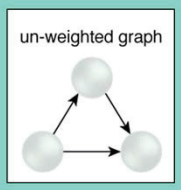
A cyclic graph is one where the edges are arranged in a manner such that a vertex can be revisited from another vertex (forming a loop that may include more than two vertices). A loop-free graph is known as an acyclic graph.





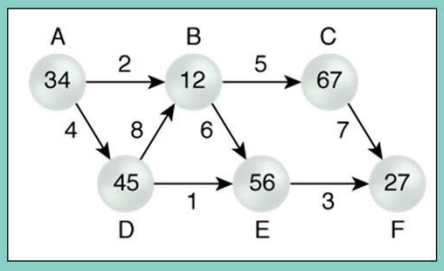
A weighted graph consists of edges that have a related numeric value representing the cost of traversal between two vertices. The opposite is known as an un-weighted graph.



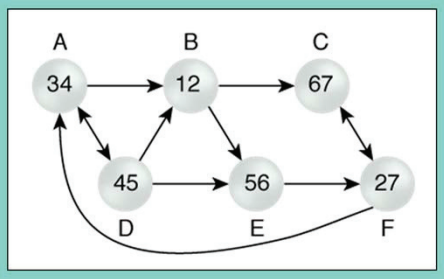


In the following two graphs, circles and arrows represent vertices and edges respectively. Letters indicate the identity of vertices while the vertex's value is shown by the number inside the circle.

The top diagram shows an example of a directed, acyclic and weighted graph. The number with each arrow shows the weight of the edge.



The bottom diagram shows an example of a uni/bi-directed, cyclic and un-weighted graph.



Graph Data Processing

- Graph data occurs in a number of shapes and forms, such as social networks, protein-protein interaction networks, networks of devices, and pipe networks.
- Relational data in an RDBMS can also be converted to graph data so that it can be processed using specialized graph processing engines. Each row in a table becomes a vertex, and each foreign key becomes an edge. Additional non-foreign key-based edges can also be added.
- Although graph data processing differs from case to case, it is generally characterized by iterative processing of vertices where the calculation of a value is dependent on the connected vertices of a vertex or vertices that are in close vicinity of a vertex (not connected, but neighbors).

Graph Data Processing

- A number of machine learning algorithms, such as k-means for clustering and other algorithms, such as shortest path and depth-first search, operate on graph data where the calculation of the result requires recursive processing of vertices and message passing between vertices.
- In Big Data environments, owing to the volume characteristic, the size of a graph can grow massively. As a result, storing and processing large graphs on a single machine is either not possible or leads to less than optimum performance levels.
- Due to its connected nature, graph data is generally kept in memory to achieve optimum processing performance. This eliminates the bottleneck of frequent disk access required to read connected vertices.

Graph Data Processing: BSP

- Processing large graphs in a scalable manner requires splitting a large graph into multiple shards and storing them over a cluster. Such a partitioned graph can then be processed using a graph processing engine that generally keeps each shard in memory and processes each shard in parallel in an iterative manner.
- BSP is one such processing engine that can be used for processing large graphs in an efficient manner due to its iterative data processing and message passing features.
- When using BSP for graph processing, each task processes pre-assigned vertices along with their edges that are kept in local memory throughout the entirety of the processing.

Graph Data Processing: BSP

Graph data processing in the context of a BSP processing engine generally consists of the following stages:

Pre-processing

- Each graph partition is divided into multiple splits where each split consists of data that represents multiple vertices.
- Data within the splits is parsed as vertices which are then allocated to individual tasks. Multiple tasks are generally executed on each cluster node.
- The partitioning of the vertices can generally be customized so that related vertices are allocated to the same task or are kept in close proximity, such as within the same node.
- All vertices at this initial stage are set to *active* state.

Graph Data Processing: BSP

Local Computation

- The parsed vertices are processed during this stage using a user-defined function that is generally known as the *compute* function. The defined functionality is at the individual vertex level, and during each superstep the compute function is executed for each active vertex.
- The code within the compute function generally follows the pattern of reading and processing any messages sent to the vertex in the previous superstep (none for the first superstep) and performing any required processing, such as executing an operation, changing the state of the vertex or its edges, or creating new vertices.

Graph Data Processing: BSP

Communication

- In this stage, a vertex communicates with other vertices by sending out messages in an asynchronous manner. These can either be the vertices connected to the current vertex via an outgoing edge, or it can be any other vertex whose id is known.
- These messages generally consist of lightweight payloads, such as a single computed value.
- Messages sent to a vertex are not available instantaneously in the same superstep, rather the messages become available only in the next superstep.

Graph Data Processing: BSP

Barrier Synchronization

- Once a vertex has performed its allocated work (data processing and message transmission), it is deactivated and set to *inactive* state.
- An inactive vertex is not processed in subsequent supersteps unless it is reactivated by the receipt of messages in any of the subsequent supersteps.
- Barrier synchronization is achieved by waiting until all tasks have processed their allocated vertices before declaring the current superstep to be complete.

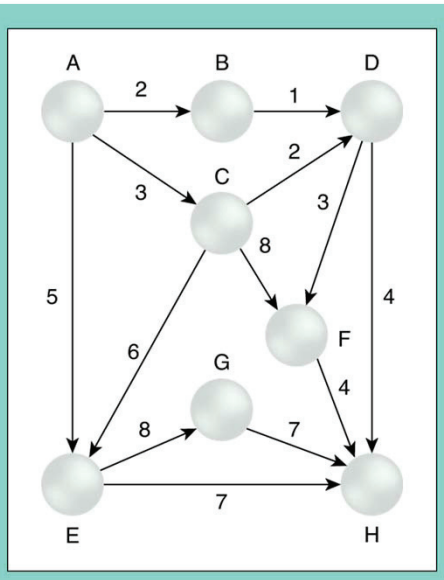
Graph Data Processing: BSP

- The *local computation*, *communication* and *barrier synchronization* stages are repeated until there are no more vertices to be processed (all vertices become inactive) and there are no messages to be sent.
- The BSP processing engine maintains the state of the vertices during the entire processing run.
- For each vertex, the state includes a flag identifying if the vertex is active or inactive, its runtime value, the runtime value of each outgoing edge along with the id of the vertex that it points to, and all incoming messages sent during the previous superstep.

Graph Data Processing: BSP Example

- Over the next few pages, an example of graph data processing using the BSP processing engine is discussed.
- Consider the graph on the next page, which represents a small road network. The vertices show the cities while the edges show the roads that connect the cities.
- For simplicity, all the roads are one-way (uni-directional graph) and there is no road that links a city back to any of the preceding cities (acyclic graph). The number next to each edge represents the corresponding distance between the two cities (weighted graph).
- The shortest distance from the source city (Vertex A) to all other cities (Vertices B to H), to which it is connected directly or indirectly, needs to be calculated.

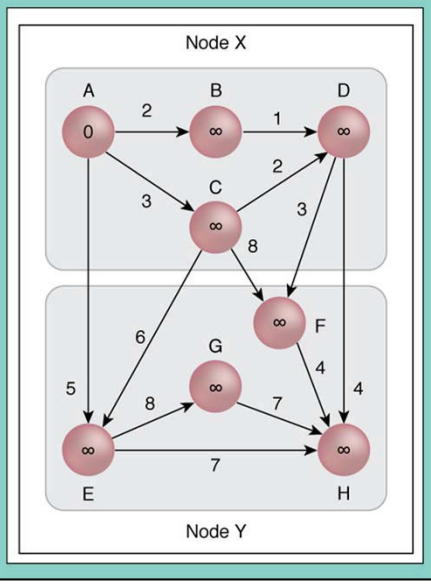
Graph Data Processing: BSP Example



Graph Data Processing: BSP Example

- The Big Data solution environment consists of a cluster comprising two processing nodes (Node X and Y). Each processing node runs a BSP task in parallel. Vertices A to D are allocated to Node X while Vertices E to H are allocated to Node Y.
- All vertices are initialized with a value of infinity (∞), except Vertex A, which is initialized with a value of zero as the distance to itself is zero. The infinity value represents that the maximum possible distance exists between the vertex and the source Vertex A, and that the current vertex has not been traversed yet (unprocessed).
- Active vertices (green) send the sum of the vertex value and the outgoing edge value as messages in envelopes to other connected vertices. Inactive vertices are shown as red.

Graph Data Processing: BSP Example



Graph Data Processing: BSP Example

The user-defined logic within the compute function works as follows:

- Each active vertex processes any messages that were sent by other vertices in the previous superstep (first superstep has zero incoming messages).
- The incoming messages' values (sum of source vertex value and its outgoing edge value) are compared against the current value of the vertex. If a smaller value is found, the current vertex's value is updated with this new value.
- All those vertices whose values get updated then send messages (sum of vertex value and outgoing edge value) to other connected vertices via their outgoing edges.

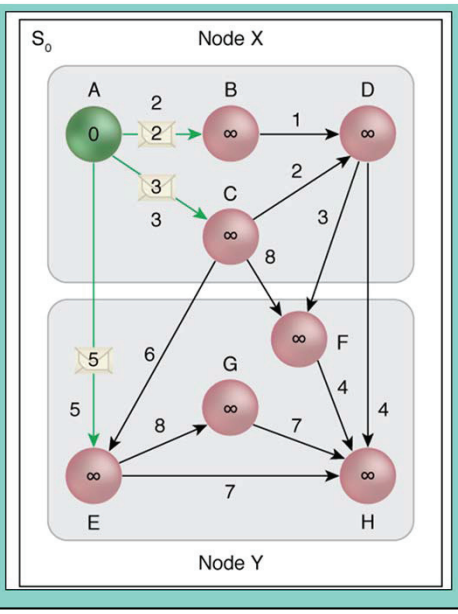
This logic is re-run until there are no more messages to be sent.

Graph Data Processing: BSP Example

Superstep S_0

In the first superstep, Vertex A is the active vertex. As there are no incoming messages to be processed, Vertex A simply sends messages 2, 3 and 5 to Vertices B, C and E respectively.

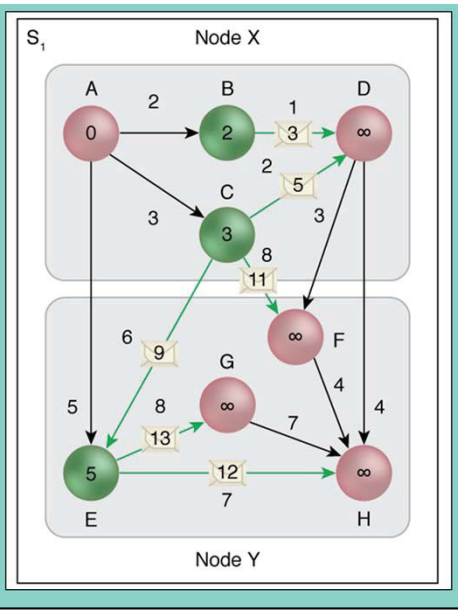
Vertex A becomes inactive after sending all the required messages.



Graph Data Processing: BSP Example

Superstep S_1

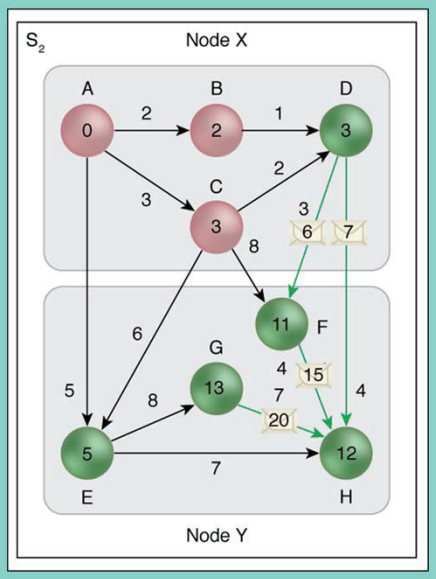
In the second superstep, messages sent by Vertex A are delivered to Vertices B, C and E, consequently these vertices become active. After comparison, values 2, 3 and 5 are assigned to Vertices B, C and E respectively.
These vertices then send messages to Vertices D, E, F, G and H, and then become inactive.



Superstep S_2

In the third superstep, Vertices D, F, G, E and H become active. Note that Vertex E is reactivated due to Vertex C's message sent in the previous superstep. After comparison, values 3, 5, 11, 13 and 12 are assigned to Vertices D, E, F, G and H respectively.

Vertex E's value remains unchanged and hence it does not send any messages. Apart from E and H, other updated vertices (D, F and G) then send messages to Vertices F and H and then all become inactive.



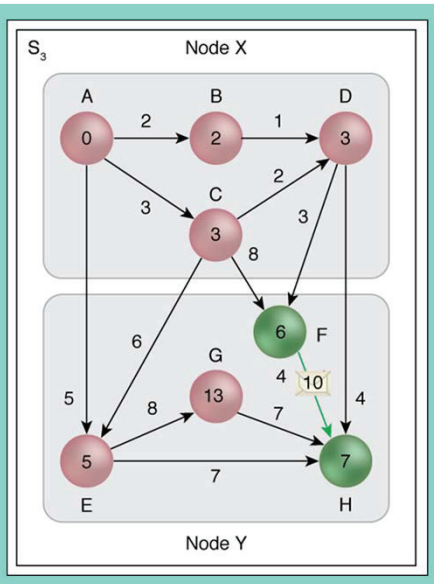
Graph Data Processing: BSP Example

Superstep S_3

In the fourth superstep, Vertices F and H become active. Note that both these vertices are reactivated due to the messages sent to them in the previous superstep.

After comparison, values 6 and 7 are assigned to Vertices F and H respectively.

The updated vertex F then sends a message to Vertex H and then both Vertices H and F become inactive.

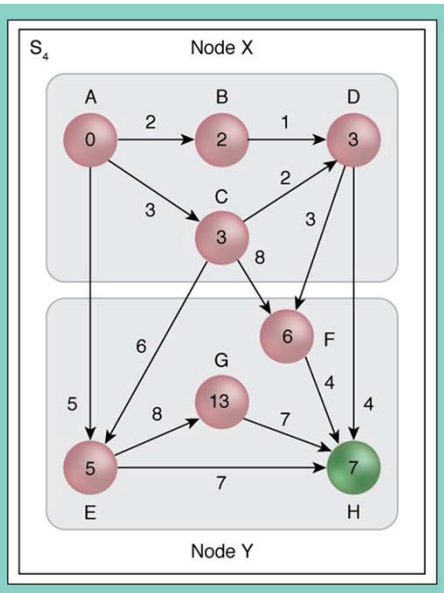


Graph Data Processing: BSP Example

Superstep S_4

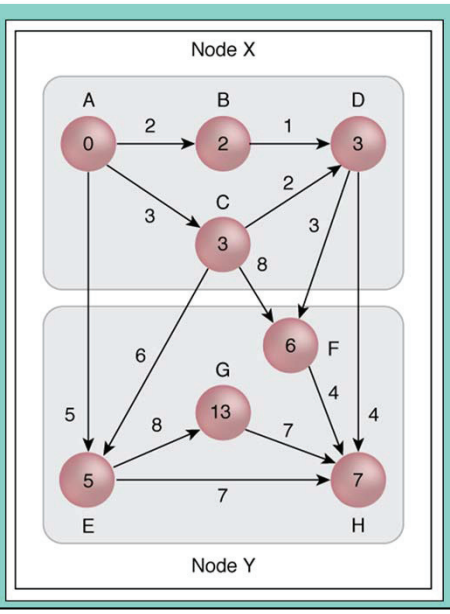
In the fifth superstep, Vertex H becomes active. Note that Vertex H is reactivated due to the message sent to it in the previous superstep. However, after comparison, its value remains unchanged.

As Vertex H's value is not updated and it has no outgoing edges, it simply becomes inactive without sending any messages.



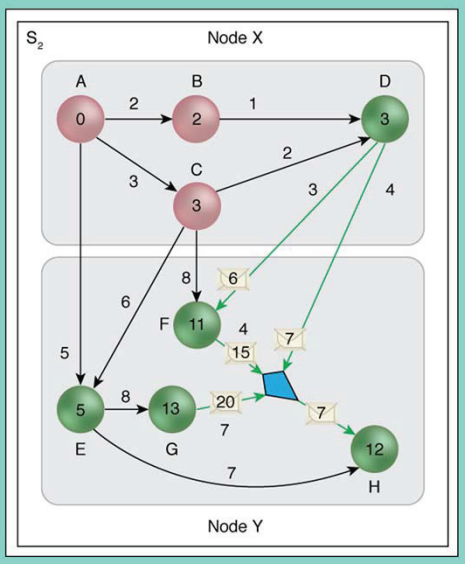
Graph Data Processing: BSP Example

As there are no more messages to be processed, the processing is terminated with each vertex carrying a value that represents the shortest distance from City A to itself.



Combiner

The logic defined in the compute function is favorable to the use of a combiner function. Instead of forwarding all the messages sent to a vertex, only a single message with the smallest value is forwarded which is then compared against the vertex's value. For example, the three messages sent to Vertex H in S_2 are replaced by a single message as shown in the diagram. Similarly, although not shown, the two messages sent to Vertex D in S_1 are replaced with a single message.



Graph Data Processing: BSP Example (MapReduce Comparison)

In comparison, a MapReduce processing engine will process the same graph as follows (a single job run is described):

Map function

- Any active vertex is set to inactive, keeping its current value and is sent to the reducer as a key-value pair. For example, in S_0 key-value pair *{A,(0,inactive)}* is generated for Vertex A.
- For each above vertex, a key-value pair is generated for each of its linked vertices, where the key is the target vertex id and the value is distance. The value also comprises active status. For example, in S_0 key-value pairs *{B,(2,active)},{C,(3,active)}, {E,(5,active)}* are generated for Vertex A.
- Any inactive vertex is sent as-is to the reducer as a key-value pair. For example, in S_0 key-value pair *{B,(∞,inactive)}* is generated for Vertex B.

Graph Data Processing: BSP Example (MapReduce Comparison)

Reduce function

- The reducer receives a consolidated list of key-value pairs for each vertex including both active and inactive vertices.
- For each vertex, if more than one value exists in the list, then its value is updated with the smallest distance value and its status is set to active (as it received messages from other vertices) and is written out. For example, in S_2 key H receives values *{(7,active),(15,active),(20,active),(12,inactive)}*, as a result Vertex H's value is updated to seven with active status.
- If a single value exists in the list, then it means the corresponding vertex did not receive any message. Such a vertex is written out as-is. For example, in S_2 key C receives a value *{(3,inactive)}*, as a result Vertex C's value remains as three with an inactive status.

Exercise

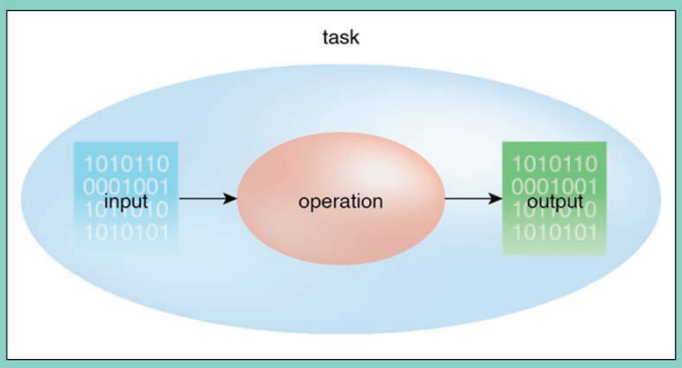
Exercise 8.4

Organize the BSP Stages

Big Data Pipeline & ELT

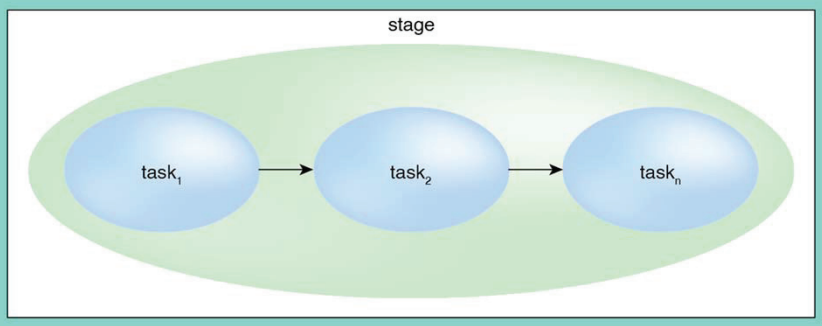
Data Pipeline

A data pipeline is a data-driven workflow consisting of tasks where each task involves input, operation, and output.



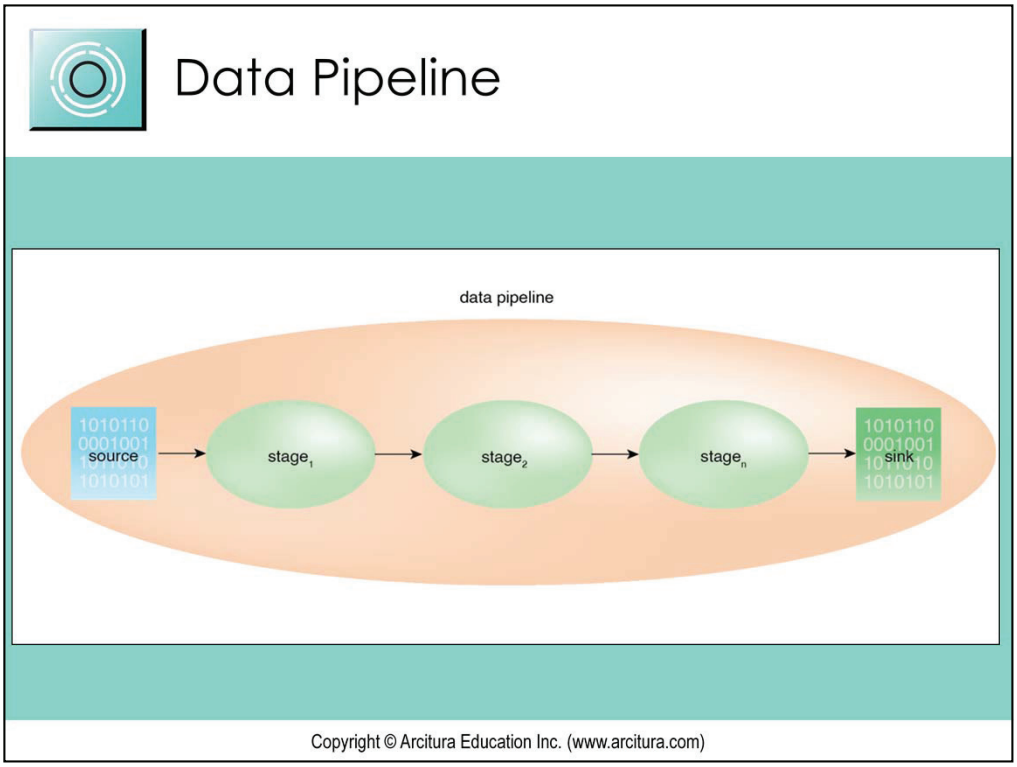
Data Pipeline

Each data pipeline consists of multiple tasks joined together in a serial manner so that the output of one task becomes the input of the subsequent task. Such a combination of tasks represents an individual stage.



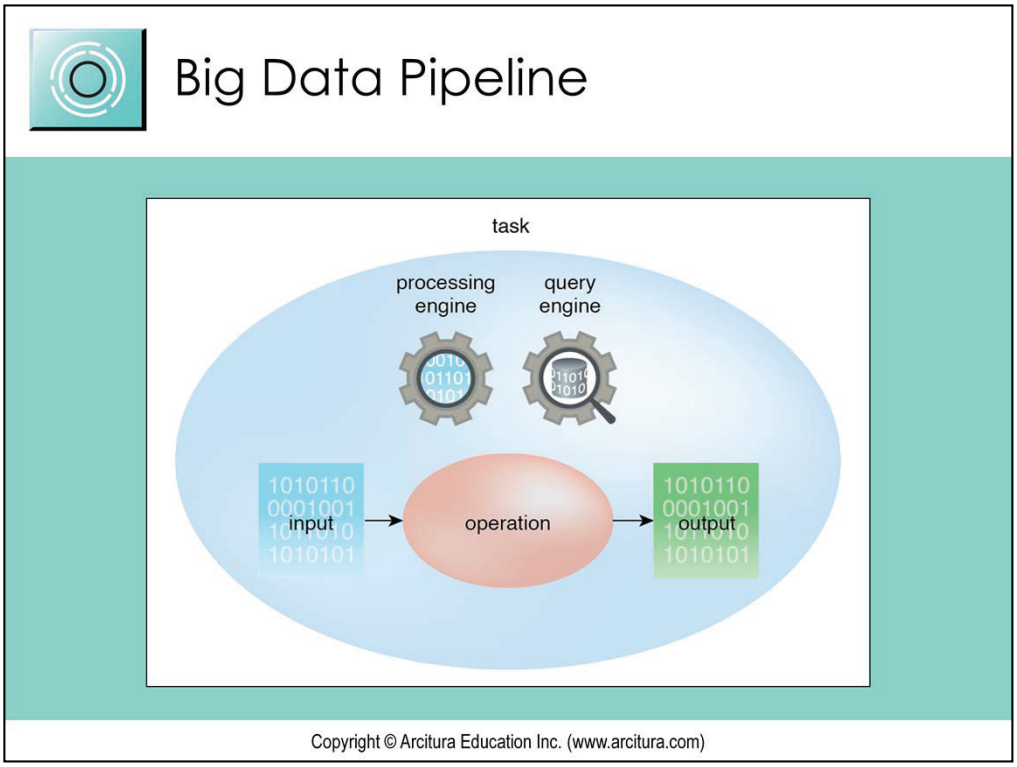
Data Pipeline

- A data pipeline may consist of a number of stages with the net effect of reading data from a source, performing some operations (multiple stages) on the input data and then writing out the processed data.
- Like an actual pipeline, a data pipeline is used to move data between the source and the sink in an automated manner, as shown in the diagram on the following page.
- This is done while performing different operations in between, such as cleansing, validating, joining, and transforming.


Data Pipeline
data pipeline
source
stage1
stage2
stagen
sink
Copyright © Arcitura Education Inc. (www.arcitura.com)

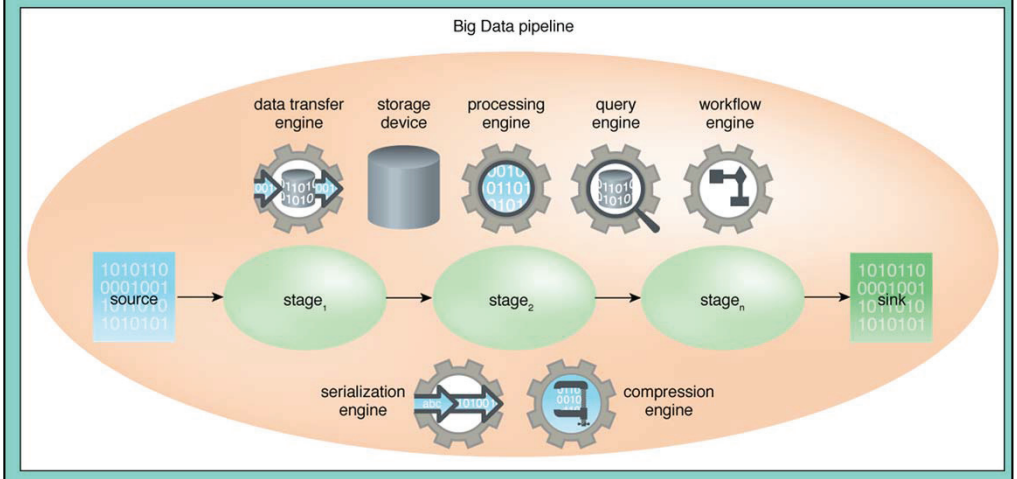
Big Data Pipeline

- A Big Data pipeline generally comprises multiple stages where complex processing is broken down into modular steps for ease of maintenance and accommodating future processing requirements.
- Each task in a Big Data pipeline generally makes use of a processing engine mechanism, such as MapReduce or Spark, or a query engine mechanism, such as Hive or Pig, as shown in the diagram on the following page.



Big Data Pipeline

The enabling mechanisms in a Big Data pipeline include the data transfer engine, storage device, processing engine, query engine, compression engine, serialization engine and workflow engine mechanisms.



Big Data Pipeline

- In a Big Data pipeline, the workflow engine acts as an orchestrator for constructing the pipeline, such as triggering the subsequent task once the preceding task is completed.
- Developing data processing solutions within Big Data environments heavily revolves around the concept of data pipelines.
- Some of the challenges faced while developing Big Data pipelines include the requirement for a scalable and reliable means of ingesting structured, unstructured, and semi-structured data, and joining structured data with unstructured data in a fully automated manner.

Big Data Pipeline

- A Big Data pipeline can be very simple, consisting of a single stage, or very complex, consisting of multiple stages.
- Within Big Data environments, data pipelines are employed whenever data needs to be modified in some manner, for example from initial data extraction to subsequent data enrichment, such as geocoding IP addresses and data denormalization.
- Generally, a Big Data pipeline represents a partial or a complete Big Data solution for Big Data analysis.
- One of the primary reasons for constructing a Big Data pipeline is to transform unstructured data into structured form in order for it to be useful to downstream systems.

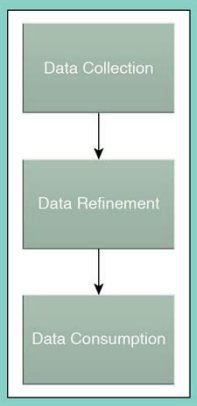
Big Data Pipeline

- To simplify and shorten the development time, various pipeline development tools can be used that can combine a subset of tasks into a micro-pipeline, such as data validation and splitting.
- As a whole, the Big Data pipeline is automated via the use of a workflow engine.
- However, before the pipeline can be deployed on the live system, it needs to pass through rigorous testing.
- The entire pipeline can first be tested in a modular fashion (unit testing) by making sure each stage produces the desired output, followed by integration testing to verify end-to-end working of the Big Data pipeline.

Big Data Pipeline: Stages

A typical Big Data pipeline consists of the following stages:

1. Data Collection
2. Data Refinement
3. Data Consumption

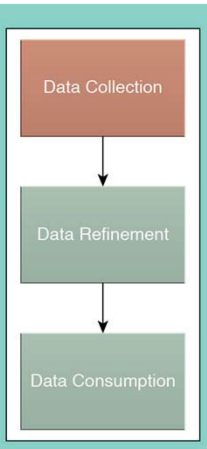


Note

It should be noted that these stages depict an end-to-end Big Data pipeline. In practice, the required stages are dependent on the data processing requirements.

Big Data Pipeline: Data Collection

- The data collection stage consists of data ingestion, filtration, compression, and storage tasks.
- The collected data is stored in raw form, using an appropriate storage device, that will later be retrieved and processed in some manner before the results are stored in a structured form.
- Data collection involves making the necessary connections for acquiring data, in other words, connecting data sources with data sinks using the data transfer engine(s).
- Data can be ingested in batch or realtime mode.



Big Data Pipeline: Data Collection

- Batch mode is generally used for relational data and file-based data that can be automated through the use of the relational and file data transfer engines respectively.
- Realtime mode is generally used for event-based data, such as sensor data or transactional data, which can be automated through the use of the event data transfer engine.
- Some basic level of filtering is generally performed at this stage to remove invalid data either using a query engine or a processing engine.
- Before storage, raw data can be compressed using a compression engine to save storage space and to achieve faster data transfer rates for later processing.

Big Data Pipeline: Data Collection

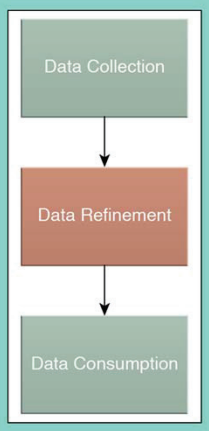
- It is also important to store data verbatim without losing any data fidelity, and to choose a suitable serialization engine that provides a storage format that can be accessed by a range of software libraries and tools.
- Gathered data is stored using an appropriate storage device, such as a NoSQL database or a distributed file system.
- Depending on the type(s) of data being captured, more than one type of storage device may be required.
- Also, depending on the processing workload requirements, either in-memory or on-disk storage devices, or in some cases a combination of both, are required.

Big Data Pipeline: Data Collection

- Depending on the processing engine being used, it may be necessary to combine together multiple small files for optimizing processing to decrease the processing time.
- This is due to the fact that a processing engine, such as MapReduce, works best with fewer but larger files. A large number of smaller files incur excessive disk-seek activity and increase memory consumption as dedicated processes are generally spawned by the processing engine for processing each file individually.

Big Data Pipeline: Data Refinement

- The refinement stage involves extraction, validation/cleansing, and joining/splitting tasks, including data model creation and data export tasks.
- Required data fields are extracted from the previously stored raw data.
- It is important to understand that it may not be known which fields need to be extracted, especially in case of exploratory analysis.
- Information about operational and strategic data requirements and business goals can help in identifying required fields.



Big Data Pipeline: Data Refinement

- A meaningful extraction criteria needs to be employed that achieves a balance between known use cases and unknown exploratory requirements. The objective is not to store extraneous or too-lean datasets.
- Extraction of data is generally required with structured and semi-structured data.
- However, with unstructured data such as images or audio files, extraction is either not required at all, or partial extraction is required, such as extracting embedded image attributes.
- Furthermore, the extracted data is validated and data that contains invalid values is removed.

Big Data Pipeline: Data Refinement

- The variety characteristic of Big Data dictates that, quite often, all required data may not be present in a single dataset, which warrants merging datasets.
- On the other hand, a single dataset may include logical groups that can be consumed separately in support of different analysis tasks, which warrants splitting a dataset into multiple datasets.
- Other tasks within this stage may include generating new fields based on existing data and data format transformation.
- Such tasks can be performed through the use of a processing engine, such as MapReduce, either directly or indirectly through a query engine.

Big Data Pipeline: Data Refinement

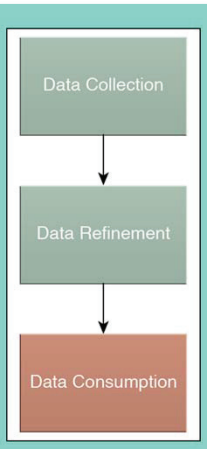
- In the case of a realtime data processing solution, the refinement stage is generally performed before the data is stored to the disk.
- In other words, the operations in this stage are performed on the in-memory data.
- Decisions regarding which type of storage device to use (for the refined data) and the most optimal schema design are also made during this stage as multiple strategies may exist for storing the same type of dataset.
- This is especially true for column-family and graph storage devices, as the same data can be stored in multiple ways.

Big Data Pipeline: Data Refinement

- For example, time series-based sensor data can either be stored as a single row with one column per reading (wide rows) or multiple rows partitioned by time, such as per day, with fewer columns (narrow rows).
- Similarly, with graph data, data size (number of vertices and edges), access patterns (transactional/batch) and use cases (search/iterative algorithms) need to be considered.

Big Data Pipeline: Data Consumption

- The consumption stage consists of tasks centered around making use of the data refined in the previous stage.
- In this stage, refined data may either be consumed directly or indirectly.
- Direct consumption involves putting models, developed by data scientists, into production by executing the underlying algorithms on live (refined) data.
- Similarly, in case of relatively simple use cases, various statistics such as *totals* and *averages* are generated from refined data and fed into downstream BI reporting systems, dashboards, and other information sinks, such as portals.



Big Data Pipeline: Data Consumption

- Indirect consumption involves provisioning refined data across multiple business functions and downstream systems.
- Specifically, it entails making sure that the refined data is in the correct format or structure so that it can be utilized by the end system.
- For example, a business intelligence tool requires data in tabular form, whereas the refined data exists in a tab delimited file format. Therefore, the refined data needs to be further processed in order to be in compliance with the required format.
- Furthermore, the refined data may need to be exported to a data warehouse in support of more traditional data analyses.

Big Data Pipeline: Data Consumption

- Irrespective of how the data is consumed, further data processing/transformation is generally required depending upon the individual use case, such as creating subsets or aggregates of data for (statistical/machine learning) model development.
- Due to the diverse nature of this stage, a host of Big Data mechanisms can be employed, including data transfer engine, query engine, processing engine, and analytics engine.

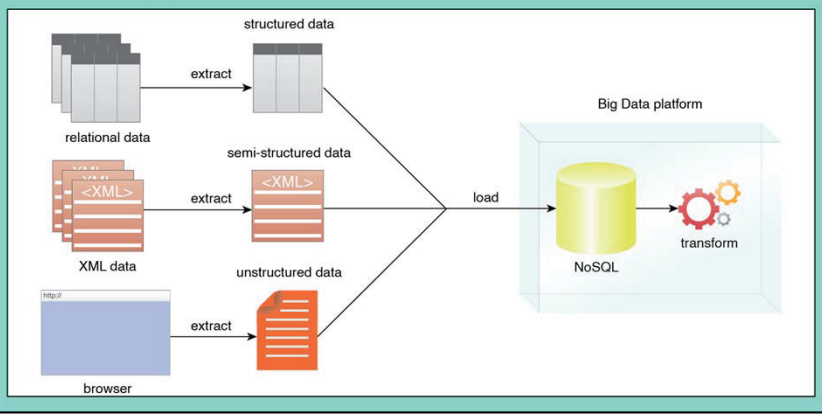
Extract-Load-Transform (ELT)

- Like ETL (introduced in Module 1), extract-load-transform (ELT) is a process of loading data from a source system into a target system.
- However, unlike ETL where the transformation takes place outside the target system and prior to populating the target system, in ELT the data is loaded as-is into the target system and is transformed within the target system.
- Use of ELT is more prevalent in Big Data environments where raw data is stored as-is, partly due to the reason that data usage scenarios are unknown.
- ELT eliminates the requirement for a staging system (database) as data can be transformed internally within the Big Data platform, such as Hadoop.

Extract-Load-Transform (ELT)

- ELT helps avoid data duplication and the need to procure additional hardware for the transform stage.
- In ETL, the transform stage is coupled with the load stage, and data is transformed according to already known consumption scenarios, whereas in ELT the transform and load stages are decoupled and data is transformed as per new, previously unknown scenarios.
- In Big Data environments, ELT generally performs better than ETL. The load stage is faster due to highly scalable data import and the transform stage is faster due to data colocation.
- ELT provides access to all data in its raw form that is ideal for exploratory analysis and building models.

- The first two stages of the Big Data pipeline, covered earlier, roughly map to the ELT concept where raw data is loaded into the distributed storage without any transformation, and the refinement takes place within the Big Data platform without the need to export data into another system.
- Required mechanisms generally include the data transfer engine, storage device, query engine, processing engine, and workflow engine.



Exercise

Exercise 8.5

Fill in the Blanks

Big Data Solution Design

Big Data Solution: Definition

- A Big Data solution is a data-driven application that processes structured, semi-structured and unstructured data acquired from a variety of data sources.
- These include traditional data sources, such as OLTP applications and data warehouses, as well as contemporary data sources, such as social media, Internet of Things (IoT) and machine logs.

Big Data Solution: Use Cases

- The primary use case for Big Data solutions can be thought of as an end-to-end solution that ingests Big Data datasets, performs some analysis and then either communicates the results via a dashboard or forwards the results to another application.
- The secondary use case can be thought of as a solution that works in a supportive capacity where its role is to facilitate other applications or systems. Such a solution ingests Big Data datasets, processes them to create summarized data, or converts them to a structured format and then exports summarized structured data to a more traditional application or system for further processing.

Big Data Solution: Characteristics

- Unlike traditional enterprise applications, such as CRM and ERP systems where the applications are data-producers, Big Data solutions in general are data-consumers.
- However, they create additional data, which can serve as an input to traditional applications and analytic products.
- Although traditional reporting applications such as BI tools are data consumers, they act on structured data and do not generally create further data for consumption by other applications.

Big Data Solution: Characteristics

- In the majority of cases, multiple heterogeneous datasets are joined together to create meaningful unified datasets that are of higher value than the individual datasets on their own.
- Unlike traditional applications, the input and output may evolve over time as more datasets become available.
- Similarly, the scope of a Big Data solution may evolve over time as further use cases are discovered.
- Although the exact nature of the Big Data solution depends on the business requirements, its main task is to make data amenable to various downstream applications, such as dashboards and OLAP/OLTP systems, and to enable data analysis via the execution of various models, such as statistical models and machine learning algorithms.

Big Data Solution: Characteristics

- In some situations, the source may be the same as the sink, such as importing daily sales data from an e-commerce system, generating the sales figures per product, and then exporting back the results into the e-commerce system for reporting purposes.
- Big Data solutions enhance traditional applications through a feedback loop.
- For example, predictions generated by a Big Data solution are fed into a web application and are refined through the feedback received from customers' interactions.

Big Data Solution: Design Considerations

- A Big Data solution needs to cater to the variety characteristic of Big Data by providing storage support for multi-structured and multi-format data captured from different data sources and the ability to process data whose schema may not be known in advance.
- A Big Data solution needs to scale massively in three different areas:
  - collection of data, especially in the case of machine generated data or IoT
  - storage of data
  - processing of data

Big Data Solution: Design Considerations

- To remain scalable, Big Data solutions are designed as distributed solutions spread across multiple commodity machines in a cluster. Consequently, machine and communication failures need to be addressed within the solution design for ensuring high availability.
- Note that with a Big Data appliance, which is a proprietary Big Data platform comprising enterprise-grade machines and interconnect, high availability becomes less of an issue, and instead cost and scalability become more of a concern.
- Furthermore, data replication and sharding need to be implemented in support of fault-tolerance, high availability and scalability.

Big Data Solution: Design Considerations

- Depending on the data processing requirements, a Big Data solution may need to incorporate support for both batch and realtime processing.
- For example, a Big Data solution analyzes customer comments in realtime for enhanced customer care while the same comments are combined with purchase history for performing pattern mining (a batch process) in support of finding recurring causes behind product return.

Big Data Solution: Design Considerations

- Testing and debugging Big Data solutions requires special consideration. This is because Big Data solutions generally consist of complex data pipelines comprising multiple stages.
- Consequently, before deploying to a cluster, a Big Data solution can be executed in a pseudo-distributed mode (a single node execution mode that mimics a cluster).
- This way, each stage can be tested or debugged comparatively easily as all log files are located on the same machine.

Big Data Solution: Design Considerations

- For both performance tuning and debugging, logging needs to be performed at various stages of a complex Big Data solution and set to an appropriate level, such as INFO or DEBUG when using log4j. This helps to get as much information as possible about the runtime behavior of the Big Data solution, thereby helping in faster and more effective debugging.
- Use of subsets of large datasets as inputs further eases debugging. This not only helps to find logic issues but also data issues, such as badly formatted data or garbage records.
- The pseudo-distributed mode is also ideal for prototype development especially if the enterprise is embarking on a Big Data project for the very first time.

Big Data Solution: Design Considerations

- Source data needs to be kept in the lowest common denominator form for catering multiple use cases. This means that the source data should be cleansed and validated. However, the source data should be saved before performing any further operations such as data model/format transformation or joining datasets together.
- Depending on the data structure, a corresponding processing engine should be used. For example, for graph data, a processing engine that supports message passing capabilities between nodes should be used, whereas MapReduce can be used for situations where data is not related to each other and each record exists as a stand-alone aggregate that can be processed independently.

Big Data Solution: Design Process

1. Establish and evaluate data inputs and outputs
   - Identify required data sources.
   - Establish access to data sources via data transfer engine(s).
   - Choose the right type of storage device(s).
   - Identify how output needs to be persisted by assessing the downstream application(s).
   - Establish access criteria (authentication and authorization) both for the persisted input as well as the output.

Big Data Solution: Design Process

2. Determine data wrangling requirements
   - Gauge the quality of the input data.
   - Develop data pipeline(s) for data parsing, cleansing, filtering, joining and format/model conversion.
3. Select data representation format
   - Store the massaged data from the previous step by choosing a data structure that can be optimally processed by the data processing engine(s).
   - Assess the query patterns in case the objective is to provide access to raw, cleansed data via SQL-like queries, such as during exploratory data analysis.

Big Data Solution: Design Process

4. Assess processing engine suitability
   - Evaluate the data processing mode in the light of the time-to-results requirement (realtime, near realtime or offline/batch mode).
   - Consider both the storage device's storage model and the processing algorithm's characteristics in order to select an appropriate processing engine. For example, data stored in a graph database and a requirement for an iterative algorithm will cause the selection of a processing engine that supports BSP-based processing.

Big Data Solution: Design Process

5. Develop data processing routines
   - Depending upon the nature of the processing, write code using a software development language, such as Java, Python etc., or write scripts using a scripting language, such as Pig Latin.
   - A typical sequence of code involves reading in the input data from a storage device, performing some data manipulation functions on the data and then writing out the processed data to a storage device.
   - Depending upon the Big Data platform capabilities, this may simply be a point-and-click exercise and in other cases, instead of writing code, it may require incorporating an algorithm via its API.

Big Data Solution: Design Process

6. Develop visualizations
   - Create various graphical artifacts for displaying the processed data in a user friendly format and in support of performing various analyses.
   - Various charting libraries are generally incorporated through their APIs for constructing dashboard-like front ends.

Big Data Solution: Design Process

7. Automate solution execution
   - A Big Data solution generally executes continuously, as in realtime mode, or at regular intervals, as in batch mode, in an automated manner.
   - Having developed the Big Data solution, necessary configuration or code needs to be in place that ensures automated data ingress, processing and egress operations.
   - This may further involve automated refresh of data visualizations, such as dashboards.

Big Data Solution: Design Process

7. Automate solution execution (continued)

- As part of automating solution execution, creating connections for establishing any necessary feedback loop(s), as in the case of some machine learning algorithms, may also be required.
- Although described as the last step, this step may also be performed while establishing the necessary input/output connections and developing the necessary processing routines.

Big Data Solution: Simple

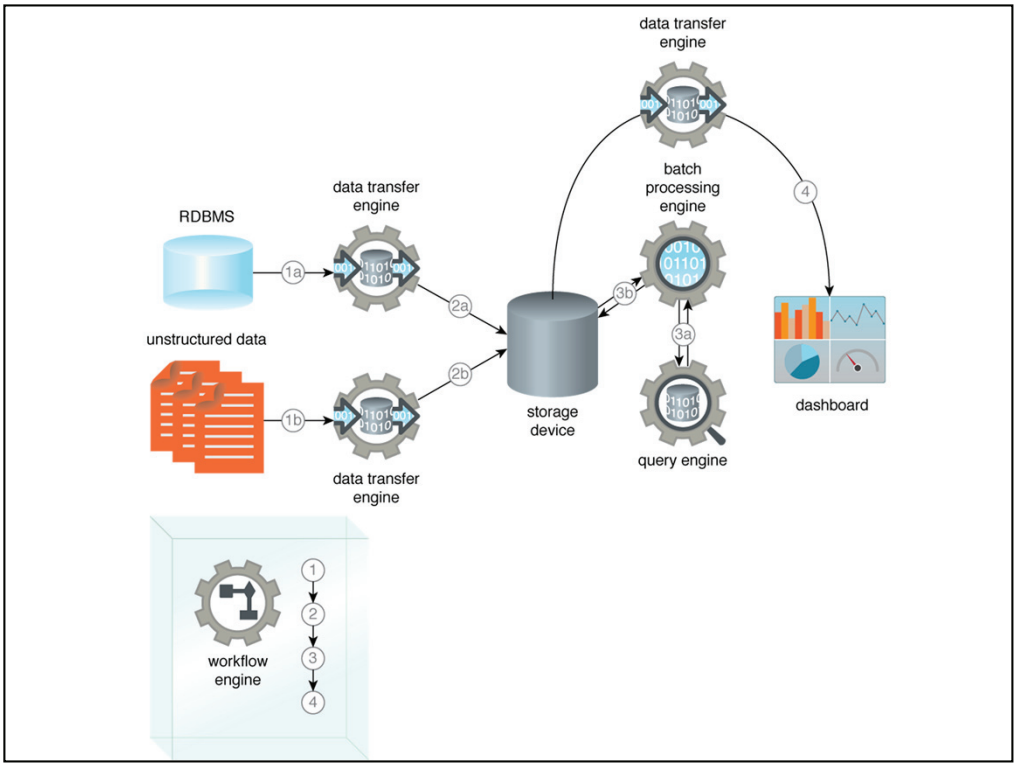
- Traditional applications are developed with a particular goal in mind.
- However, due to the exploratory nature of Big Data analysis, the associated goals with a Big Data solution may change over time, making it difficult to design Big Data solutions.
- A Big Data solution may start its life as a simple number crunching application, but over a period of time it may evolve into a data product.
- In general, a simple Big Data solution is a batch data processing application with a comparatively small number of inputs that generates certain statistics or performs simple data transformation tasks.

Big Data Solution: Simple

The Big Data mechanisms generally involved in the design of a simple Big Data solution include:

- Data transfer engine (file/relational)
- On-disk storage device
- Processing engine (batch)
- Query engine
- Workflow engine

The diagram on the next page shows how these mechanisms are generally connected together. Note that the resource manager, although part of the Big Data solution, is not included in order to portray a simpler design.


data transfer engine
batch processing engine
RDBMS
data transfer engine
1a
2a
unstructured data
1b
2b
storage device
3b
3a
4
dashboard
query engine
data transfer engine
workflow engine
1
2
3
4

In the diagram on the previous page:

1. (a, b). A relational and a file data transfer engines are used to acquire data generally from within the enterprise, such as transactional data and web server log files.
2. (a, b). The data is stored using on-disk storage devices, including distributed file system and NoSQL database.
3. (b, a). In support of the required computational requirements, a batch processing engine is used either directly or indirectly via a query engine.
4. The processed results are then exported via the data transfer engine to a downstream application such as a dashboard.

A workflow engine is used so that data ingress, processing, and egress activities take place repetitively as required without any human intervention.

Big Data Solution: Complex

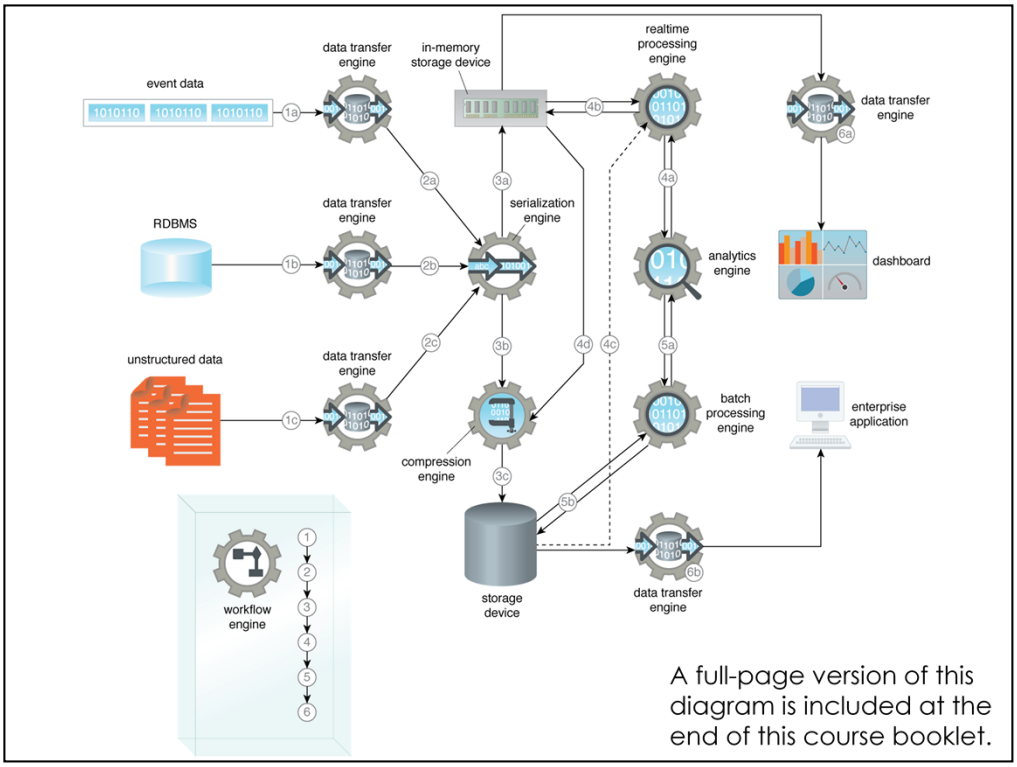
- In contrast to a simple Big Data solution, a complex Big Data solution may consist of both batch and realtime components such as in-memory/on-disk storage devices and realtime/batch processing engines.
- In general, a complex Big Data solution ingresses data from multiple data sources in order to perform complex data processing.
- The processing itself may involve creating multi-stage data pipelines that execute machine learning or other complex algorithms.
- The processing output can be exported to a dashboard or some other downstream application that incorporates the processing results, such as a risk detecting and mitigating application.

Big Data Solution: Complex

The Big Data mechanisms generally involved in the design of a complex Big Data solution include:

- Data transfer engine (event/file/relational)
- Storage device (on-disk/in-memory)
- Processing engine (batch/realtime)
- Analytics engine
- Serialization engine
- Compression engine
- Workflow engine

The diagram on the next page shows how these mechanisms are generally connected together.


event data
1010110
1010110
1010110
1a
data transfer engine
in-memory storage device
realtime processing engine
4b
data transfer engine
6a
2a
3a
4a
RDBMS
1b
data transfer engine
2b
serialization engine
analytics engine
dashboard
unstructured data
1c
data transfer engine
2c
3b
4d
4c
5a
batch processing engine
enterprise application
compression engine
3c
5b
workflow engine
1
2
3
4
5
6
storage device
data transfer engine
6b
A full-page version of this diagram is included at the end of this course booklet.

In the diagram on the previous page:

1. (a). An event data transfer engine is used to acquire data generally from within the enterprise, such as sensor data, or from outside the enterprise, such as social media data.

   (b, c). Relational and file data transfer engines are used to acquire data generally from within the enterprise, such as transactional data and web server log files, or from outside the enterprise, such as census data and genome data.

2. (a, b, c). Imported data is then serialized into a format that is suitable for subsequent data processing.

3. (a). Event data is persisted to an in-memory storage device.

   (b, c). Unstructured data is compressed first and then persisted to an on-disk storage device. Relational data may also be compressed first before storage.

Big Data Solution: Complex

4. (a, b). In-memory data is processed directly using a realtime processing engine, or indirectly via an analytics engine.

   (c). The realtime processing engine may also combine data from the on-disk storage device.
   (d). Raw in-memory data is compressed and persisted to the on-disk storage device, such as a distributed file system or a NoSQL database, where it can be joined with other datasets for further detailed analysis.

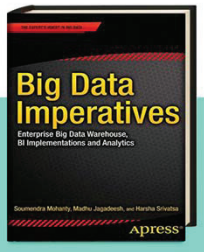
5. (a, b). On the other hand, on-disk data is processed directly using a batch processing engine or indirectly via an analytics engine for executing complex algorithms, such as a clustering algorithm for customer segmentation in support of a targeted marketing campaign.

Big Data Solution: Complex

6. (a, b). The data processing results are then exported from the in-memory storage and on-disk storage via the relevant data transfer engine to a downstream application such as a dashboard or some other enterprise application.

A workflow engine automates the data ingress, processing, and egress activities, so that these activities take place repetitively as required without any human intervention.

Reading



"Big Data Imperatives"

Developing a Recommendation System
Pages 244 - 249

Conclusion and Q&A

Exam B90.08

The course you just completed corresponds to Exam B90.08, which is an official exam that is part of the Big Data Science Certified Professional (BDSCP) program.

PEARSON VUE

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/

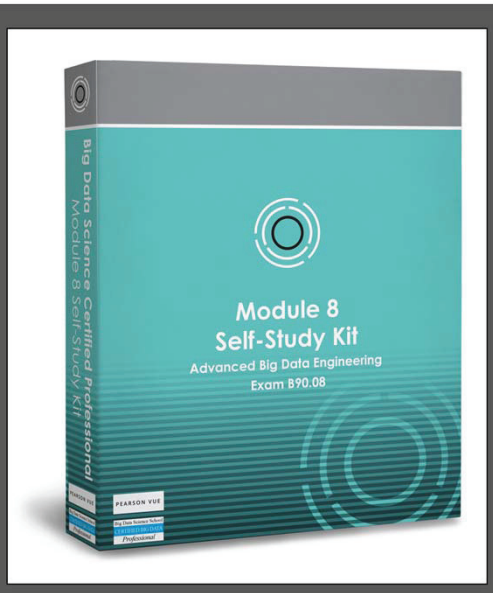
www.pearsonvue.com/arcitura/op/ (Online Proctoring)

Module 8 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 B90.08.

For more information, visit:
www.bigdataselfstudy.com

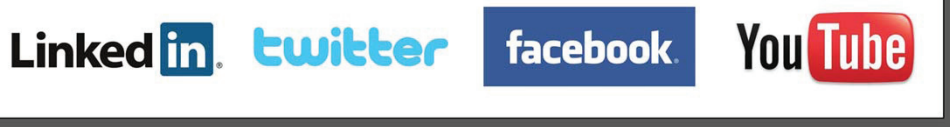


AITCP Community

Join the growing international Arcitura IT Certified Professional (AITCP) community by connecting with official social media channels available via LinkedIn, Twitter, Facebook and YouTube.

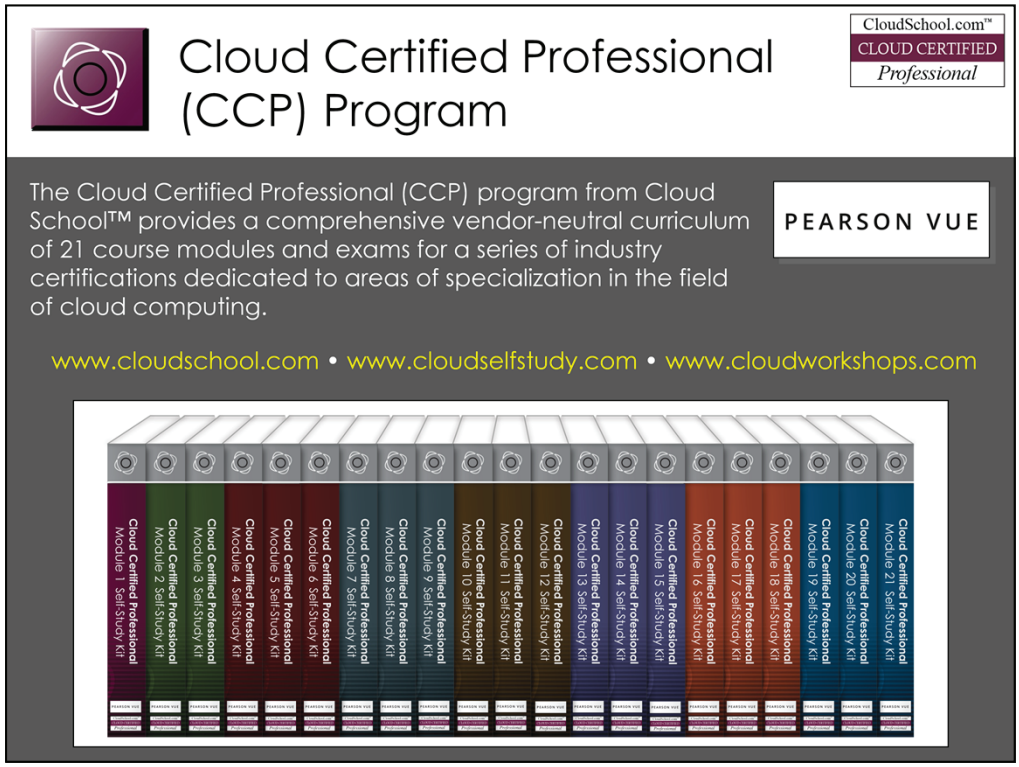
Social media and community links are accessible at:

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



Contact & Resources

Arcitura Education Inc.	www.arcitura.com
Big Data School	www.bigdatascienceschool.com
Workshops	www.bigdataworkshops.com
Self-Study Kits	www.bigdataselfstudy.com
Service Technology Book Series	www.servicetechbooks.com
Service Technology Standards	www.servicetechspecs.com
Big Data Patterns	www.bigdatapatterns.org
General Inquiries	info@arcitura.com
Automatic Updates	notify@arcitura.com
Becoming a Trainer/Partner	partners@arcitura.com


Cloud Certified Professional (CCP) Program
CloudSchool.com™
CLOUD CERTIFIED
Professional
The Cloud Certified Professional (CCP) program from Cloud School™ provides a comprehensive vendor-neutral curriculum of 21 course modules and exams for a series of industry certifications dedicated to areas of specialization in the field of cloud computing.
PEARSON VUE
www.cloudschool.com • www.cloudselfstudy.com • www.cloudworkshops.com
Cloud Certified Professional Module 1 Self-Study Kit
Cloud Certified Professional Module 2 Self-Study Kit
Cloud Certified Professional Module 3 Self-Study Kit
Cloud Certified Professional Module 4 Self-Study Kit
Cloud Certified Professional Module 5 Self-Study Kit
Cloud Certified Professional Module 6 Self-Study Kit
Cloud Certified Professional Module 7 Self-Study Kit
Cloud Certified Professional Module 8 Self-Study Kit
Cloud Certified Professional Module 9 Self-Study Kit
Cloud Certified Professional Module 10 Self-Study Kit
Cloud Certified Professional Module 11 Self-Study Kit
Cloud Certified Professional Module 12 Self-Study Kit
Cloud Certified Professional Module 13 Self-Study Kit
Cloud Certified Professional Module 14 Self-Study Kit
Cloud Certified Professional Module 15 Self-Study Kit
Cloud Certified Professional Module 16 Self-Study Kit
Cloud Certified Professional Module 17 Self-Study Kit
Cloud Certified Professional Module 18 Self-Study Kit
Cloud Certified Professional Module 19 Self-Study Kit
Cloud Certified Professional Module 20 Self-Study Kit
Cloud Certified Professional Module 21 Self-Study Kit


SOA Certified Professional (SOACP) Program
SOASchool.com®
SOA CERTIFIED
Professional
The SOA Certified Professional (SOACP) program from SOA School® provides a comprehensive vendor-neutral curriculum of 23 course modules and exams for a series of industry certifications dedicated to areas of specialization in the fields of SOA and service-oriented computing.
PEARSON VUE
www.soaschool.com • www.soaselfstudy.com • www.soaworkshops.com

Arcitura Education Inc. is a leading global provider of progressive, vendor-neutral training and certification programs. With a worldwide network of certified instructors, licensed training partners and testing centers, Arcitura schools and accreditation programs have become internationally established.

If you don't already have an online AITCP profile, visit **www.arcitura.com/aitcp** for more information.

Arcitura
the IT education company

www.arcitura.com

Exercise 8.1
Fill in the Blanks

1. In Big Data platforms, _____ is required to enable message exchange between machines and for persisting data.

2. When choosing a compression engine mechanism, it is important to understand the relationship between _____, _____ and the required _____.

3. Preferably, a serialization engine mechanism should serialize/deserialize data at a _____ speed with _____ size reduction, be _____ to future changes, and can work with a variety of data producers and consumers.

4. A _____ compression engine will generally provide more compact data requiring more processing resources while a _____ compression engine mechanism will generally provide less compact data requiring fewer processing resources.

5. When using a distributed file system, data needs to be serialized since working with _____ bytes is not only difficult but also introduces _____ issues.

Exercise 8.2
Match Terms to Statements

Answer the questions below by filling in each blank with one of the following terms (note that not all terms are used):

- on-disk storage device
- in-memory storage device
- in-memory data grid device
- in-memory database device
- read-through approach
- write-through approach
- write-behind approach
- refresh-ahead approach

1. Bob needs to add realtime capability to an existing data processing application. The current application uses an RDBMS and the data being stored consists of relational data from various OLTP systems. Due to the legacy nature of the application, changes cannot be made to the data access layer. Which in-memory storage device can Bob use to add realtime support?

   _____

2. Suzan has been asked to develop a prototype solution that performs sentiment analysis on comments left by customers across different social media Web sites. The data to be stored is semi-structured. Suzan has a limited budget and wants to keep the prototype simple as this is her first Big Data project. Which type of storage device can Suzan use for implementing the required functionality?

   _____

3. Keith is developing an operational dashboard for IT personnel that displays the health status of various IT assets across the enterprise, including servers and switches, in realtime. Various statistics are generated from the log files that are imported every five minutes. The current storage architecture consists of an on-disk NoSQL database that Keith intends to keep as it is heavily used by a number of applications. Which type of in-memory storage device can be used in this scenario?

   _____

4. Anthony has added realtime support to a data processing application by augmenting the existing data storage tier with an in-memory data grid. Recently, the number of client applications that access the in-memory data grid has grown considerably, with each maintaining multiple connections to the in-memory data grid. Users have started complaining about the performance when reading from and writing to the in-memory data grid. Which integration approach can Anthony use to reduce both the data read and write latency?

   _____

5. Justin is implementing an in-memory data grid on top of an existing NoSQL database. Justin does not have much experience with in-memory data grids and wants to start with a simple architecture that employs a simple approach for integrating the in-memory data grid with the NoSQL database. There are several applications that read data from the existing NoSQL database and these applications require up-to-date access to new data as it gets written to the in-memory data grid. Which integration approach can Justin use to fulfill the current requirements?

   _____

Exercise 8.3
Identify True/False Statements

Validate which of the following statements is true or false:

1. Polyglot persistence provides an optimum storage environment. However, when using polyglot persistence, management of different types of storage devices can become a concern. (True/False)

2. Polyglot persistence refers to a heterogeneous storage environment that always requires more than one storage device. (True/False)

3. Polyglot persistence refers to using only different types of NoSQL databases. (True/False)

4. In a polyglot persistence environment, implementation of services not only standardizes API access to various storage devices but also provides means for securing the storage devices. (True/False)

5. SCV principle only applies to distributed data processing systems. (True/False)

6. In light of the SCV principle, within a Big Data environment, a data processing application is either SC or CV. (True/False)

7. Realtime Big Data processing only refers to processing data that has just been ingested and excludes historic data that was ingested previously. (True/False)

8. Event stream processing (ESP) generally focuses on analysis of single source, time ordered events and involves the application of comparatively simple algorithms. (True/False)

9. Complex event processing (CEP) involves analysis of a number of events from disparate sources and employs comparatively complex algorithms. (True/False)

10. In MapReduce, a reduce task generally cannot start before the completion of all map tasks. The output from the map task is persisted locally on each node that runs the map function before being copied over the network to the nodes running the reduce function, introducing processing latency. (True/False)

Exercise 8.4
Organize the BSP Stages

Organize the following BSP stages into the correct order:

barrier synchronization	1. _____
local computation	2. _____
preprocessing	3. _____
communication	4. _____

Exercise 8.5
Fill in the Blanks

1. Data pipeline is a _____ workflow consisting of multiple tasks where each task involves _____, _____ and _____.

2. A data pipeline is used for moving data between the _____ and the _____ in an automated manner while performing different _____.

3. One of the primary reasons for constructing a Big Data pipeline is to transform _____ data into _____ form in order for it to be useful for downstream systems.

4. A typical Big Data pipeline consists of _____, _____ and _____ stages.

5. The _____ stage involves extraction, validation/cleansing and joining/splitting tasks.

6. As part of the _____ stage, further data processing/transformation is generally required.

7. The _____ stage consists of data ingestion, filtration, compression and storage tasks.

8. Like ETL, _____ is a process of loading data from a source system into a target system.

9. ELT eliminates the requirement for a _____ as data can be transformed internally within the Big Data platform.

Exercise 8.1 Answers

1. In Big Data platforms, **serialization** is required to enable message exchange between machines and for persisting data.

2. When choosing a compression engine mechanism, it is important to understand the relationship between **compression speed**, **compactness level** and the required **processing resources**.

3. Preferably, a serialization engine mechanism should serialize/deserialize data at a **fast** speed with **maximum** size reduction, be **amenable** to future changes, and can work with a variety of data producers and consumers.

4. A **slow** compression engine will generally provide more compact data requiring more processing resources while a **fast** compression engine mechanism will generally provide less compact data requiring fewer processing resources.

5. When using a distributed file system, data needs to be serialized since working with **raw** bytes is not only difficult but also introduces **interoperability** issues.

Exercise 8.2 Answers

1. Bob needs to add realtime capability to an existing data processing application. The current application uses an RDBMS and the data being stored consists of relational data from various OLTP systems. Due to the legacy nature of the application, changes cannot be made to the data access layer. Which in-memory storage device can Bob use to add realtime support?

   in-memory database device

2. Suzan has been asked to develop a prototype solution that performs sentiment analysis on comments left by customers across different social media Web sites. The data to be stored is semi-structured. Suzan has a limited budget and wants to keep the prototype simple as this is her first Big Data project. Which type of storage device can Suzan use for implementing the required functionality?

   on-disk storage device

3. Keith is developing an operational dashboard for IT personnel that displays the health status of various IT assets across the enterprise, including servers and switches, in realtime. Various statistics are generated from the log files that are imported every five minutes. The current storage architecture consists of an on-disk NoSQL database that Keith intends to keep as it is heavily used by a number of applications. Which type of in-memory storage device can be used in this scenario?

   in-memory data grid device

4. Anthony has added realtime support to a data processing application by augmenting the existing data storage tier with an in-memory data grid. Recently, the number of client applications that access the in-memory data grid has grown considerably, with each maintaining multiple connections to the in-memory data grid. Users have started complaining about the performance when reading from and writing to the in-memory data grid. Which integration approach can Anthony use to reduce both the data read and write latency?

   write-behind approach

5. Justin is implementing an in-memory data grid on top of an existing NoSQL database. Justin does not have much experience with in-memory data grids and wants to start with a simple architecture that employs a simple approach for integrating the in-memory data grid with the NoSQL database. There are several applications that read data from the existing NoSQL database and these applications require up-to-date access to new data as it gets written to the in-memory data grid. Which integration approach can Justin use to fulfill the current requirements?

   write-through approach

Exercise 8.3 Answers

1. Polyglot persistence provides an optimum storage environment. However, when using polyglot persistence, management of different types of storage devices can become a concern. (**True**)

2. Polyglot persistence refers to a heterogeneous storage environment that always requires more than one storage device. (**False**)

3. Polyglot persistence refers to using only different types of NoSQL databases. (**False**)

4. In a polyglot persistence environment, implementation of services not only standardizes API access to various storage devices but also provides means for securing the storage devices. (**True**)

5. SCV principle only applies to distributed data processing systems. (**True**)

6. In light of the SCV principle, within a Big Data environment, a data processing application is either SC or CV. (**False**)

7. Realtime Big Data processing only refers to processing data that has just been ingested and excludes historic data that was ingested previously. (**False**)

8. Event stream processing (ESP) generally focuses on analysis of single source, time ordered events and involves the application of comparatively simple algorithms. (**True**)

9. Complex event processing (CEP) involves analysis of a number of events from disparate sources and employs comparatively complex algorithms. (**True**)

10. In MapReduce, a reduce task generally cannot start before the completion of all map tasks. The output from the map task is persisted locally on each node that runs the map function before being copied over the network to the nodes running the reduce function, introducing processing latency. (**True**)

Exercise 8.4 Answers

Organize the following BSP stages into the correct order:

1. **preprocessing**

2. **local computation**

3. **communication**

4. **barrier synchronization**

Exercise 8.5 Answers

1. A data pipeline is a **data-driven** workflow consisting of multiple tasks where each task involves **input**, **operation** and **output**.

2. A data pipeline is used for moving data between the **source** and the **sink** in an automated manner while performing different **operations**.

3. One of the primary reasons for constructing a Big Data pipeline is to transform **unstructured** data into **structured** form in order for it to be useful for downstream systems.

4. A typical Big Data pipeline consists of **data collection**, **data refinement** and **data consumption** stages.

5. The **data refinement** stage involves extraction, validation/cleansing and joining/splitting tasks.

6. As part of the **data consumption** stage, further data processing/transformation is generally required.

7. The **data collection** stage consists of data ingestion, filtration, compression and storage tasks.

8. Like ETL, **extract-load-transform (ELT)** is a process of loading data from a source system into a target system.

9. ELT eliminates the requirement for a **staging system (database)** as data can be transformed internally within the Big Data platform.