Course Book

BIG DATA TECHNOLOGIES

DLBDSBDT01

INTERNATIONAL
UNIVERSITY OF
APPLIED SCIENCES

BIG DATA TECHNOLOGIES

MASTHEAD

Publisher:
IU Internationale Hochschule GmbH
IU International University of Applied Sciences
Juri-Gagarin-Ring 152
D-99084 Erfurt

Mailing address:
Albert-Proeller-Straße 15-19
D-86675 Buchdorf
media@iu.org
www.iu.de

DLBDSBDT01
Version No.: 001-2023-1011
Prof. Dr. Christian Müller-Kett

© 2022 IU Internationale Hochschule GmbH
This course book is protected by copyright. All rights reserved.
This course book may not be reproduced and/or electronically edited, duplicated, or distributed in any kind of form without written permission by the IU Internationale Hochschule GmbH.
The authors/publishers have identified the authors and sources of all graphics to the best of their abilities. However, if any erroneous information has been provided, please notify us accordingly.

TABLE OF CONTENTS

BIG DATA TECHNOLOGIES

INTRODUCTION

WELCOME

SIGNPOSTS THROUGHOUT THE COURSE BOOK

This course book contains the core content for this course. Additional learning materials can be found on the learning platform, but this course book should form the basis for your learning.

The content of this course book is divided into units, which are divided further into sections. Each section contains only one new key concept to allow you to quickly and efficiently add new learning material to your existing knowledge.

At the end of each section of the digital course book, you will find self-check questions. These questions are designed to help you check whether you have understood the concepts in each section.

For all modules with a final exam, you must complete the knowledge tests on the learning platform. You will pass the knowledge test for each unit when you answer at least 80% of the questions correctly.

When you have passed the knowledge tests for all the units, the course is considered finished and you will be able to register for the final assessment. Please ensure that you complete the evaluation prior to registering for the assessment.

Good luck!

BASIC READING

John, T. & Misra, P. (2017). *Data Lake for Enterprises* (1st ed.). Packt Publishing.

Kleppmann, M. (2017). Designing data-intensive applications: the big ideas behind reliable, scalable, and maintainable systems. *O'REILLY*.

FURTHER READING

UNIT 1

Luntovskyy, A. & Globa, L. (2019). Big Data: Sources and Best Practices for Analytics. International Conference on Information and Telecommunication Technologies and Radio Electronics (UkrMiCo), 2019, pp. 1-6, doi: 10.1109/UkrMiCo47782.2019.9165334.

UNIT 2

Avinash, N., Armando, F., and Ivan, I. (2021). Python Data Analysis. Packt Publishing. Chapter 6

Plase, D., Niedrite, L., and Taranovs, R. (2017). A Comparison of HDFS Compact Data Formats: Avro vs. Parquet. Science: Future of Lithuania. 2017, Vol. 9 Issue 3, p267-276. 10p.

UNIT 3

Hainaut, J.-L. (2009). *Encyclopedia of Database Systems.* Retrieved from Springer, Boston: https://doi.org/10.1007/978-0-387-39940-9_246

Stonebraker, M. (2010, April). SQL Databases v. NoSQL Databases. *Communications of the ACM, 53*(4), 10-11. http://doi:10.1145/1721654.1721659

UNIT 4

Marz, N., and Warren, J. (20). Big Data: Principles and Best Practices of Scalable Realtime Data Systems. Manning Publications. Part I – Batch layer

Van Steen, M. and Tenenbaum, A. S. (2017): Distributed Systems. Self Publshed (https://www.distributed-systems.net/index.php/books/ds3/) – Chapter 7 – Consistency and Replication

UNIT 5

Patil, N.V., Krishna, C.R. & Kumar, K. SSK-DDoS: distributed stream processing framework based classification system for DDoS attacks. *Cluster Comput* (17.01.2022). https://doi-org.pxz.iubh.de:8443/10.1007/s10586-022-03538-x

LEARNING OBJECTIVES

We live in times of the Data Revolution, and big data systems have become a part of our societies and way of life to the extent that it is hard to imagine how the world would look without these technologies. In this course, you will start to understand the concepts that enable the storage, processing, and analysis of vast and fast datasets, being the backbone of modern data-intensive applications.

To begin this course, we start with an introduction to data types and data sources. Here, you will learn about the 4V's that define big data, being data with large volume, high velocity, variety, and veracity. Next, you will learn about different data sources to feed modern data-intensive systems. As these data can be diverse, you will also learn about different common data types and data formats. Here, we will distinguish between text-based formats, like CSV, YAML, XML, and JSON, and binary data formats, such as HDF5, Apache Parquet, and Arrow, explicitly developed for big data applications.

Following this, you will learn about modern data storage and compute solutions designed for big data use cases. You will learn about NoSQL data stores, how they are different from relational databases, and how we categorize them into four basic types: key-value-, document-, column-, and graph-oriented databases.

Next, you will learn about distributed systems, such as the technologies subsumed under the heading of the Hadoop ecosystem. As one of the fundamental cornerstones of big data technologies, the Hadoop ecosystem provides technical solutions for big data storage and computing. You will learn how the Hadoop distributed file system (HDFS) constitutes the distributed storage layer, and the MapReduce engine can serve as the distributed computational layer of a big data system. Being a widespread, flexible, and easy-to-use big data processing technology, you will also learn about Spark, how it works, and how to use it with a scripting language like Python. In addition, not an entire big data environment like Hadoop but a Python package that is straightforward to install and use, you will learn about Dask to process big data in Python.

In the last unit of this course, you will get familiar with two prevalent frameworks for processing data streams, i.e., Spark Streaming and Apache Kafka.

At the end of this course, you will feel comfortable working with the digital gold of our time: big data.

UNIT 1

DATA TYPES AND DATA SOURCES

STUDY GOALS

On completion of this unit, you will be able to ...

- describe the characteristics of Big Data
- identify data flows that may overload a non-distributed system
- decide how to efficiently store data of different origins
- apply the acquired skills in creating data collections

1. DATA TYPES AND DATA SOURCES

Case Study

Hot&Cold Corp. is a multinational company that collects and processes weather data all over the world. They own and operate many small weather stations, sometimes in remote areas where internet service can be limited. To circumvent this issue, the weather stations use an IoT messaging protocol to deliver the data. Some of the larger weather stations come with a doppler radar system that creates image data every few seconds. In addition, the company has just launched their first satellite into space. This satellite gathers image data from space and constantly sends the data to earth. When the satellite enters an area without a radio communication connection, the data is queued and transmitted when the satellite establishes a new connection. The data is then sold to weather services, which use it to create local weather forecasts and reports.

Hot&Cold Corp. is now trying to grow their services and would like to connect 3rd party weather stations and satellites to their platform. They have hired you to get their data systems ready for 3rd party data entry. The project lead has asked you to create an overview over data volume, velocity, variety, and veracity. Which data is are structured, semi-structured or unstructured? Which are sensible data types to assign to temperature values, wind speed, precipitation information and comments entered by local staff at large stations? How would you store image information from doppler radar systems? Can you identify any possible threats that may lead to data corruption?

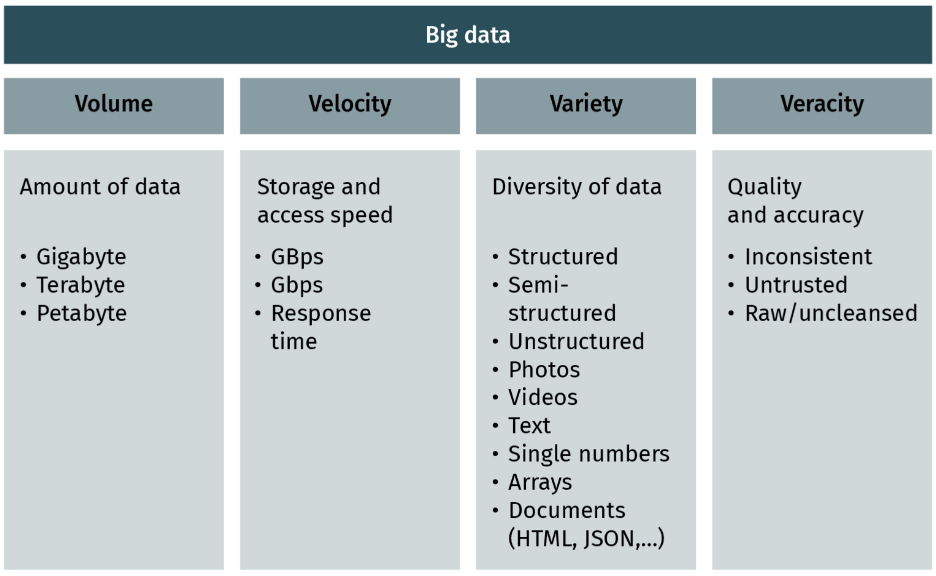
The data needs to be stored in an appropriate database. In this unit, ask yourself whether the data will conform to a predefined dataset, then pick a relational or a document-based database.

1.1 The 4Vs of data: volume, velocity, variety, veracity

The 4Vs of data can be described as the characteristics of Big Data. They describe how much data is stored, how fast it can be accessed, what kind of data is stored and the quality and accuracy of big data (Kitchin & McArdle, 2016).

The following figure visualizes the characteristics which are discussed in detail in the following.

Figure 1: Characteristics of Big Data



Source: Robert Horrion, 2022 based on John & Misra, 2017

Volume

The **volume** represents the size of the data that is stored and available for access and processing. Data-intensive applications have found their way into all of our everyday lives and the data volume available on servers worldwide grows with breath-taking speed. Units commonly used to measure the volume are gigabytes, terabytes, and petabytes. In some cases, such large amounts of data can be stored that the required measurement surpasses petabytes. We truly live in the 'Data Age' or 'Data Revolution' (Kitchin, 2021). Since a single server can currently handle a data volume of up to around one Petabyte, any dataset larger than that needs to be stored on a distributed system. There are other reasons why a dataset would need to be stored on a distributed system before reaching a critical volume size.

Volume vs velocity in units
When measuring data volume, GB or Gb is used as a unit. Velocity is measured in GBps or Gbps. Be careful not to mix up the different units.

When dealing with a database, the following units are commonly used to describe the volume:

Table 1: Volume Units

Unit	Abbreviation	Storage space
Byte	B	8 bits
Kilobyte	KB	1024 bits
Megabyte	MB	1024 KB

Unit	Abbreviation	Storage space
Gigabyte	GB	1024 MB
Terabyte	TB	1024 GB
Petabyte	PB	1024 TB
Exabyte	EB	1024 PB
Zettabyte	ZB	1024 EB
Yottabyte	YB	1024 ZB

Source: Robert Horrion, 2022

Data volume can both be described in byte and bit. As shown in Table 1: volume units (Robert Horrion, 2022), one byte contains 8 bits. Note the following expression for the giga- variant of bits and bytes:

- GB = Gigabyte
- Gb = Gigabit

When comparing the `giga`-variant of bits and bytes, the differences in storage size for the two units are easy to see:

$$1 \quad \text{Gigabit} = 125 \quad \text{Megabytes}$$

Velocity

Velocity describes how fast data can be stored and accessed in a database. Modern big data systems are designed to handle high volumes of data quickly. There are a number of different factors that need to be considered to describe velocity:

- Transfer speed
- Response time
- Batch Processing (concept)
- Messages (concept)
- Data stream (concept)
- Eventual consistency (concept)

Transfer speed

Transfer speed provides a measure of how much data is transferred for each time unit. Seconds are most commonly used as time units. The resulting unit is bits per second, details can be found in the table below.

Table 2: Transfer Speed Units

Unit	Abbreviation
Bits per second	b/s or Bps
Kilobit per second	Kb/s or Kbps
Megabit per second	Mb or Mbps
Gigabit per second	Gb/s or Gbps

Source: Robert Horrion, 2022

When enough data is transferred to or from a server, the slowest component of the server, usually the disk that the data is stored on, will reach its transfer speed limit. When this happens, the workload can be accelerated by implementing a distributed system. In this case, data is written to many servers across a server pool. Therefore, the load is split, which results in an elevated data throughput capacity. Hence, distributed systems are a necessity in high throughput environments.

As with volume, **transfer speeds** can be measured in Gigabyte per second (GBps) or Gigabits per second (Gbps). The following equation is valid when transferring data:

Transfer speed
Volume multiplied by velocity results in the time needed to store or access data.

$$1 \text{ Gigabit per second} = 125 \text{ Megabyte per second}$$

Response time

In addition to transfer rate, response time represents an important factor in velocity. Response time describes the time it takes for a database to respond to an access or storage request. This is typically measured in milliseconds (ms). System load, hardware used, the number of nodes in a distributed system, etc. are all factors affecting response time.

Batch Processing

Batch processing represents the concept of collecting a certain amount of data, then processing it in batches. Batches can be of a predefined or fixed size. Batch processing can improve performance by processing many smaller data sets all at once, but it also bears the potential of overloading even a large distributed system, if many simultaneous batch processing requests lead to a peak in database access. This is usually data of low velocity since systems arbitrarily wait to process data until a batch is present, e.g. over-night processing, or once a week etc. (Kleppmann, 2017).

Messages

Messages are a useful means of transmitting data in Internet of Things (IoT) applications before the data is ingested into database systems. Messaging systems come with a queue to offer support for scenarios where an internet connection cannot be assumed to be stable. If an IoT device sending data to a backend through a messaging system goes offline, newly accumulated data is added to the queue, where it is stored until the device comes back online. The queue is then sent to the backend based on first-in, first-out (FIFO) methodology. This may result in a large number of messages reaching the backend at the same time, in case of the resolution of widespread internet outages. This can be data of low velocity if the message remains in the queue for a while due to connection issues. In other scenarios where a stable connection is provided, this can be data of high velocity (Kleppmann, 2017).

Data streaming

Data streaming is used when a constant stream of data is delivered by an application or a device. A data stream can be thought of as data that is delivered through a pipe or a conveyor belt and that needs to be processed and stored accordingly. This kind of data is therefore usually processed in real time. Examples include monitoring systems, such as industrial applications and health systems, where biodata is streamed and processed in real time. Since this is a real time scenario, this data can be classified as high velocity (Kleppmann, 2017).

Eventual consistency

For large, distributed database systems, eventual consistency provides an important concept of improving database read and write performance. It also ensures better availability of distributed systems. When eventual consistency is applied, the data is initially written to just one node, then replicated across others. Due to this concept, data can’t be assumed to be present or up to date in a distributed database system where eventual consistency is applied. The opposite of eventual consistency is known as strong consistency. Better read and write performance in systems with eventual consistency stems from the fact that when strong consistency is applied, the data needs to be written or read from all nodes at the same time to ensure the data is present on all nodes. Better availability in systems with eventual consistency stems from the fact that when strong consistency is applied, data can’t replicate to every node when a technical failure occurs (Kleppmann, 2017). The tradeoff between eventual and strong consistency needs to be made for each application of distributed database systems and depends on individual factors.

Variety

Data variety describes the different types of data present in big data. Structured, semi-structured and unstructured data all make up the field of Big Data. Structured data conforms to a schema (e.g., defined columns and datatype) and can be stored in a relational database. Examples include just a snipped of text or a single numeric value. A large set of characters or numeric values can also be structured data. Semi-structured data is data that doesn’t conform to a schema but contains some structure. HTML is an example of semi-structured data; it contains structure, but the structure doesn’t conform to a schema. Each HTML page contains different tags. Unstructured data doesn’t contain any structure and is therefore stored as a binary file. Examples include images, videos, and audio (John & Misra, 2017).

Examples of different types of data that can all be categorized into structured, semi-structured and unstructured data include:

- Single numeric values
- Large arrays of numeric values
- Data streams
- Messages (IoT)
- Text
- Metadata
- Photos
- Videos
- Audio
- 3D models
- Documents (HTML, JSON, XML, etc.)
- etc.

Some of these examples are stored as raw data inside of database systems, others are of a semi-structured or unstructured nature and have a certain file type, thus can be saved in a file system as an individual file. Each individual file contains a filename extension that points out how to handle the file contexts by a computer system. Some examples include .html files for webpages, .mp4 files for video and .wav files for audio.

Veracity

Veracity is a more recent addition to the 3Vs. The term describes the quality and accuracy of data. It can also be described as the certainty of data. (Kitchin & McArdle, 2016) Data may be inconsistent if a failing sensor is providing data with missing sections. Without any checks, this might go unnoticed and can have severe consequences. In other scenarios, such as when dealing with parsed data originating from a social network, data might be biased or even misleading through the spread of false information, also referred to as *fake news*.

Veracity provides a characteristic to classify data into being reliable or not. Not all cases of unreliable or incomplete data can be fixed after the data has been collected.

A list of potential veracity issues includes data that is…

- Inconsistent
- Untrusted
- Raw/uncleansed
- Biased
- Incomplete
- etc.

Data of good veracity is required as a decision-making basis and is therefore very important to the data science field.

In addition to this concept of Big Data being composed of the 4 Vs, there are numerous other definitions, including the 7 Vs or 10 Vs, where also variability, exhaustivity, fine-grained and relationality are considered aspects amongst others (Kitchin & McArdle, 2016)

1.2 Data Sources

Data comes in many forms as described in the previous section. Each form of data can have one or multiple sources. It is either generated by humans or by machines and can therefore come in different formats. The extract, transform, load (ETL) process was introduced in the 1970s and represents the process of loading data into a database system. Therefore, the ETL process describes how data is stored in a database and is the first step when dealing with data that is to be processed by a Big Data system (Zhang, Porwal, & Eaton, 2020). During the extract step, data is retrieved from the source. The transform step represents the act of cleansing data to establish consistency. During the load step, data is written to the target database system.

This section will cover the different sources of data and will guide you through an example of how data can be **mined**. A selection of primary Big Data sources includessocial media data, machine data, transactional data, CT Imaging data, and geospatial data in Geographic Information Systems (GIS) (Luntovskyy & Globa, 2019). These primary Big Data sources will be further discussed in this section.

Data Mining
Data Mining is defined as the process of finding, extracting, and processing data. The process is vital when dealing with Big Data sources since it provides insights into the patterns present in the dat**ae**, and hence provides the motivation to store large amounts of data.

Social Media Data

The internet was founded on the principle of communication. Early applications required users to be skilled in certain technologies, such as writing HTML pages. The advent of social media allowed users to share data with other users, often termed “friends” or “connections” by the commercial platforms. Shareable data commonly includes multimedia files, such as photos and videos, as well as text data and often polls. In addition to just storing and displaying the data in question, these platforms also often enable users to restrict access to posts to smaller groups of users, such as their friends. Data present on these platforms is usually generated and submitted for storage by humans, although bots have provided a growing source of data on these platforms in recent years. You will be introduced in-depth to the concepts of structured, semi-structured and unstructured data in section 1.3, but note that social media data can be structured or unstructured. Since platforms follow a data schema when creating new posts, and the addition of image and video data is possible, this data usually conforms to a schema.

In the context of social media, data velocity is not only important since many users may try to access the same piece of information at once, but also the business model of these platforms since online advertisement is time critical. For example, a news report with an attached video of a *breaking news* situation may be accessed by many users as the situation unfolds. This is achieved through the application of distributed systems.

Currently, the following platforms are amongst the most popular ones:

- Twitter
- Facebook
- Instagram (part of the Facebook company)
- Snapchat
- TikTok ("Most popular social networks worldwide as of January 2022, ranked by number of monthly active users," 2022)

Twitter provides a popular and valuable data source, since their network is used by many, almost all of the posts – so called tweets – are publicly available and can be searched through the powerful hashtag system. In addition, they provide a developer API that can be polled for information with an API access key.

A note about these social media platforms: The platforms have been used many times for unethical purposes. When using data from these platforms, we should keep in mind to distrust the data as well as to reflect upon the context in which the data was generated since the primary purpose of these platforms is advertisement.

`Tweepy` is a Python library that enables easy Twitter API access. `Psycopg` is a Python library that provides PostgreSQL API access. The two libraries will be used here to retrieve ten tweets with a #BigData hashtag created since January 1st 2020 to then store them in a PostgreSQL database. In a real word scenario, this process could be used to create an archive of tweets, which could then be processed for further information extraction later on.

First, a database connection to the PostgreSQL database must be established. You should install PostgreSQL before or use an online deployment of the database. To connect to the database the following code can be used (please fill in the parameters for your database).

```
connection = psycopg2.connect("dbname='database_name'  \
user='username' host='hostname', \
 password='password'")
```

Next, a `cursor` object needs to be created.

```
cursor = connection.cursor()
```

Now, we create a query to insert data into PostgreSQL.

```
insert_query = """ INSERT INTO bigdata
 (ID, TEXT) VALUES (%s,%s)"""
```

The tweets with a relevant hashtag are retrieved from the Twitter API. First, the library needs to authenticate against Twitter's OAuth service to gain access permission. This is done using the following three lines of code. You can find your user credentials in the Twitter developer's dashboard after creating an account there.

```
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)
api = tweepy.API(auth)
```

The resulting api object can now be used to authenticate any search requests against the API. The message text for each retrieved tweet is queued for storage to the PostgreSQL database upon retrieval.:

```
# iterate over each Tweet with the specified hashtag
# and date
for tweet in tweepy.Cursor(api.search, q="#BigData", \
    count=10, lang="en", since "2020-01-01").items():

    # create a new record with two elements: the tweet
# ID and the Tweet text with UTF-8 encoding
    new_record = (tweet.id, tweet.text.encode("utf-8"))

# insert the new record into the PostgreSQL DB
cursor.execute(insert_query, new_record)
```

After all of the tweets have been loaded and written using the cursor, the new data is committed to the database by executing the following command.

```
connection.commit()
```

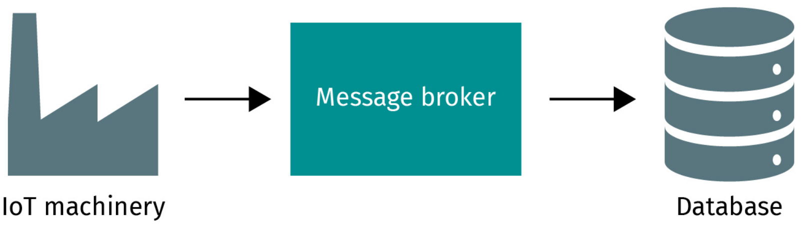
Machine Data

In connected industry applications, also referred to as *Industry 4.0*, lots of data is generated. This is often real time machine monitoring data reported by sensors represented as numerical values conforming to a schema. The connected industry setting also includes image data that is often captured on conveyor belts when checking products for quality. Therefore, this application includes both structured and unstructured data.

Modern, internet-connected industry machines are categorized as Internet of Things (IoT) devices. As IoT edge devices, they implement a messaging protocol such as MQTT to submit the captured data to a Message Broker, that is sometimes referred to as an IoT Hub. Popular options for Message Brokers include RabbitMQ and HiveMQ. The broker then makes the data available to clients, referred to as *subscribers*. The IoT edge devices are also termed *publishers*. Therefore, this methodology is referred to as the pub/sub pattern. The publisher and subscriber are never in contact with each other directly and therefore do not need to know of each other. They also do not need to be online at the same time since the Message Broker handles delivery of messages after periods of downtime on the subscriber's end (Team, 2015). In a Big Data scenario where machinery data is stored in a database, the subscriber would be a database connector, feeding delivered data into the database.

Refer to the following architecture diagram to understand how the Industry 4.0 with a Messaging Broker works:

Figure 2: The Pub/Sub pattern



Source: Robert Horrion, 2022

Transactional Data

The global financial markets produce a very large amount of mostly numeric data every day. This includes stock market prices at various exchanges around the globe, as well as transaction systems such as **SWIFT**. With markets being closed for hours each day, there are peaks and valleys in system load. A more recent appearance have been crypto currencies that rely on blockchain technologies where each transaction is stored as part of the blockchain (Luntovskyy & Globa, 2019). Due to the fact that generated numerical data usually conforms very well to schemas, transactional data can be classified as structured data. There are many open sources for us to use transactional data in our analyses. For example, in Python, we can use the `yfinance` package that allows for straightforward access to stock prices as time series.

SWIFT
The abbreviation SWIFT stands for Society of Worldwide Interbvank Financial Telecommunications S.S.. It is a payment processing system acting as a messaging service between banks to support inter-bank transactions. Therefore, the payee (sending party) does not have to use the same bank as the receiving end. SWIFT also provides SWIFT codes as routing information in transactions.

CT Imaging Data in Health Care

CT imaging led to a revolution in health care treatment in the early 1980s. It consists of the capture of many X-Ray slices of a patient's body in rapid succession to generate a 3D model of the body part that is to be imaged (Pai-Dhungat, 2020). Each X-Ray slice contains 2D image data and is saved as a file, therefore this is unstructured data.

Raw data of a CT scan results in about 22MB per uncompressed image file. Assuming 1000 slices are captured, this results in about 22GB of data. For high resolution scans, the volume may increase to up to 350GB for a single CT scan ("Department of Earth and Environmental Sciences - Frequently Asked Questions,").

Considering that many CT scans may be performed in a single hospital or lab in a single day, this quickly adds up to many terabytes or even petabytes of data. The calculation of 3D models based on these images requires fast access to the generated files, so Big Data systems play an important role. The same aspects hold for air-borne or satellite imagery that has basically the same traits as CT imagery, but in addition usually uses time slices and geospatial references.

Geographic Information Systems data

Geospatial data in Geographic Information Systems (GIS) presents unique challenges. For example, to determine the closest point in space to a given geospatial object in a database, GIS specific database extensions are usually needed. PostGIS is available as a PostgreSQL extension to cover such scenarios for PostgreSQL databases ("PostGIS - Spatial and Geographic objects for PostgreSQL,"). Data can either be structured, semi-structured or unstructured. An example for structured geospatial data is a simple numerical value for longitude or latitude. Semi-structured data could be present in the form of CSV files containing multiple geo-points and associated attributes, usually stored in so-called attribute tables. Unstructured data is typically generated in the form of satellite images.

There are many open sources for geospatial data for us to us in our analyses. In the European Union, for instance, the Infrastructure for Spatial Information in Europe (INSPIRE) is a directive that ensures that official geospatial data has to be public and open for everybody to use (INSPIRE, 2022). The data comes in geospatial data formats, such as Shapefiles, GeoPackages, or GeoTIFFs, or directly as data services in the form of data APIs, such as Web Feature Services (WFS) or Web Map Services (WMS). Also, as an overview of many open data sources for us to use (not only geospatial data), the curated GitHub repository, Our World in Data, is a good starting point (Our World in Data, n. d.).

1.3 Data Types

Big Data can be divided into three main data types: structured, semi-structured and unstructured data. Structured data can be mapped to a schema and can be further given datatypes to store it more efficiently and prepare it for machine processing. Unstructured data can’t be mapped to a scheme, instead it is represented by a single document or file. Semi-structured data doesn’t conform to a predefined schema but contains some structure.

When it comes to storing data, the concepts of flat and hierarchical data are very important, too. The flat model is also called the table model and represents the most basic way of storing data in a two-dimensional way, for example in a spreadsheet. When data is stored in a hierarchical model, it can be presented in a tree-like structure, meaning there are relationships between records.

As earlier discussed, data can be of high or low velocity. When data is streamed, a constant flow of data is provided. Therefore, the data is of high velocity. This may be the case in automotive applications when working with sensors, but the weather station scenario provided in the case study at the beginning of this unit provides a good example for streaming data, as well. In contrast, batch processed data is classified as being of low velocity. The data is accumulated by a system, then a large amount of data is processed at once. An example for batch processed data could be a grocery store supply chain system that checks the current inventory every day after the store closes. It then batch processes the current inventory in an ordering system to order more stock, if the store is running low on certain products.

Also earlier presented, data that originates from geospatial sources is called geospatial data and needs to be stored in databases with GIS extensions. This data comes with its own unique set of challenges, such as determining the closest point of coordinates stored in a database in relation to a provided coordinate. When spatial data is present in a database, spatial joins can be performed to determine the relationship between datasets. For example, when trying to determine whether a tracked cell phone is within the reception area of a given cell tower, the latitude and longitude of the cell phone's location are joined with the polygon that represents the cell tower's coverage area. This will determine whether the phone lies within the coverage area. Since joins are used, this can be performed on relational databases.

In the following, we look at each of these types in more detail.

Structured Data

Structured Data is typically stored in a Relational Database Management System (RDBMS), but can also be stored in NoSQL databases (John & Misra, 2017). Relational databases are made up of tables containing a predefined schema, meaning the structure of the table must be defined before any data is added to the table. This definition includes data types and a variable name. Data needs to be **normalized** before being entered into the database. This process is also called data cleansing to turn raw data into cleansed data. Both commercial and free solutions exist, with some being open source, others proprietary. Data in an RDBMS is typically accessed and manipulated using the query language SQL. Different dialects of SQL exist, these are custom implementations for different RDBMS. Popular solutions include PostgreSQL, MySQL and Microsoft SQL Server amongst many others available on the market.

Data Normalization
Data normalization is the process of conforming data to a predefined standard. Normalized data appears similar to other records normalized using the same standard and is stored in a format that can be processed by machines.

Examples for structured data include:

- Numeric sensor data
- Birthdates in a customer database
- Addresses
- Names
- E-Mail Addresses
- Spatial coordinates
- Phone numbers
- etc.

In order to understand how structured data can be efficiently stored in RDBMS, it is important to understand how data is encoded to be stored in computer systems in general. Strings are usually encoded in UTF-8 or ASCII, with the former offering full support for Unicode characters. These characters are encoded into sequences of 8-bit bytes (Kilbourne & Williams, 2003).

To understand how this works, let us see how integer values are encoded in binary to store them in a computer system, including databases. When encoding a decimal integer in binary, each position in the bitstring holds a defined value:

Table 3: Binary Encoding Values for an 8-bit Integer

Index	Value
1	128
2	64
3	32
4	16
5	8
6	4
7	2
8	1

Source: Robert Horrion, 2022

For example, the decimal value 11 is encoded as an 8-bit integer as follows:

11=00001011

This encoding schema results in a maximum supported value of 255 for 8-bit integers.

Floating point values are by definition susceptible to floating point error when processed by a machine. Since computer systems are only able to store and process a finite length of bits that represent a fractional numeric value, some values may need to be rounded. When rounding occurs, the numeric value is necessarily modified. Performing arithmetic on such rounded numbers can then escalate the rounding error into a considerable and noticeable error (Goldberg, 1991). This is a potential problem when dealing with mission critical systems, for example in the financial industry or early alert systems for hazards.

Programming languages commonly assign data types to variables. For some languages such as Python, this happens automatically. Starting with version 3.5, Python offers type hints when defining variables to point out the data type to a programmer ("typing — Support for type hints," 2022). Data types can be added to Python variables as follows:

```
temperature: float = 26.8
```

Other programming languages, such as Java, require data types to be set when a new variable is defined.

Each RDBMS comes with its own set of available data types. For this course, PostgreSQL is picked as a popular RDBMS and looked at on a more detailed level. You will find other systems in the field; it is important to familiarize yourself with each system and its available data types before you start working with the system in a development or production scenario.

PostgreSQL stores strings of different lengths with variable storage consumption, depending on the string in a `text` data type.

`Integer` values are stored in a 4-byte length format. Multiple formats to store floating point numbers are available with `double` taking up 8 bytes to store a single value. The `decimal` data type offers variable lengths allowing us, for example, to minimize the risk of floating point error if we chose an appropriate length.

`Bool` values in theory take up one bit of storage, in the PostgreSQL implementation one byte is consumed to store a single value. Bool values can only contain true or false information. Under the hood, this is stored as zeros (false) and ones (true).

`Date` values are stored in YYYY-MM-DD format, ISO 8601 syntax compliant and consume 4 bytes for each date stored. To prevent the floating-point error from becoming a problem in critical applications such as ones of financial nature, PostgreSQL offers a money data type that ensures a correct storage of monetary values.

PostgreSQL offers special data types for some specific scenarios, as well.

The following table provides an overview over a selection of important and frequently used data types available in PostgreSQL ("Chapter 8. Data Types**,**", n.d.).

Table 4: A Selection of Important Data Types Based on the PostgreSQL Documentation

Data Type	Stored Data	Example	Storage Size
integer	An integer	1500	4 bytes
double precision	Floating-point numeric value with up to 15 decimal points precision (small floating-point values)	1.456	8 bytes
decimal	Floating-point numeric value with user specified precision (large floating-point values)	5.74656637373736465675 88271629725465786	variable
char	A single character	b	1 byte
bool	True or false	true	1 byte
text	Unlimited length string	“Hello, how are you?”	Up to 1GB
date	Date in YYYY-MM-DD format (ISO 8601 syntax)	2006-06-09	4 bytes

Source: Robert Horrion, 2022 based on “Chapter 8. Data Types”

Special data types
PostgreSQL offers some very specific data types to store information such as geodata (coordinates), IP Addresses, UUIDs, CIDR notation for IPv4 and IPv6 network specification and MAC addresses.

Data type classification and normalization is not only important to efficiently store and find values in a database, but also for many automated processing tasks such as data science applications, that train on or evaluate large numeric datasets.

Semi-structured Data

Semi-structured data represents data that doesn't conform to a data schema but contains some structure (John & Misra, 2017). Classic examples include JSON, where each JSON document can have a different structure when compared with other JSON documents. The same is true for HTML files and other examples listed below (note that each of these examples can be structured or semi-structured depending on how the data are stored in the respective files):

- HTML files
- CSV files
- E-Mails
- Zip files
- Extensible Markup Language (XML) files
- JavaScript Object Notation (JSON) files
- Files written in markdown language
- etc.

Storing scraped HTML data in databases comes with its own set of challenges. Such data can be stored in a NoSQL database, as well as a SQL database. However, SQL databases are prone to SQL injection attacks, where a hacker is able to manipulate, access or delete data in a database that they shouldn't have access to by attempting to store a SQL request. Due to the fact that such requests can be embedded in HTML data, storing scraped HTML data in a SQL database can expose the system to SQL injection attacks ("SQL Injection", n.d.). Security measures need to be implemented to prevent such attacks. Optionally, a NoSQL database can be used to store HTML data as semi-structured data.

Unstructured Data

Data that is neither structured nor semi-structured is called unstructured data. This type of data does not conform to a schema and contains no structure that is recognizable across files (John & Misra, 2017).

Examples of unstructured data include:

- Image data
- Video
- Audio
- Binaries
- etc.

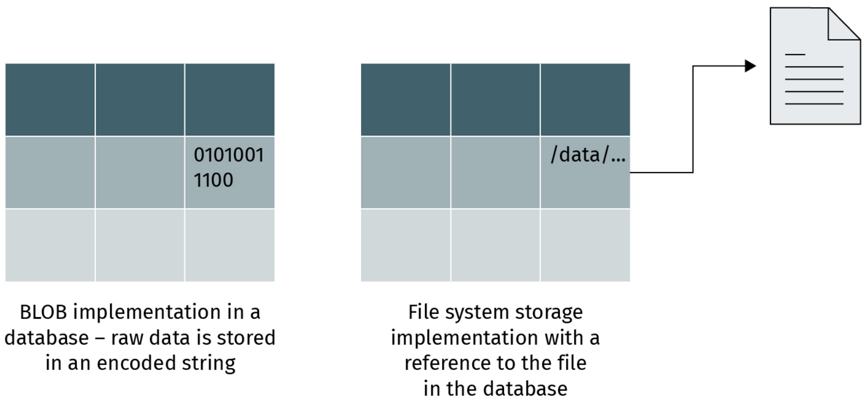
In these cases, relational databases will not do the job. This is one of the reasons why so-called NoSQL databases were developed. For example, the NoSQL databases

MongoDB or Cassandra are popular database choices when it comes to storing semi-structured and unstructured data. MongoDB represents a document-oriented DB, in which each entry in the database is stored as a document, such as a JSON file ("How is Unstructured Data Used in a Database?," 2021). Documents are grouped in collections. Collections can be searched for individual attributes in the documents they hold. MongoDB, for instance, specifically adopts BSON, a form of JSON, to structure the stored documents ("JSON and BSON," 2021). It is beyond the scope of this section to go into details about NoSQL, but his should give you an idea of what these exiting technologies can be used for.

Concepts of storing semi-structured or unstructured data in a database

Unstructured data can be stored as a Binary Large Object (BLOB) or as a file in a file system. If the file is stored in a file system, a reference to the file and its path needs to be created and updated every time changes to the file or its location are made. BLOBs can be stored in relational databases like PostgreSQL and in NoSQL databases such as MongoDB. The concept is depicted in Figure 1: Storing binary data in a database. BLOBs are defined by the SQL standard and represent a way to store binary files in a RDBMS. However, PostgreSQL doesn't implement BLOB. Instead, it is implemented as a similar file binary storage solution. MongoDB offers GridFS as a specification to store files larger than 16MB ("GridFS," 2021).

Figure 3: Storing binary data in a database


0101001
1100
BLOB implementation in a database – raw data is stored in an encoded string
/data/...
File system storage implementation with a reference to the file in the database

Source: Robert Horrion, 2022

The figure shows how a reference to a file in a file system needs to be created and updated when the storage location changes. The left hand side shows how data is stored as a BLOB – without an external reference. The reference as shown in the file system storage implementation provides a breeding ground for problems since it can be difficult to keep up with location changes in the file system. Therefore, a benefit to using BLOBs to store files in a database is that only one datapoint – the actual database record – needs to be updated when the file is changed. This keeps the storage solution simpler and is less prone to errors. GridFS offers support for an unlimited number of files. In addition, it is also able to speed up access to a file by only reading part of it.

Access and manipulate data in a MongoDB using Python

Some scenarios require a large number of emails to be stored in a database for quick access. This might be a database for a Data Science experiment that uses emails as a training set (for instance to learn the distinction between spam mails and legitimate ones). In a later step of the project, the Data Science team will also use the E-Mail attachments, so, they are to be retained. As such, the E-Mail messages are to be classified as semi-structured data. Therefore, MongoDB represents a useful storage solution. The following Python code connects to and adds a single E-Mail to a MongoDB in the form of a document. This process can be turned into a script and repeated indefinitely to accumulate a large dataset of E-Mails in the collection. The Python library pymongo is used to access and manipulate data in a MongoDB (Walters, 2017).

First, the MongoClient instance is created, which tells the library where the MongoDB installation can be reached.

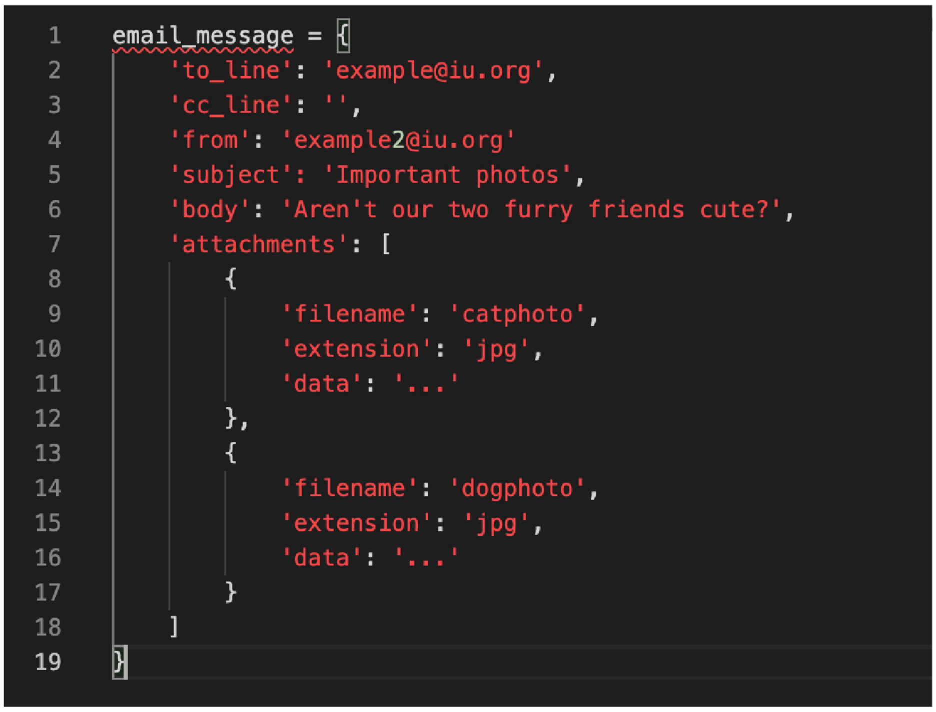
```
from pymongo import MongoClient
client = MongoClient('<<MongoDB URL>>’)
```

Then a database object has to be created to reference the database.

```
database = client.emails
```

Next, we create the document. This is achieved by creating a JSON document in code. Note that the attachments are inserted as subdocuments.

Figure 4: JSON representation of an email message to be stored in MongoDB



```
email_message = {
    'to_line': 'example@iu.org',
    'cc_line': '',
    'from': 'example2@iu.org'
    'subject': 'Important photos',
    'body': 'Aren't our two furry friends cute?',
    'attachments': [
        {
            'filename': 'catphoto',
            'extension': 'jpg',
            'data': '...'
        },
        {
            'filename': 'dogphoto',
            'extension': 'jpg',
            'data': '...'
        }
    ]
}
```

Source: Robert Horrion, 2022

The created email_message document can then be saved to the database.

```
saved = database.emaildata.insert_one(email_message)
```

Locating all records where a document attribute matches a searched term is now straightforward. The following line of code will return all emails sent to the address "`example@iu.org`".

```
example_author = database.emaildata.find({'to_line': 'example@iu.org'})
```

Just as locating all documents where a given parameter matches is easy, it is trivial to delete such records, as well. The following line of code will delete all E-Mails sent to "`example@iu.org`" from the database.

```
deleted = database.emaildata.delete_many({'to_line': 'example@iu.org'})
```

SUMMARY

Big Data is defined by the creation, storage, and computation of huge amounts of data. This data frequently needs to be stored in distributed systems that are capable of handling a very large number of parallel read and write requests. Data is stored either in Relational Database Management Systems (RDBMS) or NoSQL databases. Document-based databases represent one of the available NoSQL databases and are more closely described in this unit.

The 4Vs represent the characteristics of big data: volume, velocity, variety, and veracity. Volume describes *how much* data is present. The unit most commonly used to measure volume is bytes, kilobytes, megabytes and so on. Velocity describes the speed at which data is stored in or read from a database system. This includes transfer speed, which is most commonly measured in bits per second (b/s), kilobits per second (kb/s), megabits per second (mb/s) and so on. Another important descriptor for velocity is response time, which specifies the time until a response is provided upon querying the database. Batch processing, messages, data streaming, and eventual consistency all represent important concepts in relation to velocity. Variety characterizes the different types of data present in Big Data systems. Data may be structured, semi-structured or unstructured. Structured data conforms to a schema and can be saved without a file format. This kind of data typically includes numerical values and raw text. Semi-structured data contains some structure; unstructured data does not contain any structure and can be saved in a file format as an individual file. Examples for unstructured data include photos, videos, and audio. Veracity describes the quality and accuracy of data and may also be described as the certainty of data. It provides a measure for how inconsistent, untrusted, raw/uncleansed, biased, and incomplete data may be.

Data is either generated by humans or by machines and can be mined. Important sources of Big Data are social networks, with Twitter being an excellent data source due to its very open and public nature. Other important sources include machine data, transactional data, CT imaging data in Health Care, and Global Information Systems (GIS). Some data sources, such as CT imaging are limited to unstructured data by nature, others can include multiple data types, and yet others require database extensions, such as GIS data.

PostgreSQL is a popular RDBMS for storing structured data. Some kinds of data come with a special set of challenges and require their own data types. PostgreSQL offers special data types for values of financial nature, IP Addresses, MAC Adresses, UUIDs and CIDR notification. The document-based database MongoDB can handle the storage of semi-structured and unstructured data very well. Unstructured data may also be saved as files in a file system. Data needs to be normalized when it is stored in a database; this involves conforming it to a standard.

UNIT 2

WORKING WITH COMMON DATA FORMATS

STUDY GOALS

On completion of this unit, you will be able to ...

- define different data formats.
- apply basic data operations with Python.
- distinguish between different binary data models.
- manage binary data formats using Python.

2. WORKING WITH COMMON DATA FORMATS

Case Study

Data can be stored in a variety of ways. Since the emergence of the computer, the number of different formats has been steadily increasing. Most of the time, these formats are written and structured so that they are easy for machines to understand. CSV, JSON, and XML are examples of machine readable formats. Over time, a few formats have been chosen for specific use cases. Comma-separated values (CSV) files are often used to share small datasets, which can easily be handled in a single file and are then used in different tools for further processing. Json and XML are most dominant in the context of the web and can be used in a high variety of different scenarios. Those file formats, together with their underlying encoding, are presented in the first section of this unit. Along with explicit Python examples, we will be able to handle those file formats.

With the ever-increasing amount of data, new technologies had to be developed to store data more efficiently. Therefore, the second section of this unit focuses on technologies that were developed in the context of the big data movement. Based on HDF5, Parquet, and Arrow, three modern data formats are presented along with tangible Python examples.

2.1 Text-Based Formats (CSV, XML, JSON)

In many real-world use cases, the data provided will be in a text-based format. Since computers store information in a binary manner, different representations have been developed. One of the oldest representation formats is the American Standard Code for Information Interchange (ASCII) format.

ASCII

In order for computers to process a variety of types of information, a way of storing this information is needed. Since computers can only store zeros and ones in the form of a bit, it was necessary to find a binary representation of characters in a text. To translate the 26 letters, the 10 numerical values, and some special characters into a bit notation, a combination of bits is required. The table below shows how bits can be translated into decimal values.

Table 5: Bits to Decimal Values

Bit	7	6	5	4	3	2	1	0	

Max number	$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$	
Example	0	1	0	0	1	1	0	1	SUM
	0	64	0	0	8	4	0	1	=77

A byte, which is a block of 8-bits, represents $2^8 = 256$ values. To translate those bit values into a human readable format, the ASCII standard is used. ASCII was developed to translate binary digits into characters and back. Original ASCII is a seven-bit code that is able to represent 128 different characters by translating decimals into binary values. Other standards used the additional eighth bit to represent even more characters and make a complete byte. The figure below shows how the different ASCII characters translate into binary and decimal notation. The binary values in the first column are a direct translation of the decimal values. The first 32 characters are non-printing characters, ranging from a backspace to an escape. The following 32 characters include collating characters, such as ., +, space, and :, as well as the 10 numeric values. The 26 alphabetic values are listed in combination with some special characters, such as @ or [. The last 26 characters are the alphabetic values in lower case with a few other special characters like { or | (Interface Age Staff, 1980).

Figure 5: The ASCII table

Decimal	Binary	Octal	Hex	ASCII
000	0000000	000	00	(NULL)
001	0000001	001	01	(START OF HEADING)
002	0000010	002	02	(START OF TEXT)
003	0000011	003	03	(END OF TEXT)
004	0000100	004	04	(END OF TRANSMISSION)
005	0000101	005	05	(ENQUIRY)
006	0000110	006	06	(ACKNOWLEDGE)
007	0000111	007	07	(BELL)
008	0001000	010	08	(BACKSPACE)
009	0001001	011	09	(HORIZONTAL TAB)
010	0001010	012	0A	(LINE FEED)
011	0001011	013	0B	(VERTICAL TAB)
012	0001100	014	0C	(FORM FEDD)
013	0001101	015	0D	(CARRIAGE RETURN)
014	0001110	016	0E	(SHIFT OUT)
015	0001111	017	0F	(SHIFT IN)
016	0010000	020	10	(DATA LINK ESCAPE)
017	0010001	021	11	(DEVICE CONTROL 1)
018	0010010	022	12	(DEVICE CONTROL 2)
019	0010011	023	13	(DEVICE CONTROL 3)
020	0010100	024	14	(DEVICE CONTROL 4)
021	0010101	025	15	(NEGATIVE ACKNOWLEDGE)
022	0010110	026	16	(SYNCHRONOUS IDLE)
023	0010111	027	17	(ENG OF TRANS. BLOCK)
024	0011000	030	18	(CANCEL)
025	0011001	031	19	(END OF MEDIUM)
026	0011010	032	1A	(SUBSTITUTE)
027	0011011	033	1B	(ESCAPE)
028	0011100	034	1C	(FILE SEPARATOR)
029	0011101	035	1D	(GROUP SEPARATOR)
030	0011110	036	1E	(RECORD SEPARATOR)
031	0011111	037	1F	(UNIT SEPARATOR)
032	0100000	040	20	(SPACE)
033	0100001	041	21	!
034	0100010	042	22	”
035	0100011	043	23	#
036	0100100	044	24	$
037	0100101	045	25	%
038	0100110	046	26	&
039	0100111	047	27	’
040	0101000	050	28	(
041	0101001	051	29	)
042	0101010	052	2A	*
043	0101011	053	2B	+
044	0101100	054	2C	,
045	0101101	055	2D	-
046	0101110	056	2E	.
047	0101111	057	2F	/

Source: Weiman (2010). CC BY 3.0

Figure 6: The ASCII table

Decimal	Binary	Octal	Hex	ASCII
048	0110000	060	30	0
049	0110001	061	31	1
050	0110010	062	32	2
051	0110011	063	33	3
052	0110100	064	34	4
053	0110101	065	35	5
054	0110110	066	36	6
055	0110111	067	37	7
056	0111000	070	38	8
057	0111001	071	39	9
058	0111010	072	3A	:
059	0111011	073	3B	;
060	0111100	074	3C	<
061	0111101	075	3D	=
062	0000001	076	3E	>
063	0111111	077	3F	?
064	1000000	100	40	@
065	1000001	101	41	A
066	1000010	102	42	B
067	1000011	103	43	C
068	1000100	104	44	D
069	1000101	105	45	E
070	1000110	106	46	F
071	1000111	107	47	G
072	1001000	110	48	H
073	1001001	111	49	I
074	1001010	112	4A	J
075	1001011	113	4B	K
076	1001100	114	4C	L
077	1001101	115	4D	M
078	1001110	116	4E	N
079	1001111	117	4F	O
080	1010000	120	50	P
081	1010001	121	51	Q
082	1010010	122	52	R
083	1010011	123	53	S
084	1010100	124	54	T
085	1010101	125	55	U
086	1010110	125	56	V
087	1010111	127	57	W
088	1011000	130	58	X
089	1011001	131	59	Y
090	1011010	132	5A	Z
091	1011011	133	5B	[
092	1011100	134	5C	\
093	1011101	135	5D	]
094	1011110	136	5E	^
095	1011111	137	5F	_

Source: Weiman (2010). CC BY 3.0

Figure 7: The ASCII table

Decimal	Binary	Octal	Hex	ASCII
096	1100000	140	60	'
097	1100001	141	61	a
098	1100010	142	62	b
099	1100011	143	63	c
100	1100100	144	64	d
101	1100101	145	65	e
102	1100110	146	66	f
103	1100111	147	67	g
104	1101000	150	68	h
105	1101001	151	69	i
106	1101010	152	6A	j
107	1101011	153	6B	k
108	1101100	154	6C	l
109	1101101	155	6D	m
110	1101110	156	6E	n
111	1101111	157	6F	o
112	1110000	160	70	p
113	1110001	161	71	q
114	1110010	162	72	r
115	1110011	163	73	s
116	1110100	164	74	t
117	1110101	165	75	u
118	1110110	166	76	v
119	1110111	167	77	w
120	1111000	170	78	x
121	1111001	171	79	y
122	1111010	172	7A	z
123	1111011	173	7B	{
124	1111100	174	7C	|
125	1111101	175	7D	}
126	1111110	176	7E	~
127	1111111	177	7F	(DELL)

Source: Weiman (2010). CC BY 3.0

ASCII offers a high variety of characters to the user, but was limited to the eight-bit encoding. Many efforts were made to define standards that can represent more characters. One of the most popular ones is the UTF-8 encoding, which can additionally represent Greek letters, Asian symbols, and mathematical signs. The Universal Coded Character Set Transformation Format (UTF), follows the ASCII notation for the first 128 characters. In this section, we will focus on text-based data, which can be seen as an encoded ASCII representation. When working with a text file, it is important to understanding what kind of encoding was used.

Source: Wietreck (2021).

CSV

One of the most common data formats is the CSV format. As the name indicates, the format contains values separated by a separator. The comma is used as a standard separator, but the semicolon is also widely used. The type of separator is arbitrary and is not important to the data format. Independent of the used separator, the file extension is the same (.csv). A file that is separated by a tab can also be saved with the file extension ".tsv". CSV files are just simple ASCII text files where a separator character is used to separate a certain set of characters.

To read or write CSV files, Python has some built-in methods contained in the CSV package. Apart from this, as CSV files are simple text files, we can also use the `open()` function to interact with these text files as shown in the example below. Any text file can be read using a "with" statement to open the text file. The `read()` method can be used to extract the content, which is then returned through a print statement.

```
with open('example.txt') as f:
    content = f.read()
    print(content)
```

While this is already useful and will allow users to manipulate CSV files, there is another way to work with such files. The pandas package, which is widely utilized in the data analytics community, is used in the following examples to demonstrate how to work with tabular data like CSV. Pandas is an abbreviation of panel data.

Before loading the data into Python, inspecting the CSV data is recommended. This can be done, for example, with an editor, and will show the following data for the Islands.csv file. The Islands.csv file can be created by opening the editor on the local computer, copying the following text and then saving the file as .csv (Countrymeters, 2021).

```
Island;Year;Residents;Capital;Continent
Cape Verde;2005;471000;Praia;Africa
Cape Verde;2010;509000;Praia;Africa
Cape Verde;2015;546000;Praia;Africa
Fiji;2005;837000;Suva;Oceania
Fiji;2010;876000;Suva;Oceania
Fiji;2015;910000;Suva;Oceania
Isle of Skye;2005;10300;Portree;Europe
Isle of Skye;2010;10500;Portree;Europe
Isle of Skye;2015;10800;Portree;Europe
```

A header with the names of the columns can be identified at the top of the dataset, and every single data row consists of values separated by a semicolon. The dataset has nine rows showing the data for three different islands for three different years.

In order to use this dataset in Python, we read the data with the pandas method `read_csv()` to load the data into a pandas DataFrame. In order to load the data into a DataFrame, we specify the location of the file and the separator of the csv file. The separator can be defined in a parameter called `sep`. As a result, we obtain a well-structured tabular representation of the CSV data, including column names and a row index.

```
import pandas as pd
data = pd.read_csv("Islands.csv", sep = ";")
```

Figure 8: Island DataFrame

	Island	Year	Residents	Capital	Continent
0	Cape Verde	2005	471,000	Praia	Africa
1	Cape Verde	2010	509,000	Praia	Africa
2	Cape Verde	2015	546,000	Praia	Africa
3	Fiji	2005	837,000	Suva	Oceania
4	Fiji	2010	876,000	Suva	Oceania
5	Fiji	2015	910,000	Suva	Oceania
6	Isle of Skye	2005	10,300	Portree	Europe
7	Isle of Skye	2010	10,500	Portree	Europe
8	Isle of Skye	2015	10,800	Portree	Europe

Source: Wietreck (2021).

The DataFrame takes the first row of the CSV file and defines it as the header. If the CSV file is in the correct structure, this can be helpful in addressing certain columns. Unfortunately, there are scenarios where this does not return the expected result. In the following scenario, there is no header present in the CSV file, so this has to be added manually. The example input CSV could be as follows:

```
Cape Verde;2005;471000;Praia;Africa
Cape Verde;2010;509000;Praia;Africa
Cape Verde;2015;546000;Praia;Africa
Fiji;2005;837000;Suva;Oceania
Fiji;2010;876000;Suva;Oceania
Fiji;2015;910000;Suva;Oceania
Isle of Skye;2005;10300;Portree;Europe
Isle of Skye;2010;10500;Portree;Europe
Isle of Skye;2015;10800;Portree;Europe
```

To bring it into the desired structure, it is necessary to hand over the names argument, together with the expected column names in the right order. This code returns the same results as in the example above.

```
column_names = ["Island", "Year", "Residents", \
    "Capital","Continent"]
data = pd.read_csv("Islands_noHeader.csv", \
    names = column_names, sep = ";")
```

Next, we see another common example where the `read_csv()` method has to be modified. This can be the case when there are some specifications or metadata above the actual data, for instance, an export header for this file. This is often the case when working with data that are extracted directly from a system. The header might include the date of the extraction, the specific source, etc. The following CSV file contains a header which indicates the extraction date, the extraction format, and the requester. Since there is only one value in those rows, the separator will separate empty values from each other to meet the overall number of expected separators in each row.

```
Extraction Date: 24.05.2016 12:02:21;;;;
Data Format: csv;;;;
Requester: IU;;;;

Island;Year;Residents;Capital;Continent
Cape Verde;2005;471000;Praia;Africa
Cape Verde;2010;509000;Praia;Africa
Cape Verde;2015;546000;Praia;Africa
Fiji;2005;837000;Suva;Oceania
Fiji;2010;876000;Suva;Oceania
Fiji;2015;910000;Suva;Oceania
Isle of Skye;2005;10300;Portree;Europe
Isle of Skye;2010;10500;Portree;Europe
Isle of Skye;2015;10800;Portree;Europe
```

To skip the first four rows, the `read_csv()` method takes a `skiprows` argument, which expects the integer of each row that should be skipped. In this case, the first four rows are skipped using the range function. It can be useful to use a function to identify the header row, for example, by name. This provides additional flexibility since automatically generated headers do not always have the same number of rows. The following line of code returns the same results as in the previous examples:

```
data = pd.read_csv("Islands_meta.csv", sep = ";", \
    skiprows = range(0,4))
```

Especially with outputs from old systems encoding errors can occur. These are sometimes difficult to identify and to solve. The following example uses the encoding parameter to make sure that the CSV is read as UTF-8.

```
data = pd.read_csv("Islands.csv", sep = ";", \
    skiprows = range(0,4), encoding = "utf-8")
```

Now, as the data are loaded into a pandas DataFrame, they can be filtered, sorted, grouped, and transformed. As we have seen, when working with CSV data, it is important to understand the structure of the file to be used. Header rows and export headers need to be identified and handled. When reading or writing data back to a CSV file, encoding must also considered. The following example writes the DataFrame into a CSV file called Islands_output.csv with a separator ;.

```
data.to_csv("Islands_output.csv", sep = ";")
```

JSON

The JavaScript Object Notation (JSON) is a data format most commonly used for transfers in the web (Müller & Guido, 2017). It became popular because of its clean, easy to read structure, and for the possibility of flexible application, especially for nested semi- and unstructured data. JSON data are similar to a Python dictionary, and consist of a combination of key value pairs. JSON files start with curly brackets, then the key and value are defined, separated by a colon. Entries are separated by a comma, which indicates the start of the next element. At the baseline, a JSON file is similar to a CSV file: an ASCII text file in a human readable representation. In the following JSON file, we can see the weather for London on 24.05.2021. The data are returned from the openweathermap API (Open Weather, n.d.), which is a publicly available API for global weather data.

```
{"coord": {"lon": -0.1257, "lat": 51.5085},
"weather": [{"id": 500,
"main": "Rain",
"description": "light rain",
"icon": "10d"}],
"base": "stations",
"main": {"temp": 282.22,
"feels_like": 279.41,
"temp_min": 280.64,
"temp_max": 283.86,
"pressure": 1005,
"humidity": 80},
"visibility": 10000,
"wind": {"speed": 5.43,
"deg": 305,
"gust": 11.39},
"rain": {"1h": 0.12},
"clouds": {"all": 90},
"dt": 1621884916,
"sys": {"type": 2,
"id": 268730,
"country": "GB",
"sunrise": 1621828607,
"sunset": 1621886294},
"timezone": 3600,
"id": 2643743,
"name": "London",
"cod": 200}
```

When reading this file in Python, the JSON package is used. After reading the file using Python's built-in `open()` and `read()` functions, the JSON package is used to load the data. While the return of `json_data` is a string, the data object itself will be a Python dictionary.

```
import json
json_data = open('LondonWeather.json').read()
data = json.loads(json_data)
```

As we can see in the code above, JSON files can be nested. In order to access elements of a specific level in this nested structure, we can use the top level key of this dictionary for our query.

```
data["main"]

# console output:
{'temp': 282.22,
'feels_like': 279.41,
'temp_min': 280.64,
'temp_max': 283.86,
'pressure': 1005,
'humidity': 80}
```

Such a structure can then be easily put into a DataFrame as this is no longer nested, but in flat format. Even though pandas has its own method to read JSON files, it can be difficult to put an unstructured format like JSON into a column-based format. When handing over the “main” section of our JSON example to a DataFrame, the keys will be used as columns. In order to give more context to the weather data, the name of the city is passed as the row index.

```
weather_data = pd.DataFrame(data["main"], \
    index = [data["name"]])
```

Alternatively, we can also normalize the json directly to a pandas DataFrame using the `json_normalize()` function.

```
pd.io.json.json_normalize(data['main'])
```

In this example, the data from a JSON file were used. When accessing the API directly, the data for each city can be retrieved and the value can be stored in a data structure like a DataFrame. Together with additional information like the time of the measurement, a script can be built that periodically extracts weather information from the API, brings it into the desired order, and stores it in a DataFrame or database.

XML

Hierarchical data from the web are often represented using the extensible markup language (XML). XML data are made readable for machines and humans. The following example XML file, which can be downloaded as an XML file from the XML module documentation (Python Software Foundation, n.d.), shows information about different countries. When opening the data in an editor, the following structure will be shown.

Figure 9: XML Example Countries

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein"
             <rank>1</rank>
             <year>2008</year>
             <gdppc>141100</gdppc>
             <neighbor name="Austria" direction="E"/>
             <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
             <rank>4</rank>
             <year>2011</year>
             <gdppc>59900</gdppc>
             <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
             <rank>68</rank>
             <year>2011</year>
             <gdppc>13600</gdppc>
             <neighbor name="Costa Rica" direction="W"/>
             <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

Source: Wietreck (2021).

The XML ElementTree class will first be imported before the XML file is parsed. The `getroot()` method is used to get the root of the hierarchical data structure. In this case, this is the tag <data>. A for loop is used to display the tags and attributes of each child object using the attribute notation. A specific value can be accessed using the array notation. The following script will return the name of every single country in the dataset and the value of the of second element in the first child.

```
import xml.etree.ElementTree as ET
tree = ET.parse('data.xml')
root = tree.getroot()
for child in root:
    print(child.tag, child.attrib)
print(root[0][1].text)

# console output:
# country {'name': 'Liechtenstein'}
# country {'name': 'Singapore'}
# country {'name': 'Panama'}
```

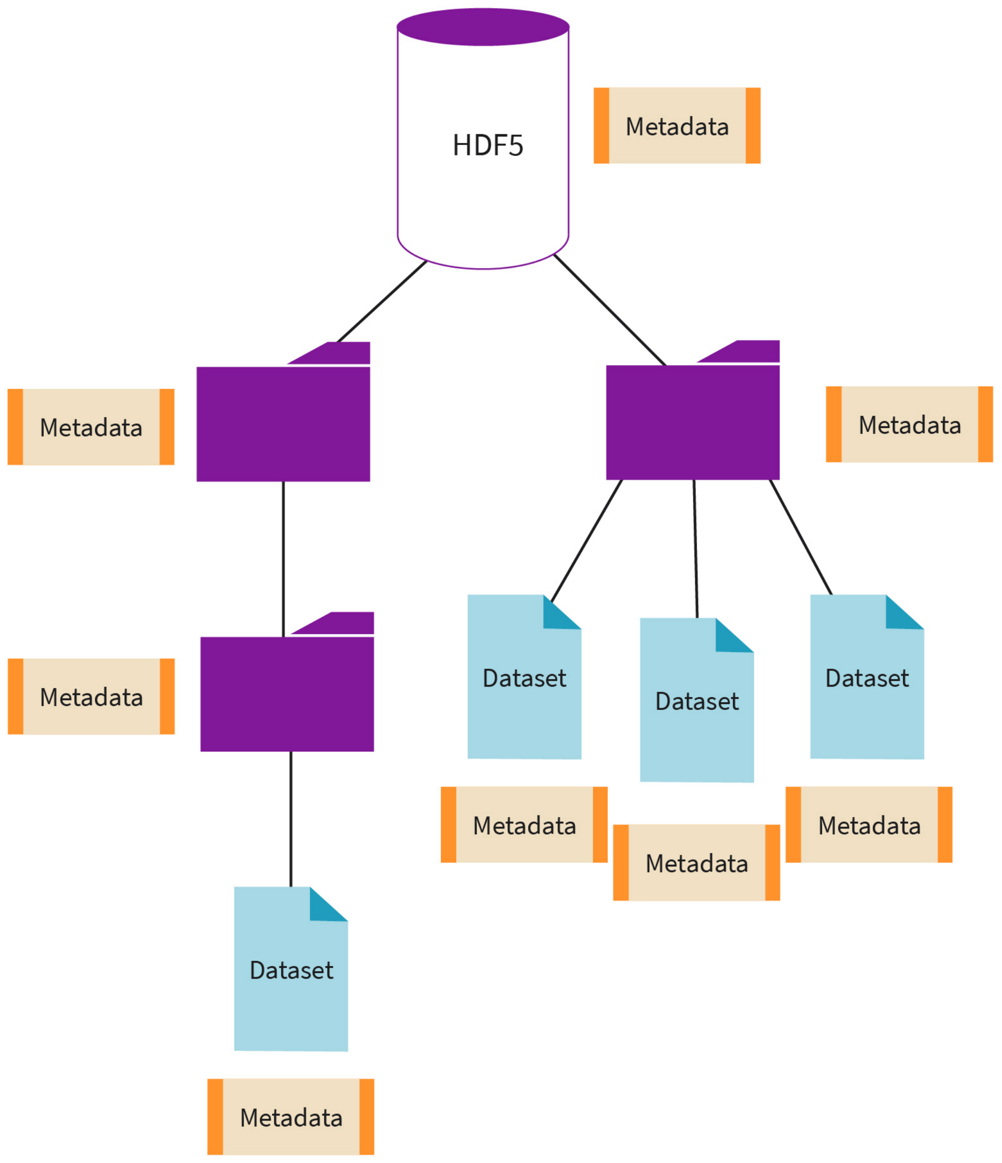
2.2 Binary Formats (HDF5, Parquet, Arrow)

In addition to the more traditional data formats, there are a variety of more modern storage methods that can store data more efficiently in specific use cases. In this section, HDF5, Parquet, and Arrow are introduced.

HDF5

The Hierarchical Data Format (HDF) contains a model that manages and stores large volumes of data. It is often used for complex and heterogeneous data. The HDF5 format is a file system that is contained and described in a single file. The contained file system works like the folder structure on a computer, though it has a different name: groups. Groups are like a folder and can contain a number of different files, which are called datasets. The figure below shows an example structure of HDF5.

Figure 10: HDF5 Data Model



Source: Wietreck (2021), based on mjones (2020)

HDF5 is a self-describing format, which means that each element (group or dataset) has an associated metadata file that contains information about the data. This includes names, descriptions, and any other documentation the user wants to add. This additional and separate metadata document allows users to automate processes. Other advantages of HDF5 include the storage of large complex and heterogeneous data, since the format was designed to compress high quantity of data. Additionally it supports data slicing, which means that only the files used for analysis are read into the computer's memory.

In the following example, the h5py module is used to create a HDF5 dataset in Python. After instantiating the HDF5 file, the `create_dataset()` method is used to build the dataset. Parameters for names, shape, and integer type are also handed over in that step. Attributes of the dataset can be accessed on the object. In the example, the shape, name, and parent are returned. In the following example, the created empty dataset does not belong to a group, i.e., parent, but we see that this data format allows us to group and nest our datasets and introduce a structuring hierarchy.

```
# load packages
import h5py

# create an HDF5 file
file = h5py.File('iu.h5','w')

# create an empty dataset in the HDF5 file
dataset = file.create_dataset("iu", (4, 6))

# print information about the dataset
print("Dataset shape is", dataset.shape )
print("Dataset name is", dataset.name)
print("Dataset is a member of the group", \
    dataset.parent )

# close the file.close()
```

In the following example, the previously created file is opened and the created dataset is accessed. Multidimensional example data are created using NumPy and then written into the dataset. From there, the data can now be read again.

```
# load packages
import numpy as np

# read/write HDF5 file
file = h5py.File('iu.h5','r+')

# list existing datasets in the file
list(file.keys())

# create an empty dataset in the HDF5 file
# if this dataset does not already exist
if not "iu_numbers" in list(file.keys()):
    dataset = file.create_dataset("iu_numbers", (4, 6))
else:
    dataset = file['/iu_numbers']

# generate sample data
data = np.random.rand(4*6).round(2).reshape(4, 6)
```

```
# add the data to the HDF5 dataset
dataset[...] = data

# read the data back from the HDF5 file
data_read = dataset[...]
print(data_read)
# console output:
# [[0.65 0.96 0.77 0.33 0.19 0.93]
#  [0.11 0.31 0.99 0.01 0.61 0.48]
#  [0.09 0.79 0.4  0.15 0.3  0.35]
#  [0.97 0.36 0.27 0.45 0.21 0.59]]

# close the file
file.close()
```

Metadata can be stored in the file using the .attrs[] method. The following script can be added to the example above and would add metadata information to the file.

```
dataset.attrs["User"] = "ME"
```

In order to access all metainformation, it is possible to iterate over the keys and return the values for each key.

```
for k in dataset.attrs.keys():
    print(k, dataset.attrs[k])
```

Parquet

Table 6: Row-Based versus Columnar Storage

Row-based storage	Columnar storage
1. Germany, Berlin, 83 2. France, Paris, 67 3. Italy, Rome, 60	Country: Germany, France, Italy Capital: Berlin, Paris, Rome Inhabitants: 83, 67, 60

Advantages of Parquet include the high compression capabilities, which can be chosen in a flexible manner. Different types of compression methods, in cooperation with extendable encoding schemas, are used to make the file as small as possible. Encoding methods include dictionary encoding, bit packing, and run length, encoding (RLE). Dictionary encoding is normally used for datasets where unique values are only available in a small number. Bit packing is used for the storage of integers with dedicated 32 or 64 bits per integer. This efficiently stores small integers. When a value occurs multiple times, RLE stores only one entry along with the number of occurrences.

Another advantage of Parquet is that it only retrieves the relevant data in an efficient manner since the amount of scanned data are small. This occurs because of the self-describing format where each file contains metadata as well as the data themselves. When querying the dataset, only the required columns and their respective values are loaded into memory.

Since Parquet is designed to keep the metadata separated from the actual data, it is possible to split columns into separate files which are then referenced using the metadata file. This enables the efficient and flexible handling of the data.

In Parquet, there are two main configurations that enable the optimization of the files. One configuration is related to the row group size, which allows the data to be chunked into larger pieces. The data page size configuration enables a single row lookup. By optimizing this, the space overhead is reduced.

To create, access, and write a Parquet file, there are a variety of possibilities in Python, and, for this example, the capabilities of pandas will be explored. First, the required modules are important. Pyarrow will be used in this example for the communication with pandas. After creating a sample dataset, it is written to an Arrow table. The `write_table()` method can be used to create a new Parquet file. Additional configuration parameters, such as the `data_page_size`, can be added.

```
# load packages
import pyarrow.parquet as pq
import pyarrow as pa
import pandas as pd
import numpy as np

# create a pandas DataFrame
df = pd.DataFrame(np.random.randn(100).\
    reshape(25,4), columns = ["one", "two", \
        "three", "four"])

# convert the pandas DataFrame to a parquet table
tableToWrite = pa.Table.from_pandas(df)

# write the parquet table to file
pq.write_table(tableToWrite, "myPQFile.parquet")
```

To read from a Parquet file, the `read_table()` method is used. After applying the `to_pandas()` method, the data are back in a DataFrame. By adding values in the columns parameter, it is possible to read only a subset of the Parquet file.

```
tableToRead = pq.read_table("parquet.writeTable")
tableToRead.to_pandas()
```

Attributes like .metadata or .schema will return information about the file, e.g., the number of columns, rows, groups, or version.

```
file = pq.ParquetFile("myPQFile.parquet")
file.metadata

<pyarrow._parquet.FileMetaData object at 0x00000279639E7E28>
  created_by: parquet-cpp-arrow version 4.0.1
  num_columns: 4
  num_rows: 25
  num_row_groups: 1
  format_version: 1.0
  serialized_size: 2739
```

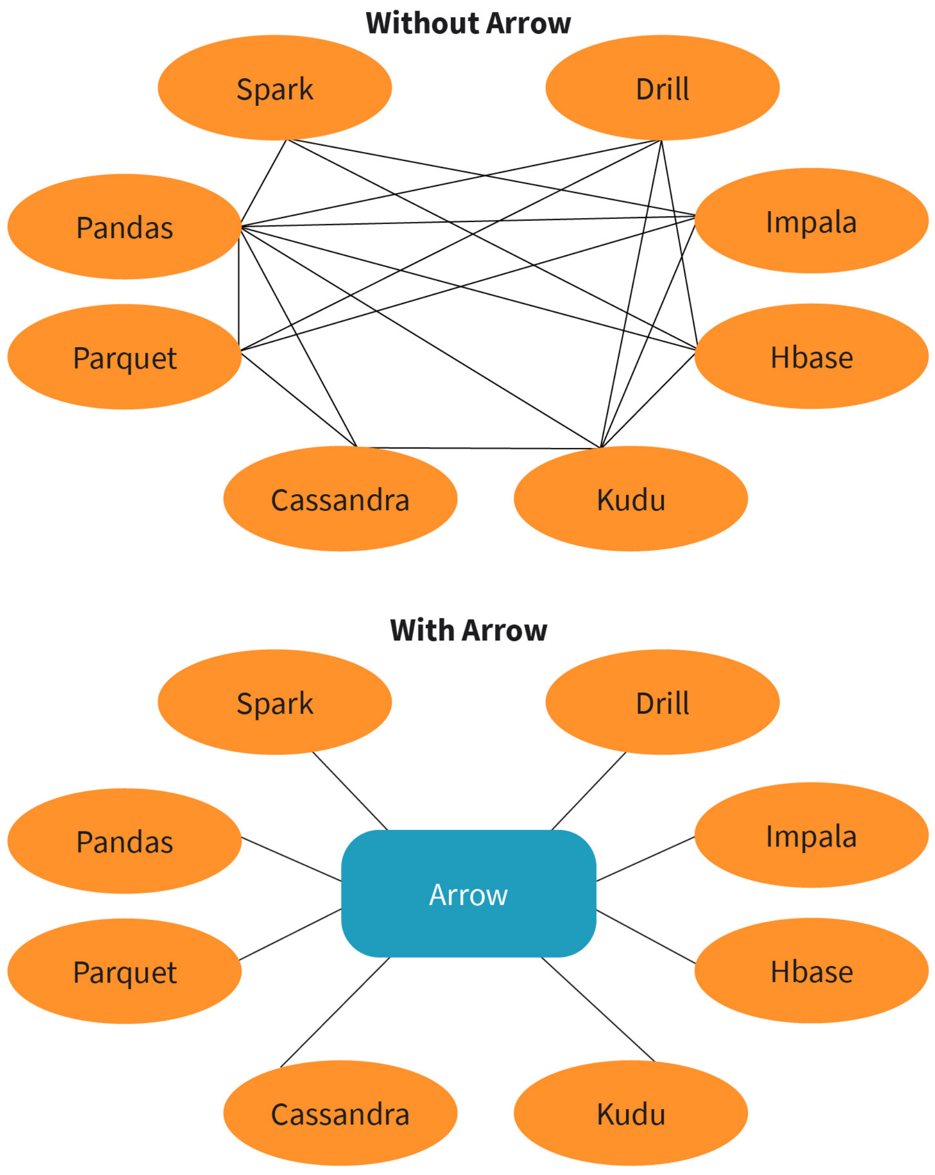
Source: Wietreck (2021).

Arrow

The development platform Arrow was built for in-memory analytics, which allows the processing and moving of data in a big data environment. This is especially useful when data need to be exchanged with a low overhead. This results in a high query performance, even for complex analytical tasks. It is optimized for the analysis of columnar data formats, either as flat files or as hierarchical data. The project is language agnostic, which allows a practical application in a variety of use cases. This facilitates an efficient communication between many components. In the following figure, Arrow helps to connect the following:

- Spark (framework for cluster computing)
- Drill (supports data intensive distributed applications)
- Impala (used for querying Hadoop)
- HBase (distributed relational database)
- Kudu (column-based data storage)
- Cassandra (database management system)
- Parquet (explained above)
- pandas (Python library for data analysis)

Figure 11: Arrow Overview



Source: Wietreck (2021), based on The Apache Software Foundation (n.d.).
The following figure shows how the data can look in memory.

Figure 12: Apache Columnar Storage Model

	id	time_stamp	value	
Row-Index 0	812316412	26.04.2021 08:34	0.213412	
Row-Index 1	812319761	12.04.2021 04:23	0.738321	
Row-Index 2	812314251	31.03.2021 18:24	0.523543	
Row-Index 3	812318463	28.04.2021 01:10	0.184732	
Row-Index 4	812315719	26.03.2021 14:43	0.965318	
	Traditional memory structure			Arrow memory structure
Row-Index 0	812316412		id	812316412
	26.04.2021 08:34			812319761
	0.213412			812314251
Row-Index 1	812319761			812318463
	12.04.2021 04:23			812315719
	0.738321		time_stamp	26.04.2021 08:34
Row-Index 2	812314251			12.04.2021 04:23
	31.03.2021 18:24			31.03.2021 18:24
	0.523543			28.04.2021 01:10
Row-Index 3	812318463			26.03.2021 14:43
	28.04.2021 01:10		value	0.213412
	0.184732			0.738321
Row-Index 4	812315719			0.523543
	26.03.2021 14:43			0.184732
	0.965318			0.965318

Source: Wietreck (2021), based on The Apache Software Foundation (n.d.).

To use Arrow in Python, Arrow tables can be created from other data structures, such as pandas DataFrames or NumPy arrays. In the following example, a few basic Arrow operations will be executed. After installing and importing the package, the `from_pandas()` method loads the example data into an Arrow table object.

```
# load packages
import pyarrow as pa
import pandas as pd
import numpy as np

# generate sample data
df = pd.DataFrame({ \
    'one': [20, np.nan, 2.5], \
    'two': ['january', 'february', 'march'], \
    'three': [True, False, True]}, \
        index=list('abc'))

# convert the DataFrame to an arrow table
table = pa.Table.from_pandas(df)
```

To make the process of reading data into an Arrow table as efficient as possible, there are a variety of methods, including `read_csv()` or `read_json()`. Additionally, the communication with the Parquet data format was made possible, which provides a standardized open-source columnar storage format. Arrow also makes it possible to write to and retrieve data from HDF5 files and can therefore be used as a link between HDFs, Parquet, and other data models. Methods like `to_pandas()` allow users to return data to a pandas DataFrame.

```
df_new = table.to_pandas()
```

In addition, it is possible to read many files in a directory as one dataset. This enables the analysis of even larger datasets. While this example only showed the main function of Arrow (to create the Arrow object), there are a couple of advantages over DataFrames. Besides containing extensible metadata information of the flat or nested data types, it is also possible to create user-defined types. Pandas objects tend to perform poorly when it comes to database or flatfile ingestions or export. Arrow, on the other hand, with its streaming and chunk-based oriented design, enables the movement and access of large datasets at maximum speeds. Overall, Arrow is designed for analytical processing performance and can be used to process very large datasets in a more efficient way than pandas DataFrames.

SUMMARY

There are many common formats that can store and handle data. Before data can be stored on a disc, they must be machine-readable. A common standard for de- and encoding is the eight-bit ASCII standard, which represents 128 characters, e.g., alphabetic, numeric, collating, or non-printing. The UTF-8 standard is also capable of displaying Greek letters and Asian symbols.

One of the most common data formats is the comma-separated values (CSV) format. It stores the values in a given structure, using a separator for the values. This results in a tabular layout that can easily be read, for example, using the pandas `read_csv()` method in Python. Afterwards, Python allows the data to be sorted and filtered while keeping the native tabular structure.

JSON does not have a tabular structure, and is mainly used for the transfer of data on the web. It follows a clean, easy to read structure, and can be adjusted to any type of data. The structure can be used like a Python dictionary.

The HDF5 format is newer than CSV. It consists of a data model and a storage model, which enable the efficient manipulation, storing, and transfer of data. The file format is based on groupings, which creates a hierarchy of different datasets. HDF5 can be created and managed using Python with the h5py library.

Parquet also emerged in the context of big data. Parquet uses a columnar storage approach where data are nested along the column headers. This enables the retrieval of data that are actually needed, resulting in higher performance and efficient use of memory.

Arrow can be used to facilitate the communication between different types of formats. While it is optimized for the handling of columnar data, it can also be applied to hierarchical data. Arrow is language agnostic and can be used in many use cases.

UNIT 3

NOSQL DATA STORES

STUDY GOALS

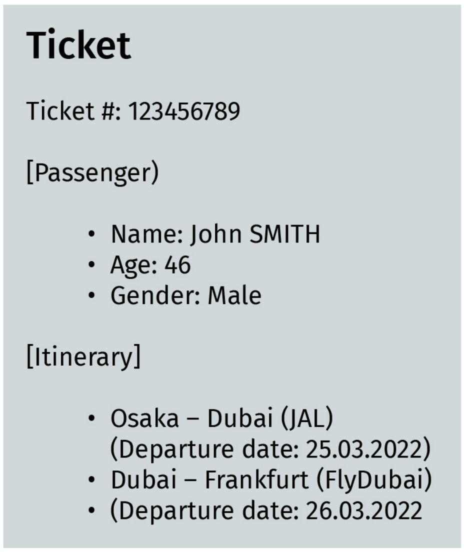
On completion of this unit, you will be able to ...

- describe the relational and non-relational data models
- explain how and why the NoSQL databases emerged
- classify the different storage solutions used in NoSQL databases
- practically use several NoSQL databases

Case Study

Imagine that you have been assigned with the task to design and implement a data-system for a travel agency. Amongst other requirements, this system should include a database for storing the information of purchased ticketsand printing them out. The tickets contain passenger's information and the itinerary (flights) are stored in the simple format shown below.

Figure 13: Sample ticket printout



Ticket

Ticket #: 123456789

[Passenger)

- Name: John SMITH
- Age: 46
- Gender: Male

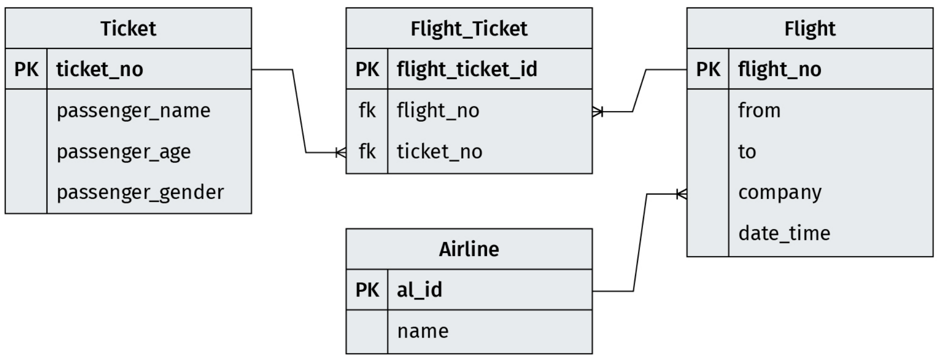
[Itinerary]

- Osaka – Dubai (JAL)
  (Departure date: 25.03.2022)
- Dubai – Frankfurt (FlyDubai)
- (Departure date: 26.03.2022

Source: Wannous, 2022

The database handles the following entities *ticket*, *airline*, and *flight,* as shown in the following figure. We will name this database *TicketDB.* We will consider that a ticket is for one passenger and it may involve several flights. On the other hand, a flight serves several passengers and is operated by one airline company.

Figure 14: The relational data model for TicketDB



Source: Wannous, 2022

During the course of this unit, we will learn that this is a use case for which a NoSQL database might be a good choice.

The term 'NoSQL' (Non-SQL or Not-only-SQL) is commonly used when referring to a collection of **database-engines**/data-stores that do not use the relational model for organizing data.

Database engine
The software that a database management system (DBMS) uses to manage data (create, read, update, and delete)

Several factors have contributed to the spread of NoSQL, like its dynamic schema, cost-effectiveness, speed, and better scalability in some cases. NoSQL database engines have been increasingly used since their introduction during the transition period between the 20th and the 21st centuries; nevertheless, NoSQL has not succeeded in changing the dominance of the relational database engines in the market.

NoSQL engines manage data in one of four primary models that we will see later. The selection of a suitable data model and, consequently, the NoSQL database engine for a specific project depends on the scenario and data dynamics.

3.1 Introduction and motivation

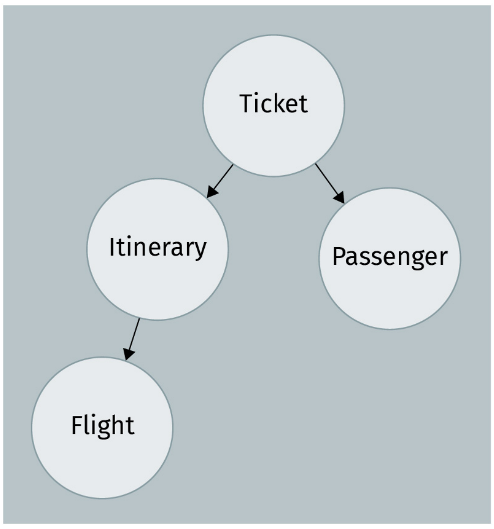
Three basic data models appeared during the last 30 years of the 20th century: 1) the *hierarchical* model, 2) the *network* model, and 3) the *relational* model. The *relational* model gained attraction and dominated the database market until this writing. In the following, we will briefly discuss all three basic data models.

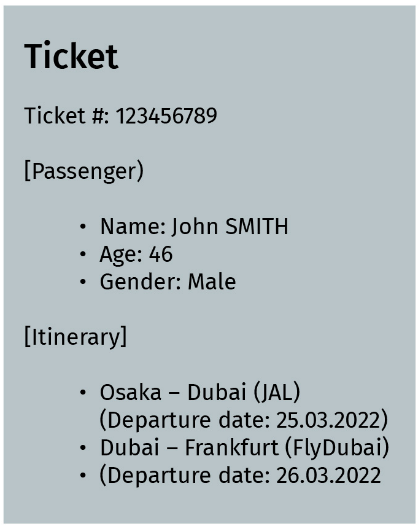
The hierarchical model

The *hierarchical* model organizes data in a tree of records nested within records; under each record (parent) come its children records. The popular database engine in the 1970s IBM's Information Management System (IMS), adopted this model (IBM, 2021).

The hierarchy can be as in the following figure for *TicketDB*.

Figure 15: The hierarchical data model





Ticket

Ticket #: 123456789

[Passenger)

- Name: John SMITH
- Age: 46
- Gender: Male

[Itinerary]

- Osaka – Dubai (JAL)
  (Departure date: 25.03.2022)
- Dubai – Frankfurt (FlyDubai)
- (Departure date: 26.03.2022

Source: Wannous, 2022, based on IBM, 2021

In this example, the root record 'Ticket' has two children, 'Passenger' and 'Itinerary,' and the record 'Itinerary' is the parent of 'Flight. One 'Ticket' may contain several 'Flights,' representing a one-to-many relationship.

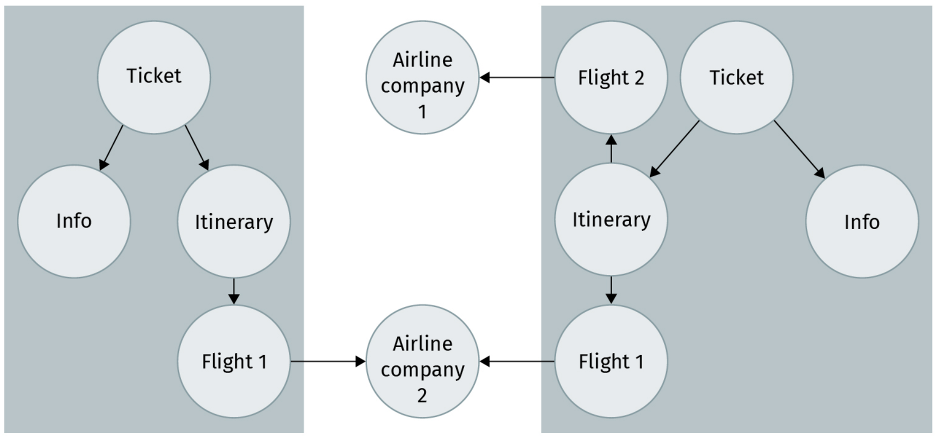
The *hierarchical* data model performed well with one-to-many relationships. Still, it wasn't easy to adopt many-to-many relationships (for example, a flight can accommodate several passengers, and a passenger may reserve a multi-flight ticket). Software developers had to handle this type of relationship in the application code, making the development process more difficult.

The network model

The *network* model appeared as a generalization of the *hierarchical* model (Hainaut, 2009). A node may have multiple parent nodes in the network model, which addresses the many-to-many relationship issue of the *hierarchical* model.

The network data model can be as in the following figure for *TicketDB*.

Figure 16: The network data model



Source: Wannous, 2022

The figure shows one configuration where a single ticket can incorporate flights from different companies, and a single airline company may serve tickets to multiple passengers.

The way to access a node in a *network* model was to follow a path (called the *access* path) from a root node through a series of other nodes while checking the matching condition. Imagine a case where a developer wants to find the tickets purchased from airline co.1. It was necessary to scan all nodes while keeping the *path* information for a possible matching until finding the specific airline company and then add the path to the results before seeing the rest of the parents.

The *network* data model is simple to understand and design as it involves only nodes and links between them (although this can become complex for a large number of nodes, edges, and relationships between them). In the following section, we will come to know about graphs which are the building block for a type of NoSQL database. Graphs are also composed of nodes and links between them, but with additional attributes that can be assigned to both. In the graph-oriented databases we will also see that it is no longer necessary to store and reproduce complex access paths.

The relational model

The *relational* model places all data in tables which are simple structures comprising rows, with each row having a set of attributes (columns) (IBM Cloud Education, 2019). It negates the need for nested structures and simplifies the various data manipulation operations via the SQL interface.

The figure shown in the introductory case study for this unit shows what the relational model could look like for our simple database, *TicketDB*. The figure shows an additional table, 'Flight_Ticket,' added to realize the many-to-many relationship between the Ticket and the Flight tables. One-to-many relationships are implemented by inserting **foreign keys** in the child tables to point to the **primary keys** in their parent tables.

Foreign key
A column (or a set of columns) in a table that links (refers) to a column in another table in a relational database

Primary key
A column (or a set of columns) in a table that uniquely identies a row in a table

Table schema
The definition of a table that includes its name and its attributes' names, data types, and constraints.

Relational Database Management Systems (RDMS) impose several rules on the values stored in the database and strictly apply these rules to ensure data consistency. Every table has a **schema** against which data values are validated and constraints checked before writing.

Altering the schema after data has been inserted into the table could be tedious and time-consuming and might fail in some cases. Still, altering the schema is necessary for many situations as new rules/policies exist all the time. For example, authorities might require adding an emergency contact to every passenger's record, which requires changing the schema of the table 'Passenger' to accommodate a new attribute, 'emergency_contact.'. While it is acceptable to have the attribute for the new records, existing rows in the table don't have it, and this might be an issue if the feature can't be allowed to be empty (NULL).

The Rise of NoSQL

Several driving forces led to the birth of NoSQL in the early 2000s. Many appeared in the software development process, while others came from the database field (Stonebraker, 2010).

The tight and inflexible schema in the relational model, issues with the mapping between an object in Object-Oriented Programming (OOP) and the column values in a table, limited scalability options, and the expensive operations to join tables are some of these factors.

Consider the case where a developer needs to fetch the information of a specific ticket and display it on the user's screen from the *TicketDB*. A query involving several joins is required to collect all the information.

```
SELECT …
FROM `Flight_Ticket` LEFT JOIN `Ticket` ON …
LEFT JOIN `Flight` ON …
LEFT JOIN `Airline` ON …
WHERE `ticket_no`=…;
```

As the database size increases, handling the information stored in it becomes heavier, and scalability comes into focus. Consider a small airline company that operates tens of flights a day. It is manageable to add new flights, and their related tickets go into the database. But new unique records constantly appear in large volumes to the other tables for a large airline company serving hundreds of flights a day is a severe issue. Individual records should go into **indexes**, and they will appear as foreign keys in other tables. You might argue that we can break the 'Flight' table to make part of it constant (flight_no, from, etc.), but this means a new table appears in the query involving the join procedure.

Index
A lookup table for quickly finding rows users frequently search

NoSQL emerged with a promise to address the weaknesses of the relational model, especially its inflexibility and scalability (especially horizontal scalability, but keep in mind that relational databases are also horizontally scalable). Also as large volumes of data are required to be analyzed with high speed in data analysis applications, NoSQL also aimed at addressessing this issue.

A document representing a *ticket* inside our ticketing example might look something as below:

```
{
    "ticket_no": 1234,
    "passenger_name": "John SMITH",
    "passenger_age": 46,
    "passenger_gender": "Male",
    "Flights": [
      {
        "flight_no": "JL12345",
        "from": "Osaka",
        "to": "Dubai",
        "airline": "JAL",
        "date_time": "25.3.2022 16:00"
      },
      {
        "flight_no": "FD123",
        "from": "Dubai",
        "to": "Frankfurt",
        "airline": "FlyDubai",
        "date_time": "26.3.2022 16:00"
      }
    ]
  }
```

It is clear that the record contains all the information needed formatted in JSON. This record can be fetched in one read operation and converted to an OOP object or even transmitted as a string directly. The record structure is not static (flexible schema); it can be determined upon reading it from the database. Records of different components are allowed (a ticket with the departure and arrival date/time, for example), unlike the case of relational databases.

NoSQL stores data similarly to the *hierarchical* data model, i.e., records are enclosed inside records rather than in tables. One of its essential difference from the *relational* data model is that a record is *self-contained,* which makes it possible to split the database among a number of servers, if necessary, to balance the load and scale **vertically** and/or **horizontally**.

vertically
Adding resources to a single machine or server to cope with the increased demand.

Horizontal scaling
Adding more machines to the infrastructure to cope

NoSQL can handle data workloads that require rapid processing and analysis of huge amounts of varied and unstructured data because of its schema flexibility.

with the increased demand.

3.2 Approaches and technical concepts

NoSQL databases hold the capability to handle a massive volume of quickly changing data (Microsoft, 2022). They also support developers working in agile environments where unplanned situations are frequent.

Based on their storage and data models, the research community as well as database developers and provideers subsume the many different flavors of NoSQL databases under four main categories (Gourav Bathla, 2018) (Microsoft, 2022):

- Key-value datastore
- Document datastore
- Columnar datastore
- Graph datastore

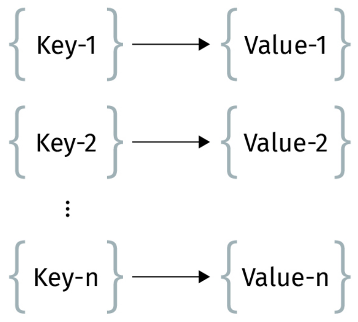
The categories are different in many aspects, and each one addresses different aspects and requirements of certain use cases. But, as different as the many NoSQL databases might be, they share the common characteristic that they are fundamentally different from relational databases in one of more aspects.

Key-value datastore

Hashmap
A structure that stores several entries in which a value is mapped to a key.

Key-value datastores associate values (of primitive and complex types) to keys, similar to a dictionary/**hashmap** implementation in several programming languages (Microsoft, 2022) (Gourav Bathla, 2018). The keys in the datastore are unique and don't repeat, while the data values do (if necessary). Key-value databases are compact and have efficient index structures to quickly and reliably locate a value using its key. They are ideal for systems that do simple lookups, like the modules that handle application preferences and user profiles. They are not a good choice if a schema is necessary.

Figure 17: Key-value data model



Source: Wannous, based on Microsoft, 2022

Key-value datastores perform simple operations on their entries including

- reading
- deleting
- updating

a value addressable by a given key.

A new entry in the datastore creates a new key-value pair, and the subsequent operations on the entry involve passing the key to access it. A new entry with a key already in the datastore results in the old value being replaced with the new one, and the key remains unchanged.

Redis (Redis, Inc, 2022) is an example of a key-value data store that uses a distributed architecture to process large amounts of data in parallel and in-memory. Redis is an open source database, but there is also a company that provides an enterprise version of it. Redis is available through the official downloads page for direct download, as a docker image, and ready for usage in the cloud on provided and managed infrastructure by the Redis company. At writing, Redis offers an online console to interact with a sample datastore, and it offers a free trial on its cloud platform.

Connect to a Redis in Python

The following steps show how to connect to a Redis database and manipulate data in a Python application.

1. Download and install Redis on your computer, or start a free trial in the cloud.
2. Create a database inside Redis.
3. Install the `redis-py-cluster` Python package by running the following command in the terminal.

   ```
   pip install redis-py-cluster
   ```

4. Open any IDE that supports Python, and write the following code in a file and name it (`Redis-Example.py`). Replace the text between < and > with the respective values of your installation. The code has been tested with a Redis database installed in the cloud.

```
# import the redis module
import redis

# if connecting to a cloud database,
# import the dns module, too

import dns
.
# connect to the database
# replace <…> with the respective values of your DB
r = redis.Redis(host='<Server's IP / cloud endpoint>', \
```

```
port=<port>, password='<password>')

# add two new key-value entries
r.set('key-1', "value-1")
r.set('key-2', "value-2")
.
# change the value of the first entry
r.set("key-1", "new value")
.
# retrive the entries from the database and print them
value1 = r.get("key-1")
value2 = r.get("key-2")
.
print("The retrieved values are:")
print(value1)
print("=============")
print(value2)
```

The code will produce the below output.

```
The retrieved values are:
b'new value'
=============
b'value-2'
```

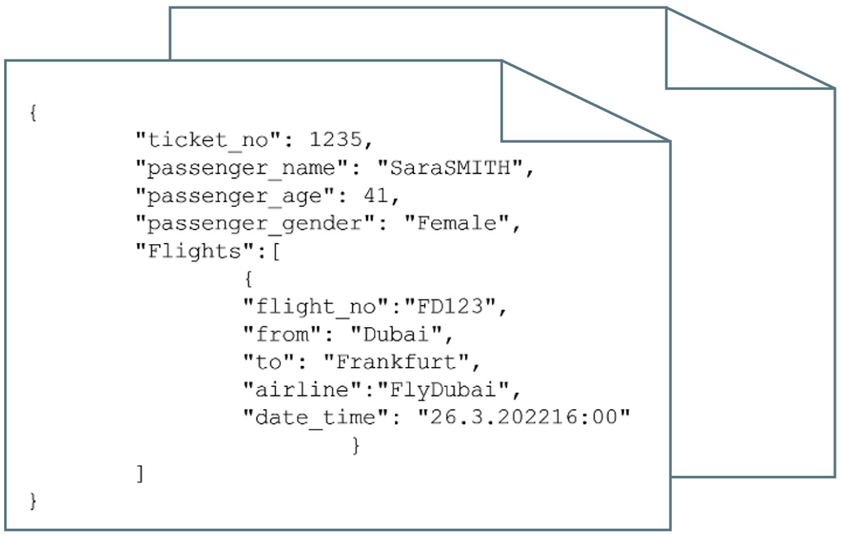
Note that the `"key-1"` entry was updated after the second writing operation (`r.set("key-1", "new value")`) and the old value was replaced.

Document datastore

JSON
An open standard format for storing data objects as text.

A set of fields and objects arranged in a specific format like **JSON** or XML is the building block in *document* datastores (Gourav Bathla, 2018). Documents can be organized in collections similar to tables in relational databases.

Figure 18: Document data model


```
{
        "ticket_no": 1235,
        "passenger_name": "SaraSMITH",
        "passenger_age": 41,
        "passenger_gender": "Female",
        "Flights":[
                {
                "flight_no":"FD123",
                "from": "Dubai",
                "to": "Frankfurt",
                "airline":"FlyDubai",
                "date_time": "26.3.202216:00"
                        }
        ]
}
```

Tickets Collection

Source: Wannous, 2022

In the example below, we see a sample collection of documents, with each representing a ticket in the NoSQL version of *TicketDB,* discussed earlier in this unit, stored as a JSON string.

```
{
    "ticket_no": 1234,
    "passenger_name": "John SMITH",
    "passenger_age": 46,
    "passenger_gender": "Male",
    "Flights": [
      {
        "flight_no": "JL12345",
        "from": "Osaka",
        "to": "Dubai",
        "airline": "JAL",
        "date_time": "25.3.2022 16:00"
      },
      {
        "flight_no": "FD123",
        "from": "Dubai",
        "to": "Frankfurt",
        "airline": "FlyDubai",
        "date_time": "26.3.2022 16:00"
      }
    ]
```

```
  }
{
    "ticket_no": 1235,
    "passenger_name": "Sara SMITH",
    "passenger_age": 41,
    "passenger_gender": "Female",
    "Flights": [
      {
        "flight_no": "FD123",
        "from": "Dubai",
        "to": "Frankfurt",
        "airline": "FlyDubai",
        "date_time": "26.3.2022 16:00"
      }
    ]
  }
```

The collection contains two documents, and each one has several simple fields like the `passenger_name` and `passenger_age` and a more sophisticated field, `Flights` (an array of objects).

A document can be self-contained, i.e., it has all the fields embedded in it. The application code needs one read operation to acquire all the fields necessary to construct an object representing the document. The previous documents are self-contained, but they have repeated fields that take uneccesarily larger space. A better approach to save storage is to use document references. The example below shows that the flight information is stored in a separate collection, and the ticket document references these flights.

```
{
    "flight_no": "JL12345",
    "from": "Osaka",
    "to": "Dubai",
    "airline": "JAL"
}
{
    "flight_no": "FD123",
    "from": "Dubai",
    "to": "Frankfurt",
    "airline": "FlyDubai"
}
{
    "ticket_no": 1234,
    "passenger_name": "John SMITH",
    "passenger_age": 46,
    "passenger_gender": "Male",
    "Flights": [
      {
        "flight_no": "JL12345",
        "date_time": "25.3.2022 16:00"
      },
      {
        "flight_no": "FD123",
        "date_time": "26.3.2022 16:00"
      }
    ]
  }
{
    "ticket_no": 1235,
    "passenger_name": "Sara SMITH",
    "passenger_age": 41,
    "passenger_gender": "Female",
    "Flights": [
      {
        "flight_no": "FD123",
        "date_time": "26.3.2022 16:00"
      }
    ]
  }
```

In the *document* data model, the schema is flexible and is determined when reading the document in the application. For example, one document of our sample collection might have the field `passenger_gender` and another might not have that information. On the other hand, a drawback of this model is that even small changes in the document require writing the whole document to the database. For that reason, it is recommended to keep the size of the document small.

MongoDB (n.d.), the name stems from “humongous”, meaning “gigantic”, is an example of a *document* data store written in C++. Until 2018, MongoDB was an open source database; today there is also a proprietary version distributed by the MongoDB company. It is available through the official downloads page for direct download and as a cloud service. In addition, there are Docker images available for MongoDB as well. At writing, MongoDB offers an online console to interact with a sample datastore, and it offers a free trial on its cloud platform.

Connect to a MongoDB in Python

The following steps show how to connect to a MongoDB database and manipulate data in a Python application.

1. Download and install MongoDB on your computer, or start a free trial in the cloud.
2. Create a database inside MongoDB.
3. Install the Python package `pymongo` by running the following command in the terminal.

```
pip install pymongo
```

4. Open any IDE that supports Python, write the following code in a file and name it `Mongo-Example.py`. The code has been tested with a MongoDB database installed in the cloud. Replace the text between < and > with the respective values of your installation. The centerpiece code in this example to connect to a Mongo database from Python makes use of the `pymongo.MongoClient()` function.
   In this example, we connect to a mongodb on a server (`srv`) using a `<user name>`, `<password>`, and the `<Server's IP endpoint>`. We also specify that writes to the database should be retried if they fail, e.g. due to network problems (`retryWrites`). Writes will be applied to several data replications in the cluster. Accordingly, we specify a strategy for writing errors during the write process to these data replicas. By setting this strategy to `majority`, writes will only be rolled back if the error occurred before writes took place to less than half of the data replications, otherwise, they will endure allowing eventual consisteny in the database.

```
# import the pymongo module
import pymongo

# if connecting to a cloud database,
# import the dns module
import dns

# connect to the database
client = pymongo.MongoClient( \
"mongodb+srv://<user name>:<password>@< Server's IP / cloud
endpoint >?retryWrites=true&w=majority")

# add a database
mydb = client["TicketDB"]

# add two collections for the flights and tickets
flightCollection = mydb["flights"]
ticketCollection = mydb["tickets"]

# add two flights documents to the collection
flight1 = { \
'flight_no': 'JL12345', \
'from': 'Osaka', \
'to': 'Dubai', \
'airline': 'JAL'}
flightCollection.insert_one(flight1)

flight2 = { \
'flight_no': 'FD123', \
'from': 'Dubai', \
'to': 'Frankfurt', \
'airline': 'FlyDubai'}
```

```
flightCollection.insert_one(flight2)

# add two tickets documents to the collection
ticket1 = { \
'ticket_no': 1234, \
'passenger_name': 'John SMITH', \
'passenger_age': 46, \

'passenger_gender': 'Male', \
'Flights': [{ \
'flight_no': 'JL12345', \
'date_time': '25.3.2022 16:00'}, \

{'flight_no': 'FD123', \
'date_time': '26.3.2022 16:00'} \
] \
}

ticketCollection.insert_one(ticket1)

ticket2 = { \
'ticket_no': 1234, \
'passenger_name': 'Sara SMITH', \
'passenger_age': 41, \

'passenger_gender': 'Female', \
'Flights': [{ \
'flight_no': 'FD123', \
'date_time': '26.3.2022 16:00'} \
] \
}

ticketCollection.insert_one(ticket2)

# create a query to find Sara's ticket
query = { 'passenger_name': 'Sara SMITH' }

# execute the query
results = ticketCollection.find(query)

# print the results to the console
for result in results:
    print(result)
```

The code will produce the output below.

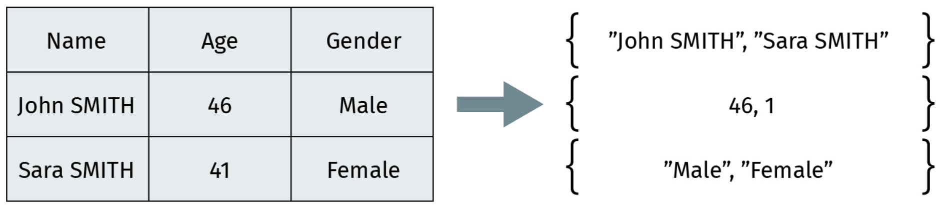
```
{'_id': ObjectId('xxxxxxxxxxxxxxxxxx'), 'ticket_no': 1234,
'passenger_name': 'Sara SMITH', 'passenger
_age': 41, 'passenger_gender': 'Female', 'Flights':
[{'flight_no': 'FD123', 'date_time': '26.3.2022 16:00'}]}
```

Columnar datastore

A columnar datastore organizes data into columns and rows as in a relational database (Microsoft, 2022) (Gourav Bathla, 2018). The difference is that relational databases are optimized for row operations while columnar databases are optimized for column operations.

The figure below shows a simple example of a table in a relational database (left) and how it appears in a columnar database (right).

Figure 19: Columnar data model



Source: Wannous, 2022, based on Gourav et al., 2018

Tables in a columnar database are named *column-families,* and each column-family may contain one column or more. At the storage level, all columns in a column-family are stored in one file making it easy to aggregate the values of a column and reducing the amount of data that is necessary to fetch from the database in **analytics applications**.

Analytics Applications
applications used to quantitively measure and improve business processes

Consider a case when a query in a relational database is executed to obtain only the 'Age' attribute from the table shown in the figure above (left). The relational database engine will run over all rows while scanning different fields of different data types and filtering the results as per the query. A similar query in a columnar datastore will run on the table (or table-family) containing the required attribute(s) of one data type. The data processed in the two cases are very different in size and type variety. When the data is written to the columns, replicas are allowed.

Apache Cassandra (Apache Cassandra, n.d.) is an example of a *columnar* data store. It is managed and licensed under the Apache Foundation umbrella, but it originated at Facebook. Like other Apache projects, Cassandra is an open-source project, it is written in the Java programming language, and it is identified as a largely distributed NoSQL datastore. It uses a query language (Cassandra Query Language -CQL) similar to SQL but has its flavor. Cassandra is available for download as a stand-alone package and a docker image.

Connect to Apache Cassandra in Python

The following steps show how to connect to a Cassandra database and manipulate data in a Python application.

1. Download and install Cassandra on your local machine.
2. Create a database called 'people'. A database in Cassandra is named *KEYSTORE*, and the query used to create it requires one argument *replication* to identify the replication strategy and options for the keyspace (Cassandra places several data replicas on different nodes to ensure reliability. The number of replicas is known as the *replication factor*).
3. Install the `Cassandra-driver` package for Python by running the following command in the terminal.

```
pip install Cassandra-driver
```

Open any IDE that supports Python, write the following code in a file and name it `Cassandra-Example.py`. The code has been tested with an Apache Cassandra database installed on a local computer.

```
# import the cassandra cluster module
from cassandra.cluster import Cluster

# import a statement object to run queries
from cassandra.query import SimpleStatement
# import the consistency level module
from cassandra import ConsistencyLevel

# create a clster and connect to the database
cluster = Cluster()
session = cluster.connect()

# create a keyspace (a database in Cassandra)
# and set it as the default
session.execute("""CREATE KEYSPACE people
WITH replication={'class':'SimpleStrategy',
'replication_factor': '2'}""")
session.set_keyspace('people')

# create a table called 'person'
# set the primary key and specify
# the datatypes of the columns
session.execute("""CREATE TABLE person (person_key text,
name text, gender text, age text,

PRIMARY KEY (person_key))""")
```

```
# make a query to insert one person into the table.
# The consistency level indicates how many data
# replicas should reply to the
# query before reporting success to the client
query = SimpleStatement("""INSERT INTO person
(person_key, name, gender, age)
VALUES ('001', 'John SMITH', 'Male', '46')""",
consistency_level=ConsistencyLevel.ONE)

# execute the query
session.execute(query)

# insert another person
query = SimpleStatement("""INSERT INTO person
 (person_key, name, gender, age)
VALUES ('002', 'Sara SMITH', 'Female', '41')""",
consistency_level=ConsistencyLevel.ONE)

# execute the query
session.execute(query)

# report progress to the console
print("Inserted two persons into the database\n")

# read the information from the database and display it
print("Trying to read the info of one of them\n")
future = session.execute_async("""SELECT * FROM person
WHERE person_key='002'""")
    rows = future.result()

for row in rows:
    print('\t'.join(row))

# clear the DB (because this is for testing only)
print("Clearing the database")
session.execute("DROP KEYSPACE people")
```

The code will produce the console output below.

```
Inserted two persons into the database
Trying to read the info of one of them

002     41      Female  Sara SMITH
Clearing the database
```

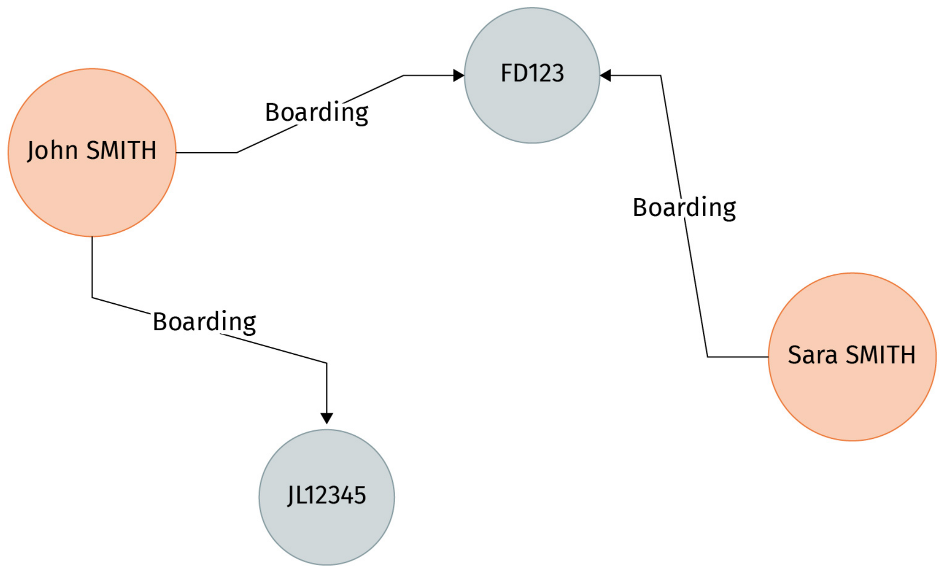
Graph datastore

Imagine that you would like to store data that is heavily characterized by relations. For example, for your use case, you would like to use a database to quickly find out which bus stop is best connected with other bus stops within a city. Or imagine that you try to find the most frequently used airports by certain airlines on long flight connections. These are examples of data for which connections between entities are the foremost relevant pieces of information. Two types of information exist in graph datastores: nodes and edges/links. Nodes represent entities, and links specify the relationships between these entities (Microsoft, 2022). The concept is similar to the *network* data model discussed earlier in this unit and the class diagram in system analysis.

Nodes and edges can be attributed in a way that we can specify additional features to them. For example, the "Boarding" edges in the figure below might have the attributes "priority boarding" or "wheel chair appropriate".

The edges in the graph are typically directed so that they have a starting and an end point. In the figure below, the "Boarding" edges origins are persons and they are directed towards the flights.

Figure 20: Graph data model



Source: Wannous, 2022

The structure in the figure above represents one implementation of *TicketDB* with two passengers boarding two flights. It makes it easy to execute queries like "Find all people who will board flight JL1234". It is possible to perform complex analyses quickly on large structures involving many entities and links.

Neo4j (neo4j, n.d.) is an example of a *graph* data store. It is licensed in a hybrid model and two versions exist open-source (community edition) and enterprise. Node4j is written in the Java programming language and it is a popular graph/OLTP DBs. Docker images of neo4j are available in addition to cloud deployment (the model used in the example below). Neo4j uses query languages such as Cypher or Apacha Tinkerpop.

Connect to neo4j in Python

The following steps show how to connect to a neo4j database and manipulate data in a Python application. The code implements the *TicketDB* comprising two tickets boarding two flights by instantiating two flights and two passengers and linking them with three links.

1. Setup a neo4j database in the cloud or install it on your local machine.
2. Create a test database in it and obtain the parameters to access it from a python application.
3. Install `neo4j` package by running the following command in the terminal

   ```
   pip install neo4j
   ```

4. Open any IDE that supports Python, write the following code in a file and name it (`Neo4j-Example.py`). The code has been tested with a neo4j database installed in the cloud. Replace the text between < and > with the respective values of your installation.

```
#import the neo4j module
from neo4j import GraphDatabase

# import dns (necessary for the cloud instance)
import dns

# establish a connection to the database
driver = GraphDatabase.driver(\
    "neo4j+s://<Ip address/cloud end-point>", \ auth=("neo4j",
<password/token>"))

# connect to the database
session = driver.session()

#insert two new nodes for the tickets
session.run("CREATE (n:Ticket {ticket_no: 1234,
    passenger_name: 'John SMITH', passenger_age: 46,
passenger_gender: 'Male'})")
session.run("CREATE (n:Ticket {ticket_no: 1235,
    passenger_name: 'Sara SMITH', passenger_age: 41,
passenger_gender: 'Female'})")
```

```
# insert two flights
session.run("CREATE (n:Flight {flight_no: 'JL12345',
    from: 'Osaka', to: 'Dubai', airline: 'JAL'})")
session.run("CREATE (n:Flight {flight_no: 'FD123',
    from: 'Dubai', to: 'Frankfurt',
    airline: 'FlyDubai'})")

# insert three relationships
session.run("""MATCH

    (a:Ticket),
    (b:Flight)
    WHERE a.ticket_no = 1234 AND b.flight_no = 'JL12345'
    CREATE (a)-[r:Boarding
        {date_time: '25.3.2022 16:00'}]->(b)
    RETURN type(r)""")

session.run("""MATCH
    (a:Ticket),
    (b:Flight)
    WHERE a.ticket_no = 1234 AND b.flight_no = 'FD123'
    CREATE (a)-[r:Boarding
        {date_time: '26.3.2022 16:00'}]->(b)
    RETURN type(r)""")

session.run("""MATCH
    (a:Ticket),
    (b:Flight)
    WHERE a.ticket_no = 1235 AND b.flight_no = 'FD123'
    CREATE (a)-[r:Boarding
        {date_time: '26.3.2022 16:00'}]->(b)
    RETURN type(r)""")
```

The following query is used to insert a new relationship between two nodes and it worth explaning due to its importance.

```
MATCH(a:Ticket),(b:Flight)WHERE a.ticket_no = 1234 AND b.flight_no =
'JL12345' CREATE (a)-[r:Boarding{date_time: '25.3.2022 16:00'}]->(b)
```

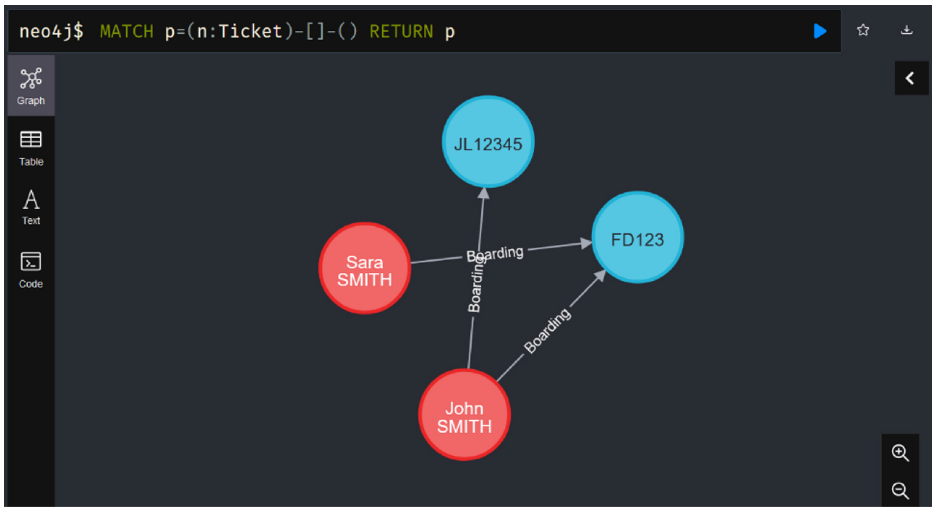
The part `MATCH(a:Ticket),(b:Flight) WHERE a.ticket_no = 1234 AND b.flight_no = 'JL12345'` simply says 'in the flowing part match a to a node of the type `Ticket` and `b` to a node of the type `Flight`. The part `CREATE (a)-[r:Boarding{date_time: '25.3.2022 16:00'}]->(b)` instructs to create a relation names `Boarding` (`[r:Boarding{date_time: '25.3.2022 16:00'}]`) originating from node `a` to node `b`. The relation has an additional attribute `date_time`.

Showing the database as a graph is beyond the scope of the code; but this part will be executed in the cloud console as follows (or alternatively in the Desktop GUI).

Run the following query to view the graph of the whole database.

```
MATCH p=(n:Ticket)-[]-() RETURN p
```

Figure 21: Graph database representation in the neo4j console – example 1



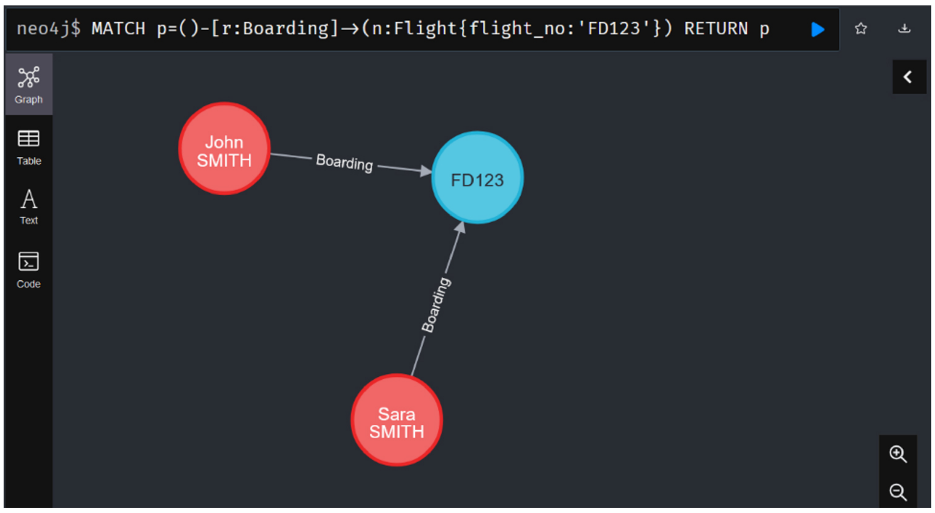
```
neo4j$ MATCH p=(n:Ticket)-[]-() RETURN p
```

Source: Wannous, 2022

Run the following query to show only the tickets of the passengers boarding the flight with flight_no='FD123'.

```
MATCH p=()-[r:Boarding]->(n:Flight{flight_no:'FD123'}) RETURN p
```

Figure 22: Graph database representation in the neo4j console – example 2



```
neo4j$ MATCH p=()-[r:Boarding]→(n:Flight{flight_no:'FD123'}) RETURN p
```

Source: Wannous, 2022

Selecting NoSQL database

NoSQL engines take fundamentally different approaches to data storage than relational databases, and they have distinct features that make each of them suitable for a range of applications.

The table below summarises the characteristics of the four NoSQL data models we have come to study and the type of application that each of them fits.

Table 7: Comparison of NoSQL Database Categories

Parameter	Key-value	Document	Columnar	Graph
Storage	Unique key with value	JSON, XML, BSON	Columns	Nodes and edges
Example Applica-tions	Indexing, IoT sen-sor data	Programming objects storage, semi-structured collections of data, IoT sensor data	Sparse Data, structured data with frequent column access	Modelling rela-tionships
Example Databa-ses	Redis DynamoDB Ignite Oracle NoSQL	MongoDB CouchDB OrientDB	Cassandra BigTable HBase CosmosDB (multi model)	Neo4j InfoGrid

Source: Müller-Kett, 2022, based on Gourav et al., 2018

SUMMARY

NoSQL database engines have emerged to address several issues with the *relational* model, such as schema flexibility. They have been around since the beginning of the 21st century and have proven NoSQL is a tough competitor to relational databases for some modern use cases. Nevertheless, NoSQL engines have not so far succeeded in shifting the relational database engines from dominating the database market.

NoSQL databases follow several approaches to storing data, and each of these approaches is suitable for a specific range of applications. The only characteristic common to them is that they don't follow the relational model. According to the different approaches they persue in modeling the data, NoSQL databases can be categorized into key-value, document, columnar, or graph databases. They support developers working in environments where requirements change frequently. This comes from the fact that these databases do usually not follow a strict schema, are distributed across multiple machines and come with strategies for efficient storing and querying the data. This makes them applicable for many modern applications that use data with high volume, velocity, variety and veracity, aka Big Data applications.

UNIT 4

DISTRIBUTED SYSTEMS

STUDY GOALS

On completion of this unit, you will be able to ...

– explain the difference between types of distributed systems
– describe the goals of a distributed system.
– describe the components of a distributed system.
– discuss the benefits and challenges of distributed systems.
– use distributed systems in big data applications.

4. DISTRIBUTED SYSTEMS

Case Study

Let's imagine a fictional social network that abides by general data protection ethics and rules that we call CivilConnect. This network brings people and opinions together by providing reflective views and insightful dialectics on complex issues. CivilConnect constantly aims at improving its sincere intrinsic motivation to engage people to cooperate and solve problems in a civilized manner. Every day, uses of CivilConnect share billions of pieces of content, including photos, videos, and thoughts. All this data needs to be processed so users can see what their friends are up to. To do this processing, CivilConnect uses a distributed system called Hadoop. Hadoop is a framework that allows for distributed processing of large data sets across many servers. It does this by splitting the data into smaller chunks, which are then processed independently. The advantage of using a distributed system is that it can handle large amounts of data more efficiently than a single server can. This makes the overall process faster and more efficient. CivilConnect, in addition to Hadoop, uses a distributed system called Spark for processing the data. Spark is similar to Hadoop, but it can be faster for some use cases and is easier to handle. This makes it a good choice for applications that require fast processing speeds or the ability to experiment with large amounts of data, such as in near real-time analytics or machine learning.

Big data technologies are used to process large amounts of data. In this unit, we will learn how Hadoop and Spark can work together to process big data. Hadoop is a family of technologies that use distributed systems to handle large amounts of data. MapReduce is part of the Hadoop ecosystem. It is a programming model that splits up big data sets into smaller pieces that can be processed by multiple nodes in a distributed system. Spark is a fast and efficient processing engine that can run on top of Hadoop or standalone.

4.1 Hadoop

Hadoop is a distributed system for managing big data collections that can grow up to thousands of nodes. As one part of the so-called Hadoop ecosystem, the MapReduce algorithm is used to process the data in a Hadoop cluster. MapReduce splits a data set into smaller pieces, which are then processed by individual nodes. The results are then recombined and returned. Hadoop is scalable and fault-tolerant, making it ideal for processing huge data sets. It is also open source, making it free to use and further develop. The fundamental features of Hadoop can be summarized in the following (White, 2015; Azarm, 2016):

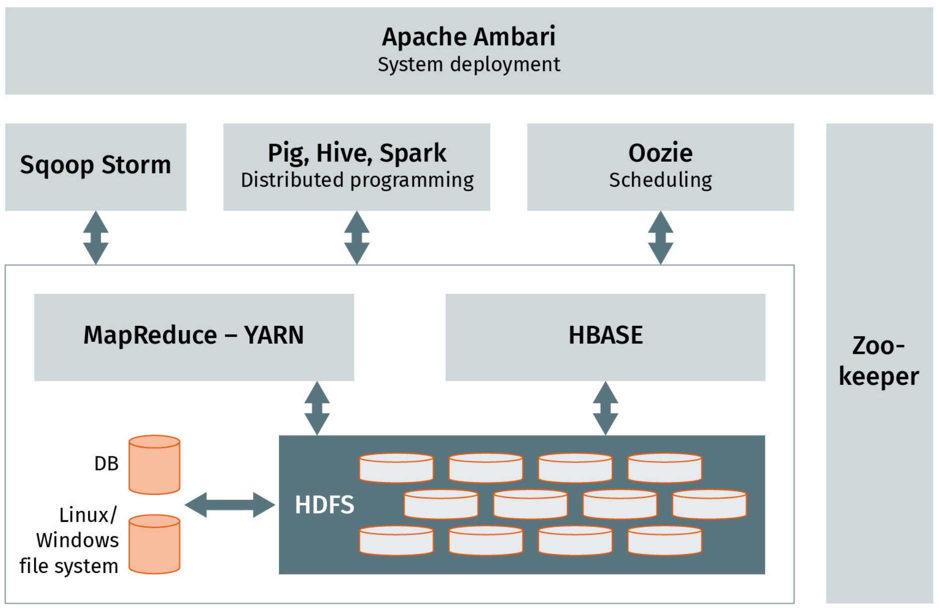
- As one part of Hadoop, the ecosystem has a very easily expandable Hadoop Distributed File System (HDFS). HDFS manages the distribution and storage of data on its various nodes. In order to increase the storage capacity, it suffices to add data nodes to the system.

- The processing codes are routed to the data, as opposed to the traditional approach where the data is loaded into the environment in which the code is executed. This strategy is most effective for large volumes of data stored on standard machines connected by standard networks. The data nodes are therefore transformed into computing nodes during the processing time. Consequently, increasing the number of data nodes increases both, the storage capacity and the processing capacity.
- The platform integrates fault tolerance mechanisms. Since Hadoop was designed to run on standard hardware, frequent outages are assumed to be unavoidable. Data is replicated across multiple nodes in order to ensure better availability and reliability of the system. When a replica disappears (following a failure), its copies are replicated again to maintain a good replication rate. Similarly, the processing tasks that are executed on the data nodes are monitored and restarted on the node of another replica if a failure occurs. This is implemented "under the hood", so that the user does not have to worry too much about fault tolerance.
- Hadoop uses the MapReduce algorithm to process data. This paradigm is suitable for retrieving and filtering data stored across data nodes, as well as performing other processing task on the data. Its integration into the Hadoop ecosystem makes it very easy to use with other Hadoop components.

The figure below shows the Hadoop software stack that is also known as the **Hadoop ecosystem**. In the following, we will present the main components of the Hadoop ecosystem.

Hadoop ecosystem
The Hadoop ecosystem is a collection of technologies that allow for distributed processing of large data sets.

Figure 23: Simplified software stack of the Hadoop ecosystem (Ayman Khalil, 2022)



The Hadoop Distributed File System (HDFS)

Commodity hardware
It is a hardware that is mass-produced and is not intended for a specific purpose

HDFS is the fundamental component. It is a distributed file system for storing and analyzing large amounts of data in a distributed environment. HDFS is part of the Hadoop ecosystem and is used by many big data applications. HDFS is a Java-based file system that runs on **commodity hardware.** It is made up of several slave nodes and one master node. The master node is in charge of the file system management, while the slave nodes are in charge of data storage and processing. HDFS stores data in blocks, and each block is replicated on multiple slave nodes. This allows HDFS to react in case of any failure (fault tolerance). This set of components and functionalities forms a middleware layer that is almost invisible to the user and which has evolved a lot between versions 1 and 2 of Hadoop. HDFS also supports data streaming, so applications can process data as it is being written to the file system. HDFS is used by many big data applications, including MapReduce, Apache Spark, and Apache Hive.

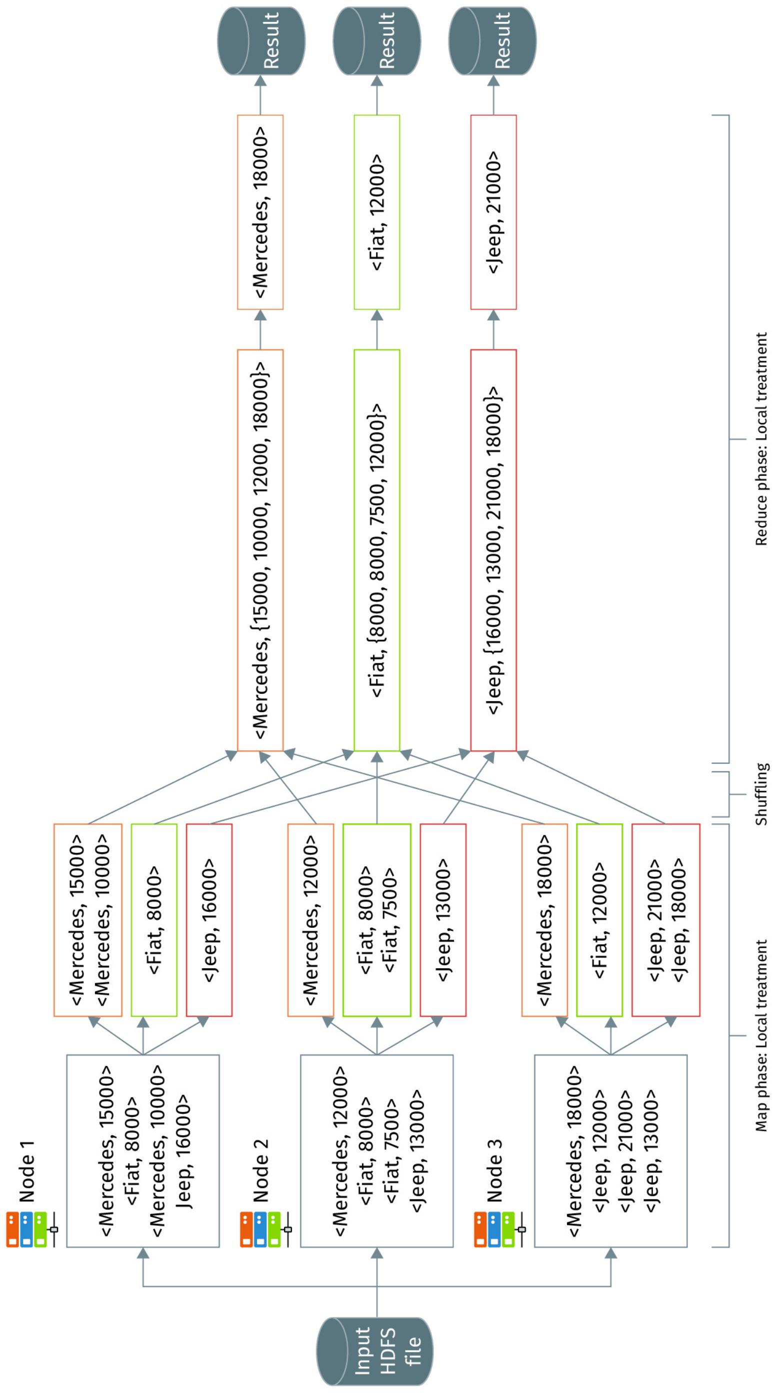
MapReduce

In a higher layer, an API allows an easy implementation of applications in the Map-Reduce paradigm. MapReduce is a programming methodology meant to process and analyze huge data sets. It is based on the map and reduce functions used in functional programming. The map function accepts a key/value pair as input and outputs a list of key/value pairs. The reduce function takes all values associated to a key and aggregates these values to a single value as output. MapReduce was developed at Google in 2004 (Dean & Ghemawat, 2004), and the open-source MapReduce implementation was released by Google in 2006 (Lammel R., 2008). MapReduce has been adopted by a number of other companies, including Yahoo!, Facebook, and Amazon. To better understand the Map-Reduce process, an example is given below. In this Python example, we execute a series of three functions calls, `getPrice()`, `map()` and `reduce()`, on a data set containing the description of four cars.

```
# importing functools for reduce()
import functools

# create the sample data
data = [ \

    {'id':1,'brand':'Mercedes','model':'CLA', 'price':15000},
    {'id':2,'brand':'Fiat','model':'500', 'price':8000},
    {'id':3,'brand':'Mercedes','model':'C300','price':10000},
    {'id':4,'brand':'Jeep','model':'Laredo','price':16000},
    {'id':1,'brand':'Mercedes','model':'CLA', 'price':12000},
    {'id':2,'brand':'Fiat','model':'500', 'price':8000},
    {'id':3,'brand':'Fiat','model':'C300','price':75000},
    {'id':4,'brand':'Jeep','model':'Laredo','price':13000},
    {'id':1,'brand':'Mercedes','model':'C350','price':18000},
    {'id':2,'brand':'Fiat','model':'500', 'price':12000},
    {'id':3,'brand’:'Jeep','model':'Limited','price':21000},
    {'id':4,'brand':'Jeep','model':'Laredo','price':18000}]
```

```
## define a function to return a car's price
def getPrice(car):
  return car['price']

## create key-value pairs, where the cars are the keys
# and the car's prices are the values
kv = map(getPrice, data)


## aggregate values (car prices) for each key (cars)
# use the maximum as the aggregation function
max_price = functools.reduce(max, kv)
print max_price
```

Figure 24: Distributed Execution of a MapReduce Treatment



Source: Ayman Khalil, 2022

The map and reduce functions constitute a MapReduce couple. The key point is the possibility of parallelizing these functions in order to calculate much faster on a machine with several cores or on a set of machines linked together. To make things clearer, let's consider that a functionM creates key-value pairs where the cars are the keys and the car prices are the values; this is the mapping part. A `functionR` takes these key-value pairs and aggregates the values (car prices) for each key (cars), for example, by taking the average of the values per key, or summing the value, or finding the maximum value; this is the reducing part.

The map function is parallelizable by nature, because the calculations are independent for each data entry, cars in our case. For instance, to map four elements …

- `value1 = functionM(element1)`
- `value2 = functionM(element2)`
- `value3 = functionM(element3)`
- `value4 = functionM(element4)`

These calculations can be performed concurrently, across distinct machines, provided that the data is copied there.

It is important to note that the mapped function must be a pure function of its function arguments, with no side effects such as changing a global variable or memorizing its former values.The system will create the set of related values for every distinct key, then the reduce function will be invoked so the key/value pairs having the same key will be treated as one single group.

Note: As shown in the figure above, the application of the function reduce has generated three results (Maximum price for each distinct car brand).

- `interMercedes = functionR(value1, value2, value3, \`
  `value4)`
- `interFiat = functionR(value1, value2, value3, value4)`
- `interJeep = functionR(value1, value2, value3, value4)`

The Hadoop ecosystem is very rich, and there are in particular higher-level applications allowing for example to process data in a formalism close to SQL (as in a relational database), such as Hive and Pig, and tools making is very easy to import external data into HDFS or export Hadoop data to external sinks (Akil, Zhou & Rohm, 2018). In the following we will present a brief description of some important applications.

YARN

YARN is an abbreviation for "Yet Another Resource Manager". To manage resources for big data applications, YARN allocates resources to different applications, manages the data flow between applications, and monitors the overall health of the system. While HDFS is the storage layer in an Hadoop system, YARN is the compute layer that is closely integrated with HDFS. Architecturally, YARN is built on top of HDFS and it used by the MapReduce programming model. If a MapReduce job is triggered to run on some nodes of the cluster, YARN makes sure that the job is running as close to the data as possible by giving the right jobs to the nodes holding the respective data blocks and also, it will allocate the appropriate resources to these nodes like CPU and RAM. YARN is a key component of the Hadoop ecosystem and is essential for running big data applications in the Hadoop ecosystem that use resources across multiple machines in a cluster.

Tez

Apache Tez is another Hadoop component that runs "under the hood" without us even noticing in most cases. It is an alternative to the MapReduce engine and, like the MapReduce engine, is built on top of YARN. It is much faster than the MapReduce engine, because it uses so-called **Directed Acyclic Graphs (DAGs)**. These simplify the sequence of mapping and reducing of the MapReduce engine to avoid unnecessary intermediate steps, unnecessary data access, and removing dependencies within the sequence. Using DAGs, Tez does not go through the MapReduce sequence step-by-step, but first evaluates the overall process. Ultimately, after an initial mapping phase, there are only reducers that follow to complete the task of concatenated MapReduce jobs. This happens for us without any configuration, as Tez is the compute engine by default from Hadoop version 2. It can be used to run Hive, Pig, and even MapReduce jobs instead of using the MapReduce engine.

Directed Acyclic Graph (DAG)
DAG in the Big Data processing context is a workflow paradigm where each task (represented as a node of the workflow) in connected to the following task via an edge that points in one direction only. This paradigm dictates that there cannot not be closed loops in the workflow but only processes moving forward in the overall workflow.

Sqoop

Scoop is a tool for efficiently transferring data between HDFS and external data sources like text files or relational database management systems (RDBMS), such as MySQL, Oracle, PostgreSQL, and Microsoft SQL Server. It can, for instance, be used to import data from a RDBMS into HDFS, or to export data from HDFS into a RDBMS. Sqoop uses so-called connectors to connect to RDBMS.

Oozie

Literally, "oozie" means "elephant keeper" in Burmese. It is a Java-based system that can be used for scheduling and managing jobs. This technology is a Hadoop component that allows us to orchestrate a sequence of actions in a cluster, such as a Sqoop import, followed by a Hive job, followed by a couple of MapReduce jobs, followed by a Pig job. These workflows constitute Directed Acyclic Graphs (DAGs) that, as we already know are quite efficient, and we can specify them in XML format.

Pig

Pig is a data processing platform that runs on top of YARN using the MapReduce or Tez engine (the latter being much faster). MapReduce jobs are very powerful, but writing a sequence of MapReduce jobs to execute a complex task is in many cases simply too hard to program. Consequently, we can use Pig to simplify the programming of complex big data operations. To do that, we can use Pig Latin, the language used to write Pig scripts. It is designed to be easy to learn and use. This is achieved by an additional layer of abstraction that hides the complexity of MapReduce from us, so that we can focus on the program logic instead.

Hive

As we know by now, Hadoop is very powerful when it comes to storing and processing big data by distributing the storage and compute across multiple nodes in a cluster. The downside to this is its complexity. Hive is a technology, just like Pig, to simplify this complexity by adding a layer of abstraction. Hive sits next to Pig and on top of MapReduce and YARN in the Hadoop ecosystem stack. It is designed so that we can write well-known SQL statements to query our data. Using Hive presents the data in HDFS as if they were stored in a relational database. Under the hood, these SQL statements are then translated into MapReduce jobs that are executed using the MapReduce or Tez engine on HDFS managed by YARN. The beauty of Hive is its simplicity coupled with the powerful data processing capabilities of Hadoop.

Spark

Until this point you might get the idea that the Hadoop ecosystem is very powerful and extensive. But would you program an anomaly detection algorithm, a Support Vector Machine or a graph analysis using Pig, Hive, or even MapReduce? Spark can be seen as a more modern alternative to this classic Hadoop stack. Using Spark we can use Java, Scala, Python, R and more languages that might be more "natural" for data analysis than Pig, Hive and MapReduce to program more complex tasks like machine learning, graph analysis, and complex data processing tasks. In the Hadoop ecosystem, Spark runs on top of YARN (or alternatives like Mesos) which in turn runs on HDFS. It is much faster than MapReduce, Hive, Pig, or Tez by using in-memory data processing and also Directed Acyclic Graphs (DAGs). It is designed for speed and efficiency, making it an ideal choice for big data applications and also all kinds of data analysis. Spark can be used for a variety of tasks, including data processing, machine learning, streaming and graph analysis. More details about Spark will be explained later.

HBase

HBase is a column-oriented NoSQL database, which means that it does not use fixed schemas, row-oriented design, or traditional SQL query language. Instead, it uses a language called HQL. Another particularity of HBase are the so-called column-families that group several columns in the database which is very effective for sparse data. HBase is a Java-based system, so it can run on any platform that supports Java. It also supports a variety of programming languages, including C++, Python, and Ruby. HBase is built on HDFS and thereby is highly scalable and reliable. Historically, it was built as an open source project on top of Google's BigTable. It allows for CRUD operations (Create, Read, Update, Delete) and auto-sharding which means that data is partitioned on-the-fly in a way so that the data is optimally distributed across nodes.

Kafka

Apache Kafka is a popular message broker that helps to manage large volumes of streaming data. It is a Pub/Sub system that decouples the producers of messages, e.g., sensors, from data consumers, e.g. applications using the data that is organized in topics. In between the producers and consumers is the Kafka cluster with various brokers. Kafka uses a so-called immutable commit log to ensure that messages are not lost in transaction. It can be used to manage data in a variety of ways, including streaming data, managing logs, and managing data pipelines in general.

Storm

Storm is designed for real-time data processing, meaning that it can handle large volumes of data quickly and reliably. It is also fault-tolerant, meaning that it can continue to operate even if individual machines in the system fail. Storm is used by a number of large companies, including Twitter, Netflix, and The Guardian. It is a powerful tool, but it can be difficult to learn and use. For this reason, it may not be suitable for all applications. Since it was acquired by Twitter, it is under an open source license. Architecturally, it is similar to MapReduce with the Tez engine. It also uses DAGs, but being designed for processing continuous streams of data instead of batch jobs, Storm DAGs run until they are actively ended. The nodes in a Storm DAG are called spouts and bolts and the edges represent the streams of data from one node to another.

ZooKeeper

As we learned, much of Hadoop's power and efficiency comes from its distributed nature of storage and compute. This is built around a master/slave architecture. But what happens if a machine goes down? Hadoop is designed to run on commodity hardware that will fail at a given time. This is where ZooKeeper comes into play. Let's say a worker node goes down. ZooKeeper recognizes this and restarts the node. ZooKeeper keeps also track of what tasks are being performed on which node. This helps in case of failure to continue an interrupted task on a new or restarted node. More severely, imagine the master node going down. In this case a worker takes over and becomes the new master. But what happens if two or three workers decide to be the new master. ZooKeeper prevents this, making sure that there is only one master in a cluster avoiding civil war in the cluster and thereby contributing to data consistency and reliability of the system.

Ambari

Ambari is management platform for Hadoop developed by Hortonworks. It provides a graphical interface for managing Hadoop clusters including an easy way to install Hadoop components. Ambari allows administrators to monitor the health of the system, such as individual nodes, the storage capacities of HDFS, the compute resource usage by YARN, and the ZooKeeper status. It also provides the ability to provision new nodes and configure the cluster's settings in a graphical user interface.

4.2 Hadoop File System (HDFS)

Presentation

HDFS (Hadoop Distributed File System) allows users to access distributed data in Hadoop clusters in a very performant way. HDFS has become an important tool for large data sets management and analytical applications.

Once HDFS collects data, the system splits it into many bricks, replicates them n times, and distributes them across several cluster nodes for parallel processing. Each piece of data is copied many times and distributed to each one of the nodes, with at least one copy stored on a separate server in the cluster. As a result, the data that has been stored on failing nodes can be accessed from other nodes in the cluster. Processing can continue despite the failure (White, 2015).

HDFS organization

HDFS is developed to support applications with large volumes of data, such as individual files that can amount to terabytes. It is based on a master/slave architecture. Each HDFS cluster is made up of machines playing different roles. We distinguish three types of nodes: the active NameNode or HDFS Master, the Secondary NameNode and the Data-Node (White, 2015).

A node called the active NameNode establishes and maintains a distribution map of all files stored in HDFS. This map is updated frequently. Meanwhile, the insertions of new files and deletions of old ones give rise to some changes in the mapping which are stored in the form of logs both in the active NameNode and in the secondary NameNode. An up-to-date map is therefore obtained by applying the evolutions described in the logs to the **meta-data** of the NameNode.

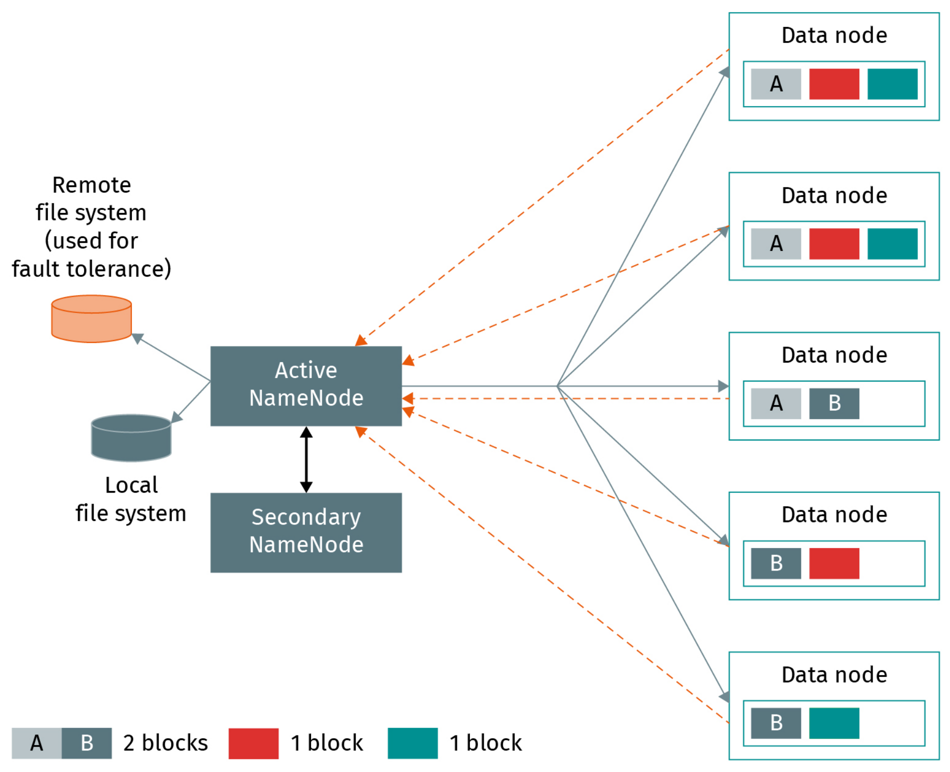
Metadata
It is data that describes other data. It can include information such as the name of the author of a document, the date it was created, or the size of a file.

The Secondary NameNode, despite its name, is not a backup of the active NameNode, but rather its helper. It is responsible for recalculating from time to time an up-to-date map of the distributed file system by applying all the logs, and then updating that of the active NameNode without influencing the latter (having been slowed down by the calculations made). It can be used to restore the active NameNode in case of failover to a certain extent. But this would cause data loss in most cases because of state differences between the two.

Each file is split into blocks, typically 64MB or 128MB, and each block is replicated n times, usually three times by default. The replicas of the same block are stored on different machines, in order to avoid data loss when nodes fail, or even two! In the case of the dis-appearance of a node and its blocks, each disappeared block is reconstituted on a new node from one of its still accessible replicas. HDFS thus quickly reconstitutes a set of n rep-licates for each block.

In the figure below, the blue file is large enough to be stored as two blocks, each replicated in three copies, while the orange and green are each composed of a single block replicated three times.

Figure 25: HDFS Master/Slave Architecture



Source: Ayman Khalil, 2022

A single DataNode can very well store several different blocks, coming from the same file or from different files, but it cannot store the same block several times. In a large Hadoop cluster, the replicas of the same block must be stored on nodes located in different **racks** (therefore electrically independent).

Rack
It is a metal frame that supports computer equipment.

During the creation of a new file, the active NameNode will distribute the replicas of its blocks on the different DataNodes available (*blue arrows in the figure above*), and each DataNode will keep the active NameNode informed of its state, and the success or the failure of its block creations (*orange dashed arrows in the figure above*). The active NameNode will thus maintain an up-to-date knowledge of the HDFS file system.

HDFS Fault Tolerance and High Availability

Data stored on HDFS is replicated but the active NameNode constitutes a **Single Point Of Failure** (SPOF) of the system. Without it, the data is still stored and replicated on the DataNodes, but inaccessible due to lack of mapping. A failure of the active NameNode could therefore render the HDFS file system completely unusable.

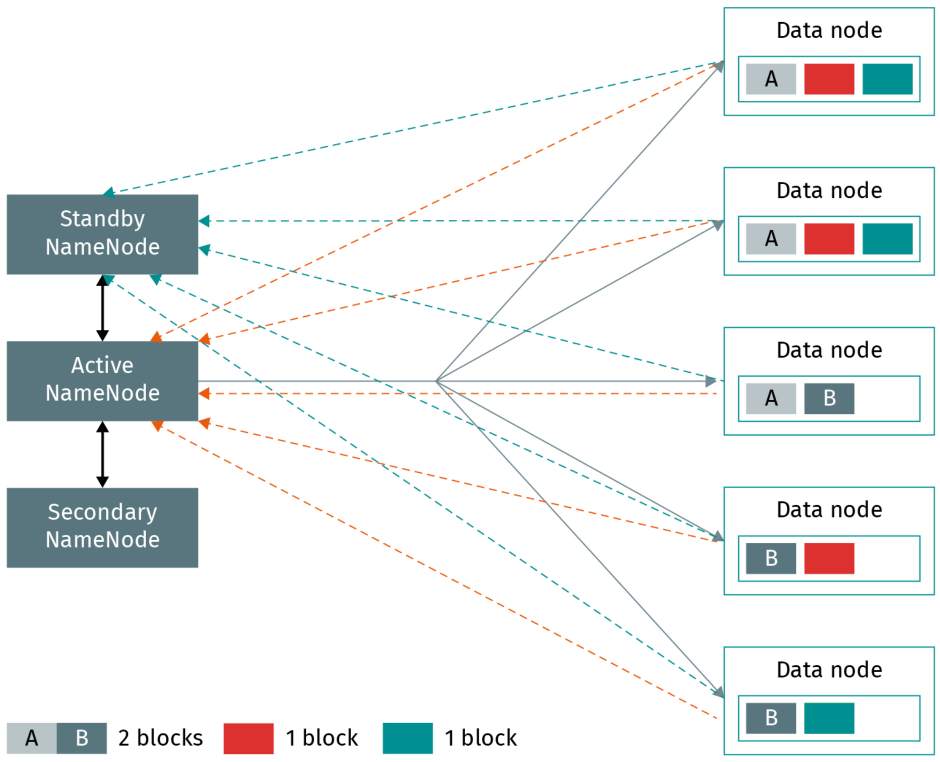
Single Point Of Failure
A system is said to have a single point of failure if the failure of a single component of the system results in the failure of the entire system.

To overcome this weakness, two complementary fault tolerance mechanisms then high availability were introduced (Azarmi, 2016; Kleppmann, 2017):

Fault tolerance: The metadata of the active NameNode are regularly saved on a local file system (fast access) but also on a remote system. In the case of an incident on the active NameNode, it will thus be possible to reconstitute the Hadoop file system from a remote copy of its metadata and continue to operate its data nodes. You might ask if the Secondary NameNode cannot be used for this. This is not entirely the case. The Secondary NameNode periodically copies and processes the logs and metadata from the active NameNode. But as the state of the active NameNode changes continuously, data loss is highly likely in the case of recovery from the Secondary NameNode.

High availability: It is possible to duplicate the active NameNode, and create a standby NameNode which permanently receives and stores the same metadata and logs as the active NameNode (*see figure below*). The standby NameNode is therefore also awake but does not act on the data nodes.

Figure 26: High availability solution for HDFS in the case of a failure on the NameNode



Source: Ayman Khalil, 2022

On the other hand, it is ready to replace and become the active NameNode at any time and almost without delay, making the failure almost imperceptible. Obviously, this strategy requires an additional machine.

HDFS Access Mechanisms

Hadoop offers a full Java API for accessing HDFS files. It is based on two main classes:

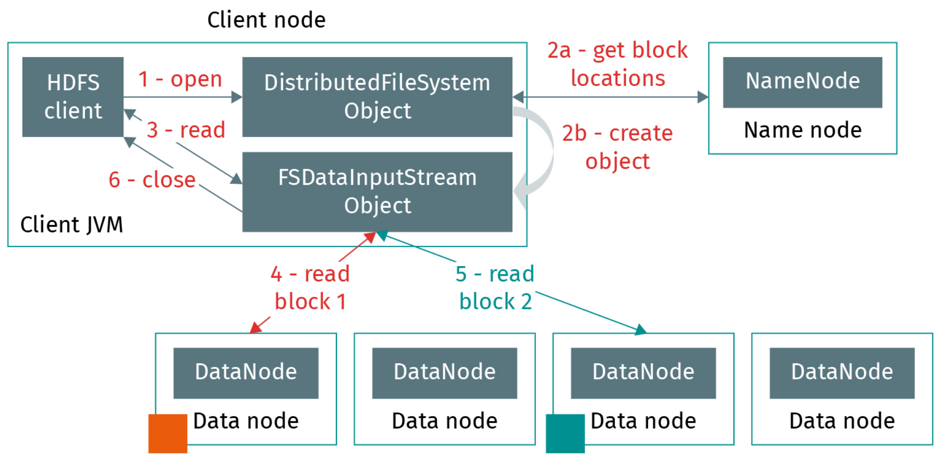
a) *FileSystem* represents the file tree (file system). This class allows copying local files to HDFS (and vice versa), renaming, creating and deleting files and folders
b) *FileStatus* manages the information of a file or folder:
   - Its size with *getLen*(),
   - Its nature with *isDirectory*() and *isFile*(),

These two classes need to know about the configuration of the HDFS cluster, using the class *Configuration*. On the other hand, full file names are represented by the Path class.

HDFS Reading Mechanism

Reading a file in HDFS is quite simple, and can be summed up in the figure below.

Figure 27: HDFS Reading mechanism



Source: Ayman Khalil, 2022

The client code starts by creating a local object of the DistributedFileSystem class, which will

act as a **stub** or proxy with the HDFS file system. The client code therefore addresses this local object and asks to open a file (*step 1, open command*).

Stub
It is an object or short piece of substitute code that holds predefined data or functionality and uses it to answer calls during tests instead of using the more complex system it represents.

The stub then talks to the NameNode of HDFS to find out the location of the replicas of all the blocks in the file (*step 2a*). Then the stub creates another local object, of the FSDataInputStream class (*step 2b*), which will act as a reader specialized in reading the target file, knowing the nodes to contact.

The client will then perform read operations on this specialized reader (*step 3*), which will interrogate one of the nodes storing the first block of the file (*step 4*). Once the first block has been read, if the read operations continue on the part of the client, the specialized reader will interrogate a node containing the second block (*step 5*) and so on.

Finally, the client will ask to close the file opened for reading with the specialized reader (*step 6, close operation*). Note that the specialized reader verifies the integrity of the data read (checksum calculations) and signals any anomaly to the stub, which retransmits them to the NameNode.

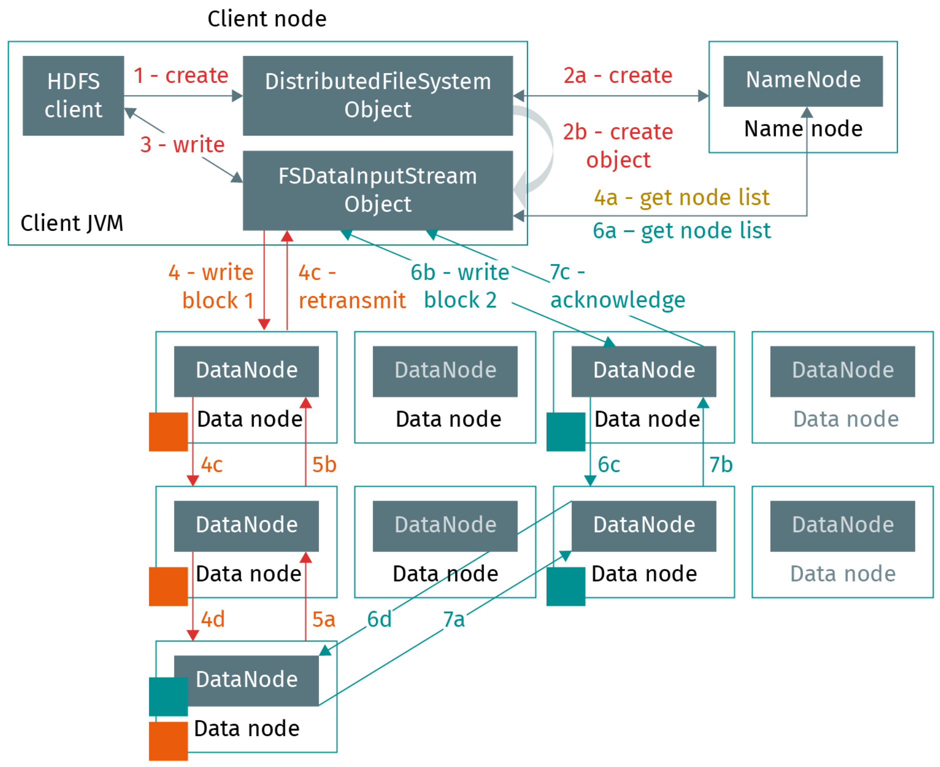
HDFS writing Mechanism

Writing a file in HDFS is more complex than reading it. At first, the client creates a local DistributedFileSystem object that acts as a stub or proxy to the HDFS filesystem, as was already the case for a read operation.

The client can ask the stub to create a new HDFS file (*see figure below step 1, create operation*). The stub then addresses the NameNode of HDFS to obtain the right to create such a file (*step 2a*), and to be able to locally create an FSDataOutputStream object (*step 2b*) which will play the role of a writer specialized in writing of the target file. Subsequently, the client speaks locally to this specialized writer to ask it to write data to this file (*step 3, write operation*).

The writer then dialogues with the NameNode of HDFS to know where to create a first block and its replicas (*step 4a*), then begins to write the first replica of the first block to a data node (*step 4b*). A pipeline mechanism is then set up: the node of the first replica retransmits its data to the node chosen to host the second replica (*step 4c, ack = acknowledges*), which itself retransmits it to the node chosen to host the third replica (*step 4d*), and the process continues if more than three replicates per block are specified.

Figure 28: Writing new data in HDFS



Source: Ayman Khalil, 2022

When the writing on the last replica is finished, an acknowledgment message goes up to the node of the previous replica (*step 5a*), and so on until reaching the node of the first replica (*steps 5b*). This node then returns an acknowledge to the specialized writer (*step 5c*), and if everything went well, the writer will proceed by processing the next request to write data. When the first block is full, the writer again asks the HDFS NameNode for a set of nodes to write a second block and its replicates to (*step 6a*), and the pipelined write process repeats itself (*steps 6b to 6d, then 7a to 7c, ack = acknowledges*).

After all data writes are complete, the client asks the writer to close the new file then informs the stub, which in turn informs the HDFS NameNode, which updates its metadata with a new file in its mapping.

HDFS hands-on with Python

Luckily, in everyday practical work, we seldomly have to worry too much about this complexity. In the following example, we see how easy it can be to work with the powerful HDFS. In this example, we use Python to work with HDFS as the file store.

```
# import modules and connect to HDFS
# (twitter in this case)
from hdfs.hfile import Hfile
hostname = 'hadoop.twitter.com'
port = 8020
hdfs_path = '/user/travis/example'
local_path = '/etc/motd'
```

First, we open the local and the HDFS files.

```
hfile = Hfile(hostname, port, hdfs_path, mode='w')
fh = open(local_path)
```

We then copying the content of a local file into HDFS, line by line.

```
for line in fh:
  hfile.write(line)
```

Finally, we close the local file and the HDFS file.

```
fh.close()
hfile.close()
```

Our data resides in distributed HDFS and is ready to be processed in parallel.

Now, we can read the file from HDFS.

```
# specifying the file that we want to read

hfile = Hfile(hostname, port, hdfs_path)

# reading the file
```

```
data_read_from_hdfs = hfile.read()
print data_read_from_hdfs
# closing the file
hfile.close()
```

4.3 Spark

Conceptual Background

Scalability
It is the ability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged to accommodate that growth.

Distributed systems are a key part of big data technologies. In a distributed system, data is spread across multiple nodes, allowing for faster processing and improved **scalability**. Spark is a distributed system for processing big data. It is designed to be fast and efficient, and to handle a large number of tasks simultaneously. Spark can be used for a variety of purposes, including data analysis, machine learning, and streaming data (Chambers & Zaharia, 2018).

In Spark we can write complex processes that consist of several MapReduce phases. This can be done also with Yarn. However, there are a few key differences between Spark and Yarn that are worth mentioning. First, Spark is a standalone program that does not require Yarn, while Yarn is a cluster management tool that works with Spark. Second, Spark is much faster than Yarn when starting up, because it does not need to initialize the entire cluster. Moreover, Spark has a more user-friendly interface than Yarn. The processes can be written in different languages namely Scala, Java and Python. We will take an example based on Python for its pedagogical simplicity.

In this section, we will discuss the basics of Spark, including its architecture, features, and benefits. We will also see how we get started with Spark and how to use it for data analysis and machine learning.

How Spark works

Spark is designed to provide high performance for data processing tasks. It also includes a number of built-in libraries that can be used for data analysis. Spark is considered a parallel data programming API. It is based on the concept of Resilient Distributed Datasets (RDDs). An RDD is a fault-tolerant collection of elements that can be operated on in parallel. RDDs are created by splitting a dataset into partitions, which are then distributed across the nodes in the cluster. RDDs can be transformed and manipulated in various ways, and the results are also distributed across the nodes. This allows Spark to scale up to large data sets while still providing a high level of performance (Chambers & Zaharia, 2018).

RDDs can be cached in memory to improve performance. When an RDD is cached, Spark caches the data in memory, so that it can be accessed quickly without having to go through the network, every time the data is accessed. RDDs can be transformed and manipulated in many ways, including filtering, mapping, and reducing. RDDs are immutable, which means that once they are created, they cannot be changed. This ensures that the data is always consistent and eliminates the need for locks.

The Spark API provides two ways to operate on RDDs: **Transformations** that are operations that create a new RDD from an existing RDD; and **Actions** that are operations that return a result to the caller.

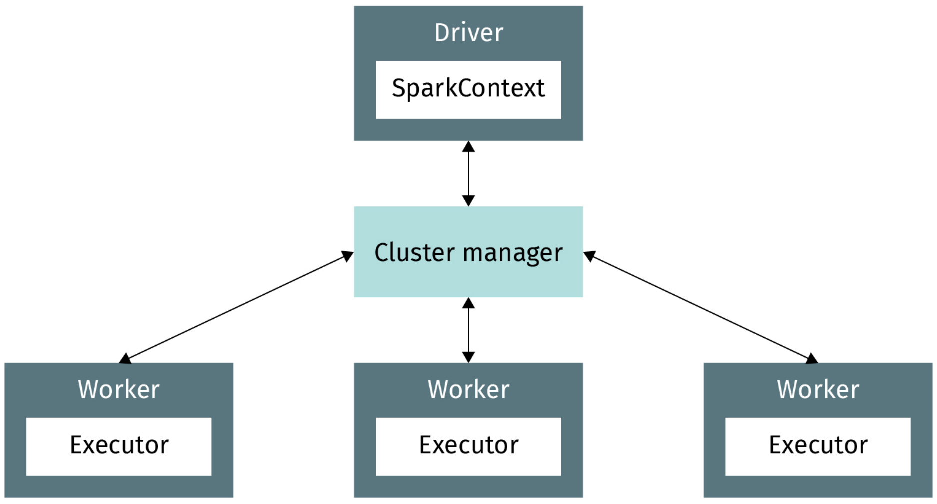
The following Python example shows how to use the Spark API to calculate the sum of the elements in an RDD (the values between <> are placeholders; more details about the Spark API will be presented later).

```
from pyspark import SparkContext
SparkContext().parallelize(<data>).\
    map(<mapping_function>).\
    reduce(<reducing_function>).\
    collect()
```

Architecture of Spark

Spark is a distributed system that runs on a cluster of machines. The Spark cluster manager distributes the workload across the machines in the cluster. The Spark driver program is responsible for launching the Spark cluster, and for submitting jobs to the Spark cluster (Karau, Konwinski, Wendell & Zaharia, 2015). The driver program also manages the interaction between the Spark cluster and the user. The Spark executor program is responsible for running the tasks that are submitted to the Spark cluster. The Spark worker program is responsible for processing the data that are assigned to it by the Spark executor program. Spark uses a master/slave architecture. The master node is responsible for managing the Spark cluster, and for distributing the workload across the slave nodes. The slave nodes are responsible for running the Spark executor programs, and for processing the data that are assigned to them by the master node.

Figure 29: Spark Architecture



Source: Ayman Khalil, 2022

Components of Spark

The key Spark components are: Spark core, Spark SQL, Spark Streaming, MLLib, and GraphX (Damji, et. al, 2020).

Spark Core

Spark Core provides the basic functionality of Spark, including data parallelism, scheduling, and memory management. There are APIs for Scala, SQL, Python, Java and R. These APIs provide a comprehensive interface to Spark. They control Spark from the respective runtime, e.g., Python or R, including starting and stopping Spark, loading and manipulating data.

SQL

Spark SQL enables users to interact with data stored in Spark using standard SQL commands. It includes support for reading and writing data in **Parquet format** and also as DataFrames and Datasets.

Parquet format
It is a columnar storage format that is used by many big data systems. It offers several advantages over other formats, including the efficiency for reading and writing data.

Streaming

Spark Streaming enables users to process continuous data streams in near real-time. It includes support for processing data in mini-batches and processing data as it arrives. Spark streaming also includes support for processing data with multiple processors.

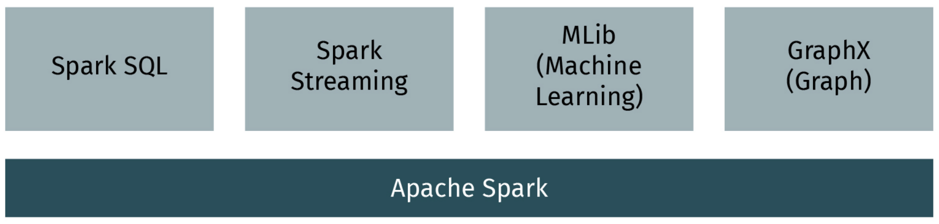
MLlib:

MLlib is a library of machine learning algorithms for use with Spark. It includes support for linear regression, logistic regression, clustering, and much more.

GraphX

GraphX is a library for manipulating graphs and performing graph analytics in Spark. It supports traversing graphs, finding shortest paths, and more.

Figure 30: Spark Components



Source: Damji, et. al, 2020

Advantages over other big data processing systems

Spark is a distributed system that has a number of features and advantages over other systems.

- Spark can be run on a single machine or on a cluster of machines, and it can process data in memory or on disk.
- Spark also has a number of built-in libraries for data processing, including libraries for machine learning, graph processing, and streaming data.
- Spark was created by the team at UC Berkeley that also created Hadoop, and it is designed to be compatible with Hadoop.
- Spark can read and write data in HDFS, and it can run on the same clusters as Hadoop.
- Spark is also designed for performance. It can process data faster than Hadoop MapReduce, and it can use more memory than Hadoop.

How Spark can be used in Python

Spark can be used in Python for data analysis, machine learning, and streaming applications. It can run on clusters of computers or on a single computer. In Python, Spark can be used with the PySpark module. PySpark provides a Python interface to Spark and allows you to run Spark jobs on a cluster of computers. PySpark also includes the Spark SQL and the MLlib modules.

Steps to install Spark

1. Download Spark from the Spark website and install it on your local machine (the following steps only apply for this option). Alternatively, use a preconfigured image for a virtual machine or a Docker container. Another straightforward way to install Spark locally is to use the `sparklyr` package for R which has a convenience function, `spark_install()`.
2. Extract the files to a location on your computer.
3. Run the Spark executable file to start the **Spark shell**.
4. Use the Spark shell to create a new Spark application.
5. Add the required libraries to your application.
6. Run your application on a cluster.

Spark shell
It is an interactive tool that allows you to run Spark jobs. It can be used to submit jobs, inspect the results, and explore the data. The Spark shell is started by running the following command:
$ spark-shell

Using Python to interact with Spark

A pySpark program should start with this:

```
from pyspark import SparkConf, SparkContext
name = "test1"
config = SparkConf().setAppName(name)
sc = SparkContext(conf=config)
```

sc represents the Spark context. It is an object that has several methods including those that create RDDs.

An RDD is an abstract collection of data, resulting from the transformation of another RDD or creation from existing data. An RDD is distributed, i.e. distributed over several machines in order to parallelize the processing.

You can create an RDD in two ways:

1. Parallelize a collection
   If your program contains iterable data (array, list. . .), it can become an RDD.

   ```
   data = ['one', 'two', 'three', 'four']
   RDD = sc.parallelize(data)
   ```

   It is called a “parallelized collection”.
2. Spark can use many data sources
   For example, data can be read from HDFS, Hbase, etc. and in many file formats, e.g., text and Hadoop formats such as **SequenceFile**.
   Here's how to read a simple text or CSV file into a RDD (in this case, the file is stored in HDFS, but for testing purposes, you can also load a text file from your local disk).

   ```
   RDD = sc.textFile("hdfs:/share/data.txt")
   ```

As with MapReduce, each line of the file constitutes a record. The transformations applied on the RDD will process each row separately. The lines of the file are distributed to different machines for parallel processing.

Some Spark processing uses the concept of pairs (key, value). The keys allow for example to classify values in a certain order. To efficiently store this kind of RDD, we can use a so-called SequenceFile.

The following function reads the pairs from a SequenceFile stored in HDFS and creates an RDD.

```
RDD = sc.sequenceFile("hdfs:/share/data1.seq")
```

The following method saves the (key, value) pairs of the RDD to a file system (HDFS in this case).

```
RDD.saveAsSequenceFile("hdfs:/share/data2.seq")
```

Actions

Actions are methods that are applied to an RDD to return a value or a collection.

```
# return the number of elements in an RDD
count = RDD.count()

# return the RDD as a Python list
list = RDD.collect()

# return the first element of the RDD
first = RDD.first()

# return the first n elements of the RDD
first =RDD.take(n)

# apply an aggregation function of the type fn(a,b)-> c
result = RDD.reduce(<function>)
```

where `<function>` is the aggregation function. It can be passed as an argument or it can be the lambda function that defines the aggregation.

Note that the functions that return a Python list instead of an RDD are to be used with care. Imagine that you work with massive amounts of distributed data. The command `RDD.collect()` will probably crash your session as all distributed data will be imported into the memory of the current session as a Python list.

Transformations

RDDs have several methods that resemble Python functions, e.g. map, filter, etc. In plain Python, map is a function whose first parameter is a lambda or the name of a function, the second parameter is the collection to process. For example, to multiply each element of a list by 2, we can execute the following map function.

```
list = [1,2,3,4]
double= map(lambda n: n*2, list)
```

In pySpark, map is a method of the RDD class, its only parameter is a lambda or the name of a function:

```
list = sc.parallelize([1,2,3,4])
double = list.map(lambda n: n*2)
```

In the latter case, double is an RDD.

The following transformations handle RDDs whose elements are pairs (key, value).

```
# return an RDD whose elements are pairs
# (key, list of values having this key)

RDD.groupByKey()

# return an RDD whose keys are sorted (set True or False)
RDD.sortByKey(ascending=True)

# group the values having the same key and
# apply the function (a,b) -> c (string concatenation
# in this case using hyphens as separators)
RDD = sc.parallelize ([ \
    (1, "Tom"), \
    (2, "Claude"), \
    (1, "Chris"), \
    (2, "mary"), \
    (1, "Victor") \
    ])
print RDD.reduceByKey(lambda a,b: a+"-"+b).collect()

# console output:
# [(1, "Tom-Chris-Victor"), (2, "Claude-mary")]
```

To launch the entire Python script in a Spark context, execute the following command on a command line.

```
spark-submit test1.py
```

4.4 DASK

Conceptual background

Pandas and Scikit-learn are popular data science libraries for Python. They are both designed to work with data stored in memory. This can be a limitation when working with large datasets that do not fit in memory. Dask is a Python library for working with large datasets. It is a distributed system that can scale to hundreds of processors. Dask can work with data stored in memory or on disk. It is a powerful tool for data science.

Pandas and Scikit-learn
Pandas is a robust data analysis tool that makes working with enormous datasets simple. Scikit-learn is a machine learning library that includes a variety of data mining and predictive modeling algorithms.

Dask is a distributed computing system that helps you analyze and process large data sets. It is composed of a number of individual "workers" that can be spread across many machines, allowing you to scale out your processing power.

How Does It Work? Dask connects to a number of different **back-end** systems, such as Hadoop, Spark, or Pandas. This allows it to work with a wide variety of data formats and storage solutions. Dask then uses a task-based programming model to allow you to easily create parallelized processing pipelines.

Back-end system
It is a computer system that is used to store and manage data. Back-end systems are typically used to support front-end systems, which are used to interact with users.

Why Use Dask? Dask offers several advantages over traditional distributed systems. First, it is very easy to use, even for non-experts. Second, it provides an intuitive "dataframe"-style API that makes working with data much easier. Finally, it scales out very well, allowing you to process large data sets on many machines simultaneously.

Parallel computing

Dask is a library for parallel computing in Python. It enables you to break up a problem into smaller chunks that can be computed in parallel. Dask can be used to parallelize both CPU- and **GPU-based computations**. One of the key benefits of using Dask is that it provides a uniform interface for parallelizing a variety of computations, regardless of the underlying hardware. This makes it easy to switch between different types of hardware, or to move a computation from one machine to another. Dask also provides several features for managing distributed systems, including automatic load balancing, fault tolerance, and job scheduling (Daniel, 2019).

GPU-based computation
GPUs are well-suited for the types of computations required for big data applications, such as matrix operations, convolutions, and sorting. In addition, GPUs can exploit the parallelism of many-core processors to accelerate big data applications.

Dask is built on top of two libraries:

- Dask.distributed: This library provides the basic infrastructure for distributed computing. It handles communication between nodes, scheduling of tasks, and fault tolerance.
- Dask.array: This library provides an API for parallel computing that is similar to **NumPy**. It allows you to create arrays that can be divided into chunks and processed in parallel. Dask can be used to parallelize any code that can be run in NumPy.

NumPy
It is a library for scientific computing that provides efficient, high-performance operations on arrays of data. NumPy arrays are similar to Python lists.

For example, the following code can be run in parallel using Dask.

```
import Dask.array as da
a = da.ones((10, 10))
print(a)
```

This code will create an array of 10x10 zeros. You can run it on a single machine by using the Dask command line tool.

```
Dask array --nthreads 4 ./zeros.py
```

This will use four cores on your machine to create the array. You can also run it on a cluster of machines by using the Dask-cluster command line tool.

```
Dask-cluster --nthreads 4 ./zeros.py
```

Dask cluster

A Dask cluster is a collection of machines, usually connected through a network, that can be used to run Dask applications. Dask can be installed on any machine and can use any number of cores. However, to get the best performance, it is recommended to install Dask on a machine with many cores and large amounts of memory. When creating a Dask cluster, the number of workers and the number of cores per worker should be specified (Daniel, 2019). For example, a Dask cluster with 4 workers and 8 cores per worker would have 32 cores in total.

How to create a Dask cluster

Dask clusters can be created in several ways:

- Manually create a Dask cluster using the Dask-cluster command line tool.
- Use the Cloud Manager to create and manage Dask clusters on Google Cloud Platform, Amazon Web Services, or Microsoft Azure.
- Use the Batch Scheduler to create and manage Dask clusters on Kubernetes.

How to execute tasks in a paralyzed manner

In a distributed system, tasks can be executed in a paralyzed manner. It enables you to execute tasks in a paralyzed manner, meaning that you can continue to work on your local machine while the task is executed on a remote machine.

Python example on how to paralyze data

```
import dask from dask.distributed
import Client clients = [Client("127.0.0.1:8786")
for i in range(3)] data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
parallel_map(lambda x: x + 1, data) # map function across all workers
results = [x + 1 for x in data] # results will be a list on the workers
```

Example –parallel processing of a large dataset

The following example shows how to use Dask to process a large dataset in parallel. The example uses the Titanic dataset, which is available on **Kaggle**. The titanic dataset is a large collection of data points about passengers on the titanic. We will use a random forest model to predict whether or not a passenger survived. The first step is to import the necessary libraries.

Kaggle
Kaggle is a platform for data scientists to share their work, learn from others, and compete in predictive modeling competitions

```
import dask
import pandas as pd
import numpy as np import sklearn
```

Next, we will read in the data.

```
train = pd.read_csv("titanic/train.csv")
test = pd.read_csv("titanic/test.csv")
```

We will then split the data into training and testing sets.

```
X_train, y_train = train[:,:], train[:,1]
X_test, y_test = test[:,:], test[:,1]
```

Now we can create our model.

```
model = sklearn.random_forest(criterion='gini', n_estimators=100)
```

We can then parallelize the computation of the model using Dask.

```
dask_model = dask.distributed.Client()
dask_model.parallel(n_workers=4)
dask_model.compute(X_train, y_train)
dask_model.compute(X_test, y_test)
```

SUMMARY

UNIT 5

STREAMING FRAMEWORKS

STUDY GOALS

On completion of this unit, you will be able to ...

– explain setups of streaming frameworks.
– distinguish relevant factors for use cases of each.
– identify which streaming framework suits which use case.
– apply code to for a stream processing system that is up and running.

5. STREAMING FRAMEWORKS

Case Study

Imagine that, *GuiltyPleasure*, a company producing ice cream, uses sensors to monitor the temperature of its machines. This temperature must remain permanently within a certain target range. The temperature is measured every second and short term fluctuations in the temperature are common. But for the production cycle, these fluctuations are not that relevant, but rather the average temperature in the last two minutes is calculated for every second. If the temperature in this sliding window falls outside the defined range, the complete production will be affected. So, before this happens, an employee or even an automated process must intervene to regulate the temperature. These temperature changes may occur within a very short time and require direct processing of sensor data to capture this time-critical process. It must therefore be possible to analyze the data and output a signal in near real-time if possible. Streaming frameworks are designed to be able to intercept exactly such time-critical data, so *GuiltyPleasure* assigns you with the task to find out if a streaming framework is appropriate for their use case.

Streaming frameworks refer to the processing of data streams. These are characterized by the fact that there is a continuous flow of incoming data in a specific temporal sequence. Data streams can also be described as a theoretically infinite process of incoming data elements.

Many companies produce vast amounts of data, be it sensor data from production machines, transaction data, data from user activities within a system. There is an almost infinite list of examples involving the processing of such a volume of data. But what do they work and what makes them special in contrast to other data processing frameworks?

There are different possibilities in the processing of data: One is processing in batches, which means that a certain time window or file size is given in which the data are collected before they are processed. This bears the risk, as seen in the *GuiltyPleasure* example above, that they may already be outdated at the time of processing.

Furthermore, reality has shown, that the time or file size boundaries that define which data entry should be processed in which batch are artificial, in many cases. Often, the to be processed data are not bound, as they are not limited to a certain time frame or amount, but, in most processes, data is produced continuously. Therefore, it came to mind that the batches need to be sliced into smaller time frames, eventually processing the input right when it arrives, therefore a continuous stream of data is processed. (Kleppmann, 2017, S. 14)

This near real-time analysis requires certain processing abilities which are presented in many different available stream processing frameworks, each with its own strengths and weaknesses. Some of the more popular frameworks include Apache Spark Streaming, and Apache Kafka which will be introduced further in the next sections.

5.1 Spark Streaming

Before looking at Apache Streaming, it's worth looking at the Spark Environment in general.

Apache Spark

Apache Spark describes itself as a "multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters" (Apache Software Foundation, 2022a, S. 1). It contains functions that let you import data from various sources, with important file formats and systems being supported and provides mechanisms for processing the data such as in-memory computing and the generation of key-value pairs. It can do this on a cluster with a single machine, such as a desktop environment for development purposes, as well as on clusters that can scale as large as it is needed with hundreds of machines running in the cloud. These factors, namely the in-memory processing and distributing the data on clusters is what makes it possible to process big data and what makes Apache Spark incredibly fast.

Regardless of the cluster size, the same code can be used making it straightforward to scale an application in Spark. For this reason, the Spark Framework is ideally suited for horizontal scaling with ease. If more data needs to be processed, only more hardware is required, no new code needs to be written.

Since Spark Streaming can be seamlessly integrated with Spark, these same benefits can be leveraged for a continuous stream of data. This enables both, high throughput and scalability. Unlike Spark, Spark Streaming is designed for near-real time analysis. Instead of large batch jobs, micro-batches are processed mimicking the continuous stream of data. In the following section, we will focus on the Spark Streaming module and its main aspects.

Features of Spark Streaming

Spark Streaming is a library, which is used for processing data in near real-time. Spark Streaming was the first stream-processing framework based on the distributed processing capabilities of the Spark Core Engine. The idea behind Spark Streaming can be simplified to this: Use the capabilities of Spark Core to process streaming data by transforming it into discrete collections of data that Spark can process. You can use the same Spark tools for both, stream processing with Spark Streaming and batch processing with Spark Core. Spark Streaming allows for multiple data sources to be connected and continuously processes the incoming data before storing the data, e.g., in a Hadoop Distributed File System (HDFS). Spark Streaming basically uses the same functions as the Spark Core, but adds another abstraction, the Discretized Stream, or DStream, which is a programming model to operate on the data present in the stream (Garillot, van Maasakkers, & Maas, 2019, S. 14).

Resilient Distributed Datasets (RDDs) and DStreams

In order to address the Spark Streaming Module, it is first necessary to review the underlying programming abstraction in Spark itself: The Resilient Distributed Dataset or RDD. All operations in Spark are performed on these **in-memory** objects. RDDs are collections, where individual entities in the collection can be anything. These individual entities are called Rows or Records of the collection. All of these, as said, are held in-memory which makes processing much faster than first having to read the data from disk.

In-memory
An in-memory database stores data not on regular hard disk storage, but directly in RAM.

This is the fundamental characteristic of an RDD.

But what does this term RDD mean specifically?

- **Resilient**: fault tolerant missing or defective partitions can be recovered as they are redundant in the cluster.
- **Distributed**: Data is distributed in partitions across different nodes in a cluster enabling easy parallelization of jobs.
- **Dataset**: is a set of partitioned data.

Furthermore, the following characteristics should be noted in connection with RDDs:

- **Partitioned**: Split across data nodes in a cluster
- **Immutable**: RDDs, once created, cannot be changed

An RDD in Spark can be considered analogous to a collection object in Java. Such a collection object can be assigned to a variable so that methods can be called upon it. The methods that can be called for an RDD either retrieve a result or retrieve a subset of the entities in an RDD and output them to a screen, or assign them to another RDD. Individual entities in an RDD can also be transformed by mutating them and then obtaining a resulting RDD with the mutated entities.

To make this more tangible, let us take a look at how this works practically using Python as an example. The **PySpark** module can be used to work with Spark in Python at ease. In the following PySpark example, Spark code is shown in Python to create an RDD from data that exists in a text file.

PySpark
This refers to an interface in Python that enables you to write Spark applications with Python APIs. PySpark supports many Spark functions like Spark Core, DataFrame, Spark SQL, Streaming, and MLLib. (Apache Software Foundation, 2022d)

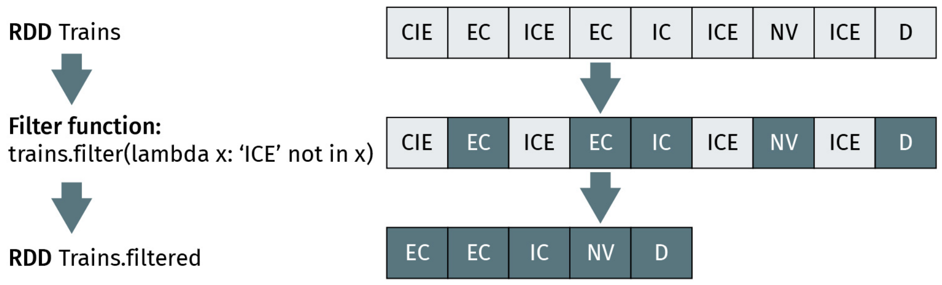
```
sc = SparkContext('local', 'MyFirstSparklingExample')
trains = sc.textFile(trainData)
```

SC stands for Spark-Context, which is basically the connection from inside a program to the external Spark world. The trains RDD will be a collection of train data and can be visualized in an array form like in the graphic below. As we can see, the inner workings of an RDD are abstracted and hidden from us to make it very easy to work with distributed data in Spark: We simply load the data into an RDD, that’s it. Should one be interested in only a subset of the train data, filters can be applied to the RDD.**:**

```
trainsFiltered = trains.filter(lambda x: ‘ICE’ not in x)
```

This filter takes a lambda expression that returns true or false. If the return value of the method is true when applied to an entity of this collection, this entity is present in the resulting RDD. In the opposite case this entity is omitted from the resulting RDD. In our example, we filter for trains that are not Intercity Express (ICEs).

Figure 31: Filter Function for an RDD


RDD Trains
CIE EC ICE EC IC ICE NV ICE D
Filter function:
trains.filter(lambda x: 'ICE' not in x)
CIE EC ICE EC IC ICE NV ICE D
RDD Trains.filtered
EC EC IC NV D

Source: von Bargen, 2022

To print the data from an RDD the method collect can be used.

```
trains.collect()
```

This function should be used with care, though, because it will translate the data in the RDD to a regular Python object. Accordingly, if we want to harvest the distributed compute power of Spark, we should not continue working with collected data. Depending on the size and number of elements within the RDD, it might be reasonable to use the take method instead which will print the first 3 entities within the RDD.

```
 trains.take(3
```

The Spark module makes it possible to work with streaming data by providing a high - level abstraction for these data, called discretized streams or DStreams. They are somewhat similar to the RDDs, but add another level of abstraction making it even easier for us to work with streaming data. Let us explain the concept of DStreams with a concrete example. Namely, the streaming of log messages from an important website that is to be monitored. These messages can be available on a socket that a monitoring tool is retrieving the data from or stored in a directory where new files are constantly added to this directory. We fetch the log messages from this directory. Log files usually consist of a large number of text logs containing **,** the time the log arrived and the information about what happened in the application. These files can be reorganized and displayed as a stream where each message represents an RDD entity.

This stream of entities is called a discretized stream and is represented by a particular class called DStreams. Therefore, a DStream equals a sequence of RDDs. The advantage that comes with the abstraction of the stream of RDDs to a DStream is that we are able to perform operations on an entire stream of data at once instead of executing the same command over and over again on each individual RDD entry. For example, imagine that in the website logging the time was not set right so that there is a time shift between the correct time and the logging time in the file of three hours. Using DStreams, we can simply add three hours to the logging time and this operation will be executed for all incoming RDDs for us under the hood. DStreams exists regardless of the used programming language, be it Java, Scala, or Python. All of the data is organized into RDDs and sequences of these form the DStream. So now when operations are performed on DStreams, they are actually performed on all individual RDDs within the DStream. Every DStream has a batch interval associated to it, because remember that actually, Spark Streaming performs batch processing. But to handle streams of data, the batches are very small so that they seem continuous. (Apache Software Foundation, 2022c). Now for example we can use the filter function on a DStream as well.

```
sc = SparkContext('local', 'MyFirstSparklingExample')
ssc = StreamingContext(sc, '1')
myDStream = ssc.textFileStream('./data_directory/')
myTrains = myDStream.filter(lambda x: 'ICE’ not in x)
```

Data Sources and Sinks

Spark Streaming provides built-in support for a range of streaming data sources and sinks (such as files and Kafka), as well as programmatic interfaces that let you specify any data writer.

In Spark Streaming, a basic distinction can be made between two categories of sources:

- Basic Sources: This type of sources is available firsthand in the StreamingContext API. These can be file systems or socket connections.
- Advanced Sources: This type of sources relates to systems like Apache Kafka or Kinesis, which must be actively linked.

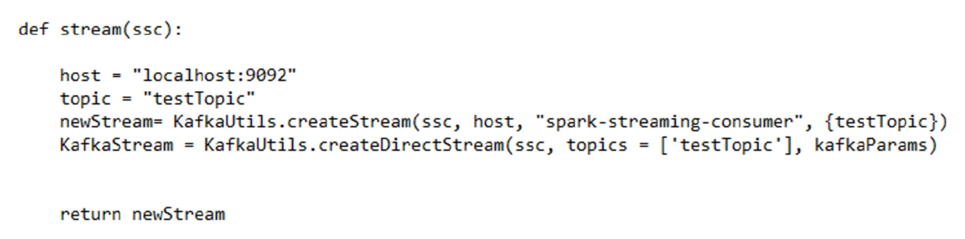
Because Spark Streaming is a widely used application, a variety of open sources and proprietary sources can be connected. The most common data sources, besides the basic sources, are the socket source and Kafka source.

A socket source behaves like a TCP client and is implemented as a receiver-based source. It connects to a TCP server on a network location that can be identified by its host-ip:port combination. The method available in SparkContext can be represented with the parameters hostname, port, converter and storageLevel. This source is often used as a test source because it is comparatively easy to create and is often used as an example in Spark Streaming. For simple test cases, both client and host can run on the same machine, so the host specification localhost is used. There is also a simplified version for text streams, which only queries host and port and is often used as an example due to its simplicity (Maas & Garillot, 2019, S. sec. 19).

```
newStream = ssc.socketNewStream("localhost", 9092)

val newStream = ssc.socketNewStream("localhost", 9092)
```

In the context of streaming applications, Apache Kafka is often used as a data source which will be explained in detail later in this unit. The Kafka source is available as a separate library, which can be imported into the streaming project's dependencies. The method to create a Kafka direct stream is `createDirectStream` in `KafkaUtils`. This could like like the following:

Figure 32: Code for creating Kafka direct stream



```
def stream(ssc):

    host = "localhost:9092"
    topic = "testTopic"
    newStream= KafkaUtils.createStream(ssc, host, "spark-streaming-consumer", {testTopic})
    KafkaStream = KafkaUtils.createDirectStream(ssc, topics = ['testTopic'], kafkaParams)


    return newStream
```

Besides Kafka, there is also the possibility to include a variety of other data sources, many of which can be found under **Apache Bahir**. Among them are Apache Spark and Flink extensions such as the following (Maas & Garillot, 2019, S. sec. 19):

Apache Bahir
This repository provides extensions such as streaming connectors as well as SQL data sources.

- Apache CouchDB/Cloudant
- Akka
- MQTT
- Twitter
- ZeroMQ
- PubNub
- Google Cloud Pub/Sub

After the data has been processed in SparkStreaming using DStreams, it is to be externalized using so-called output operations. The component for this is called sink.

The core library of Spark Streaming provides some built-in output operations (Apache Software Foundation, 2022b):

- `DStream.pprint (num = 10)` will print the first 10 elements of the DStreaming at every streaming interval.
- `DStream.saveAsXYZ (prefix, suffix)` allows output to a file-based sink. Using pre- and suffix, the name can be located in the target file system. The syntax for this is `prefix-<timestamp_in_ms>[.suffix]`. For example, the operators `saveAsObjectFiles`, `saveAsTextFiles` or `saveAsHadoopFiles` can be used.

- `DStream.foreachRDD(func)` provides access to the underlying RDD and is a general-purpose output operation. All other output operations are fundamentally based on it; it can be called the native output operator. This operator will perform the function provided as an argument on every RDD in the stream. Accordingly, it can also be used to transform the data.

Spark within the Hadoop Ecosystem

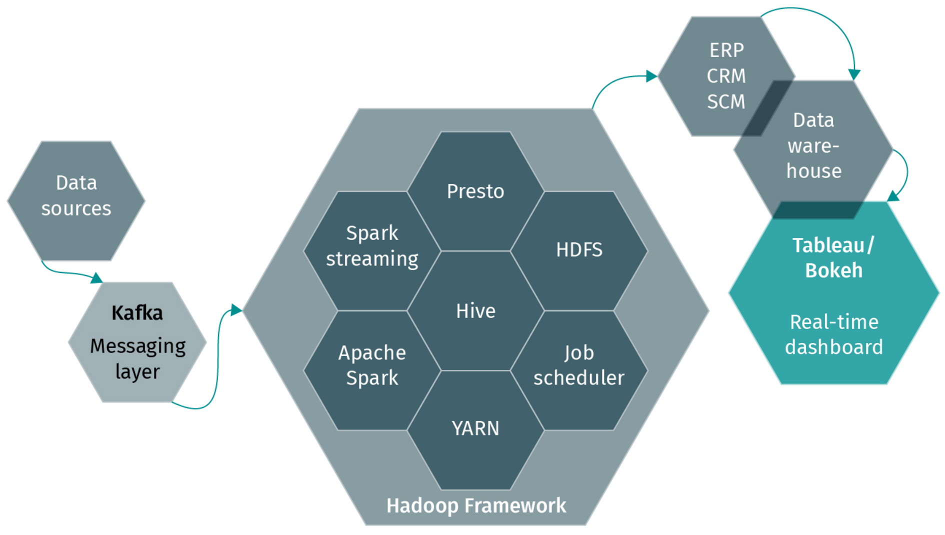
Apache Spark, just like most Hadoop components works on multiple machines in a cluster and can be a replacement for Hadoop in several situations, such as replacing the map-reduce function of Hadoop with batch-processing of Spark, or even extending its functions with micro-batching in Spark Streaming. Spark can also work with **YARN**.

YARN
The abbreviation stands for Yet another resource negotiator, which splits up functionalities of resource management and job scheduling

Since Spark is feasible as a replacement for some Hadoop functions, it is possible to fully integrate it in the Hadoop ecosystem. This allows you to take advantage of both worlds.

For example, a setup like the one shown in the following graphic can be used for this purpose.

Figure 33: Big Data Architecture: Spark in the Hadoop Framework



Source: von Bargen, 2022

As shown in the graphic above, the setup initially consists of the data sources, a messaging layer, the Hadoop framework, and various export functions such as a dashboard, a **data warehouse**, or even the transfer of the data to ERP, CRM or SCM systems.

Data Warehouse
The Data Warehouse is a central database, which is optimized for analytical operations, in which data from heterogenous sources can be stored.

Here, depending on the needs of the company, an individual structure of components can be selected.

For example, to build a data lake, an organization needs to decide the following points (Damji, Wenig, Das, & Lee, 2020, S. sec. 9):

- **Storage system** - Either HDFS can be used or a cloud object store for example those of Microsoft Azure (Data Lake Storage), Amazon Web Services (S3) or Google Cloud.
- **File Format** - Depending on the downstream workload, the organization needs to know what file format the data is in, either structured such as Parquet or ORC, semi-structured, such as JSON, or even unstructured as in image, audio or video files.
- **Processing engine(s)** - Depending on the analyses to be performed, a processing engine is selected. This can be either a batch processing engine, such as Spark, Presto or Hive, or a stream processing engine, such as Spark Streaming. Machine learning libraries could also be integrated, such as Spark MLlib.

Spark is often a good choice here, as it includes various key features that are needed. On the one hand, a variety of different workloads are supported, but also various file formats. In addition, Spark allows data to be accessed from any storage system that supports the Hadoop APIs. These are reasons why Spark has become a de facto standard in the Big Data environment and most cloud or **on-premise** storage systems already offer implementations for this. However, it must be considered that, especially for some cloud environments, a special configuration may be necessary to access the data in a secure manner (Damji, Wenig, Das, & Lee, 2020, S. sec. 9).

On-premise
The term on-premise or on-prem refers to a licensing model for local use of server-based computer programs. The alternative to this would be cloud use.

5.2 Kafka

Amongst the most popular streaming frameworks are Spark Streaming and Apache Kafka. In this section, we discuss the streaming framework Kafka. First the origin and main concepts are introduced, in order to regard afterwards the components and function in more detail.

Kafka: Origin and Main Concepts

Origin

Kafka has been originally created by LinkedIn, who needed to solve their data pipeline problem. LinkedIn had two systems in use: One for internal uses, such as application monitoring, and one for tracking user activity. The requirements of both use cases could not be met using the same backend service: The monitoring service had a data format which was not suited for activity tracking, and the polling model was not compatible with the tracking service push-model, in which frontend servers would periodically connect and publish a batch of messages to the HTTP service. The tracking service used batch-oriented processing, so it was not a good fit for real-time monitoring. On the other hand, due to the similarity of the collected data, it was reasonable to find a solution that would be able to combine both systemic approaches. After confirming there was no scalable solution for providing real-time access to the data available yet, LinkedIn developed a message queue system themselves: Key goals were to use a push-pull-model to decouple producers and consumers of information, as well as providing persistence for messaging data within the

Kafka
A writer’s name for a writing system: The name actually derives from Franz Kafka.

system., Therefore, the system should allow for multiple consumers while keeping high throughput by horizontal scaling. Their publish/subscribe messaging system, Kafka, was released on GitHub as an open-source project in 2010 (Narkhede, Shapira, & Palino, 2017, S. 14f). Today it is an open-source project mainly maintained by Confluent, a company with its origin at LinkedIn, under the Apache stewardship.

Main Concepts

In today's view, Kafka can be used in different ways:

1. It's original use case of moving all occurring event data to a central data warehouse
2. Kafka also persist data making it able to be read hours or even months after it has been written. This opens another use case opportunity. Kafka can also be used as a central system for not simply exchanging data from one system to another but making it the central hub for working with data in general.
3. Every event is saved in Kafka and every other service can act upon the stored data. Thus, it can also be an interface for numerous software services to communicate with one another. Thus, Kafka can also be seen as a data pipeline. (Zelenin & Kropp, 2022, S. 3)

Latency
This describes a time delay between request and response from a system.

The main aspects of Apache Kafka are it's distributed, resilient architecture, which makes it fault tolerant, highly available and horizontally scalable. A Kafka Cluster can scale to a thousand brokers that process trillions of messages per day or petabytes of data. It is built for high throughput and has a **latency** of less than 10ms – which makes near real- time processing possible. It offers built-in stream processing and various client libraries allowing for processing the data in various programming languages. This shows in Kafka's popularity: It is reportedly used by over 80% of the Fortune 100 companies as of early 2022 (Apache Kafka, 2022).

The following table provides a short comparison of Kafka and another popular stream processing system, Spark Streaming.

Table 8: Comparison of Key Features of Streaming Frameworks

Spark Streaming	Kafka
Based on Spark clusters, HDFS or a similar system is needed	Offers a Java library called Kafka Streams, no additional data store needed
Micro-Batch Processing	Event-at-a-time Processing
Higher latency	Low latency thanks to continuous processing
Multiple programming languages supported (Python, Scala, Java, R)	Kafka Streams library limited to Java and Scala as programming language, but through a REST Proxy connection to a large number of clients it is possible to use different programming languages

Components

Now, how does Kafka work? We learned that it combines various components in a distributed system. But what are these components and how do they interact with each other? Before looking at the components in more detail, the terminology within the Kafka framework should first be known.

Terminology

In a Kafka Architecture we will usually be working with the following components:

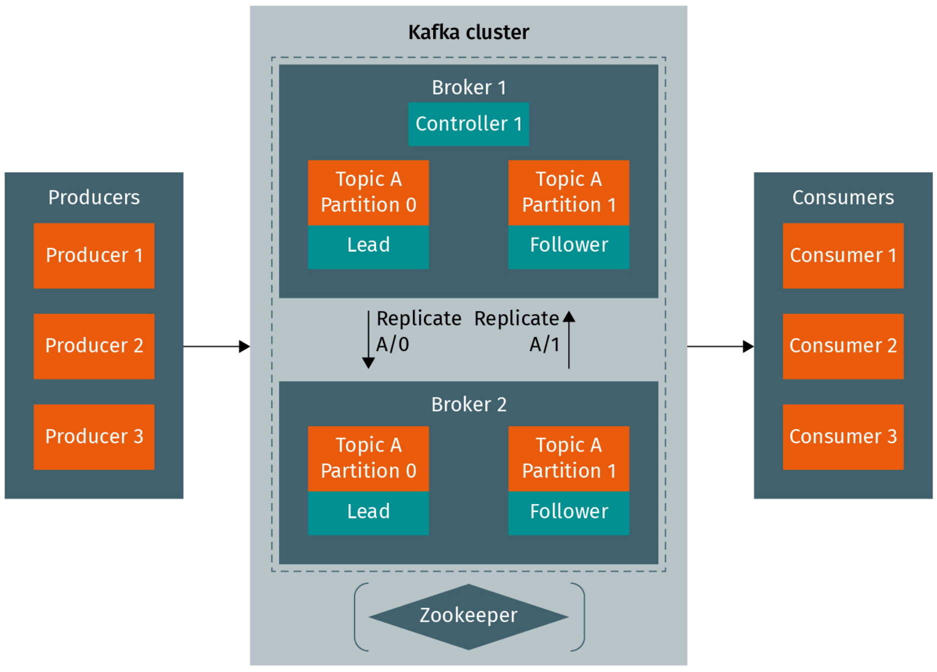
- **Producer**
- **Kafka Cluster** holding multiple brokers and possibly the ZooKeeper
- **Consumer**

There are also two components that describe how the data is organized in Kafka:

- **Topic**
- **Partition**

Messages are sorted by so-called Topics in Kafka. To distribute the data across the cluster for better parallelization and system reliability, it is also partitioned. Each Broker can hold multiple topics, which then are split between multiple partitions.

Figure 34: Kafka Components



Source: von Bargen, 2022

Apart from this, the following terms are also relevant in the Kafka context and need to be understood.

- **Offset**
- **Lead**
- **Follower**
- **Controller**

We will start with the point, where the data is collected, this is called a producer. A producer is usually a data producing application or sensor.

ZooKeeper
This is a centralized service for atomic synchronization of requests in a distributed network.

The data are sent by the producers to the brokers in a Kafka Cluster. Data in the Kafka Cluster are appended to their respective log within their topic. In previous versions of Kafka, **ZooKeeper** was needed in order to have a possibility of communication between the brokers. Since version 3.0 the broker can fulfill this task self-efficient, but there are cases in which it would still be recommended to have ZooKeeper in place (Karantasis, 2021). There are different roles, a broker can take: Controller, Lead, and Follower. In one cluster, there is always one controller broker. The task of the controller is to manage the states of the partitions and their replications as well as administrative tasks. The process of selecting a broker to be the controller is called Kafka Controller Election. After the data are stored within the brokers in their respective topics and partitions, this data is getting replicated. This is done in a way so that each part of the data is stored on a lead broker and, in addition, at least one follower broker.

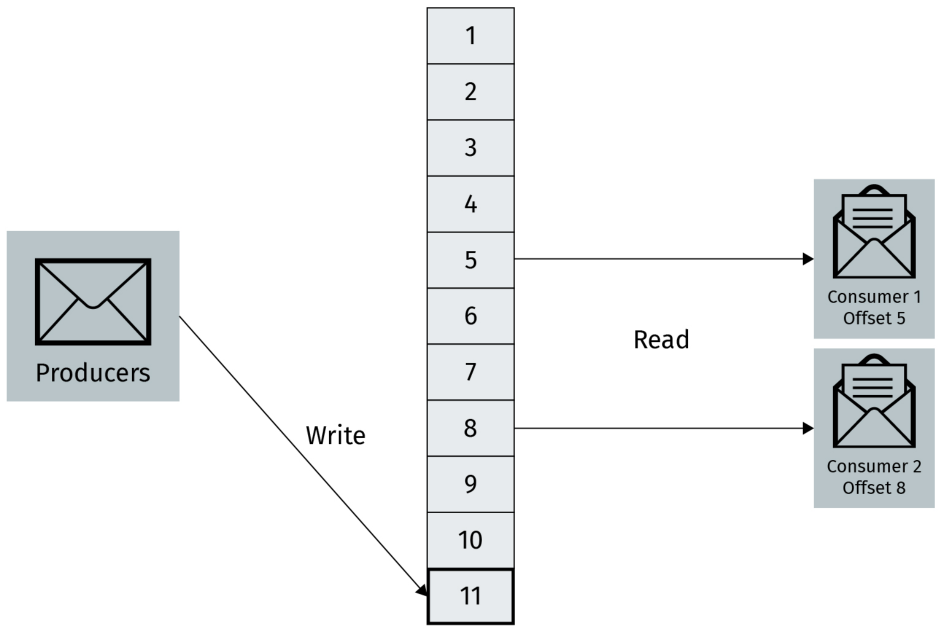
From here the data can be retrieved by any number of consumers subscribed to a certain topic. Consumers are instances that are reading the data for further action. This step is asynchronous to the write process. (Kleppmann, 2017, S. 447).

But let's take a closer look at how the data is stored in topics and partitions. The process of partitioning is the division of a topic into smaller units. The number of partitions can be chosen upon creation of a topic. Partitions are used for easy replication of the data and parallelization capabilities. The so-called replication factor describes to how many brokers a partition of a topic is replicated. In the graphic above, a factor 2 replication is shown, where for each lead partition there is one replica. A partition with factor x exists x-times in the cluster. For each of these existing identical partitions there exists one lead partition, the replicas are called followers. Within the cluster each broker is the leader for some partitions and follower for other partitions.

One key characteristic of Kafka is its so-called structured immutable commit log. This means that the order of messages is preserved (immutable). One of many advantages of this is that old messages remain readable even after updates to the way data is structured in the system. Also, this allows for central updates on topics to all parts of the systems. Thisе logfile has a default segment size of 1GB and a new segment will automatically be created after either a certain time, `log.retention.ms` with a default time of 168hr or 7 days, or when the size of one individual file reaches the `log.segment.bytes`. Depending on the set retention policy, these log files can be persisted or cleaned up. This is either defined on a global scale or specifically per topic. Kafka can only delete full segments though and not specific data entries which is desired for most use cases with respect to data quality (Apache Software Foundation, 2022).

This brings us to another term that must be known in the context of Kafka: Offset. New messages are appended to the end of each topic and assigned a number. This number is incremented according to the log sequence and it is called the offset. This number is relevant both for writing and for reading the messages, as this provides the only consumer specific metadata. The asynchronous setup allows the current offset number of each consumer to be stored, so that the consumer can always continue working with the last message it read. No data is lost for any application and information is not skipped even if the network connection is temporarily interrupted. This is illustrated in the following graphic.

Figure 35: Offset Writing and Reading



Source: von Bargen based on Kleppmann, 2017, P. 448

Kafka Ecosystem

The Apache Kafka ecosystem includes Apache Kafka Connect and Apache Kafka Streams, as well as the Confluent Schema Registry and the Confluent REST Proxy (Apache Software Foundation, 2022). Let's discuss each of these components and their respective role in the Kafka ecosystem.

- **Apache Kafka Connect** is a framework for connecting, importing, and exporting third-party data.
- **Apache Kafka Streams**, in contrast to Apache Kafka Connect, is not an entire framework, but solely a Java library that is used to stream data in near real-time in a fully automated manner. It is used to enable stream processing chains of aggregations and transformations of the data. Kafka Streams fetches the messages as soon as they are created by the producer and can already perform operations and analyses on these messages on-the-fly. The consumers can then read the already processed data in near real-time.
- The **Confluent Schema Registry** is a service for decoupling producers and consumers at the data level. A consumer can use this to retrieve the schema before the data is processed in order to validate the data. The schema itself is a JSON file (Confluent, 2022b).
- The **Confluent REST Proxy** provides a service for producers or consumers who cannot connect natively to Apache Kafka, for example, due to a firewall that prevents such a connection. Another use case would be that producers or consumers are written in a programming language that Apache Kafka does not support. In this case, communica-

tion can take place over the HTTP protocol by producers or consumers sending REST commands to the proxy, which in turn converts them into Apache Kafka commands and sends them to Kafka (Confluent, 2022a).

Example

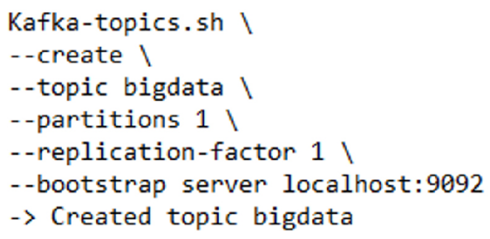
Now that we know how Apache Kafka works in principle, how can we put this framework to use? In the following section we will learn how to create a topic and how to send and receive messages in Kafka.

Apache Kafka should be installed first. Detailed instructions on how to install Kafka can be found in the documentation for the latest version of Kafka (Apache Software Foundation, 2022).

Once Kafka is installed on your local machine, we want to create our topic, which we will call "bigdata". The following code example shows the necessary parameters**:**. You can create this command in a **shell script**. We will go through each command step-by-step.

shell script
A shell script is a program, which gets interpreted and executed by theUnix shell. Ultimately it is an executable text file, in which instructions can be used that a user can also use in the command line of shell.

Figure 36: Code for creating topics in Kafka



```
Kafka-topics.sh \
--create \
--topic bigdata \
--partitions 1 \
--replication-factor 1 \
--bootstrap server localhost:9092
-> Created topic bigdata
```

Source: von Bargen, 2022

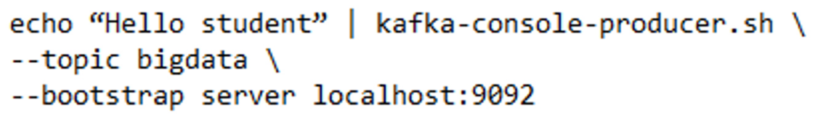
First, with `--create --topic` we specify that we want to create a new topic and give it a name. Optional are the parameters `replication-factor` and `partitions`. In this case, both are one, as in this example we assume, that only one broker is started. However, it is generally good practice to get used to including these parameters with every command. Default values should never be relied upon. Later, when we run Kafka on a larger cluster with hundreds of brokers, for example, in the cloud, we simply increase these numbers for better reliability and availability of our data system. The last parameter this script needs is the `bootstrap-server`. This refers to the information where this script can find Apache Kafka. For local installations, Apache Kafka is started on the localhost and listens to port 9092 by default.

As an optional step, we can look at the created topic (which does not contain any data yet). The command for describing a topic looks very similar to the create command. The difference is that instead of `--create`, we use `--describe`. Executing this command gives us the internal `TopicId`, as well as the number of partitions and the replication factor. Furthermore, additional configurations are shown. In this case, the default size of a segment, which is specified as one gigabyte. Although topics can also be created via graphical user interfaces, the command line is still the best way, because this way the command can additionally be documented. Also, for automations the command line is not to be underestimated.

To send the first messages to Kafka, we need the command line tool `kafka-console-producer.sh`. Using this tool, the producer can use the command line to write data into a topic in Kafka.

Remember that in order to send data to Kafka, we must select an appropriate topic. In this example we want to write the message "Hello student" to our "bigdata" topic.

Figure 37: Code for writing messages to topics

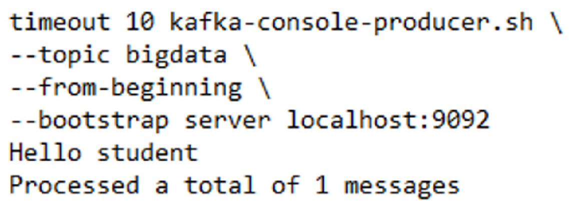


```
echo "Hello student" | kafka-console-producer.sh \
--topic bigdata \
--bootstrap server localhost:9092
```

Source: von Bargen, 2022

Finally, we want to start a consumer and read the message we just sent.

Figure 38: Code for reading messages



```
timeout 10 kafka-console-producer.sh \
--topic bigdata \
--from-beginning \
--bootstrap server localhost:9092
Hello student
Processed a total of 1 messages
```

Source: von Bargen, 2022

In this case, we have added the `--from-beginning` argument to output all messages for this topic. In addition, we have created a `timeout` so that the consumer automatically ends the search after 10 seconds.

It is also possible to display these messages in a continues way. To do this, the consumer is started without the `timeout` command. As soon as new messages arrive, they are displayed in the command line. Also, using a specific offset, we can read data from a particular point in time instead of all the data in the topic.

SUMMARY

Spark Streaming and Kafka are two concepts within the streaming context that can only be compared to a limited extent. Spark Streaming is initially a single tool for processing streaming data, while Kafka is a complete system for message queues.

Spark Streaming is optimized for a wide range of applications, and can be integrated within different frameworks and seamlessly connected to Spark itself. This allows incredibly fast processing through distributed systems. Spark Streaming works with DStreams, which consist of a sequence of RDDs on which basically the same operations can be performed. Spark Streaming's DStreams enable micro-batch processing and can be used in parallel to batch processing with Apache Spark, for example in a Hadoop environment.

Kafka offers individual microservices through the integration of different clients, such as Kafka Streams, which is initially just a Java library and as such is more difficult to compare. With Kafka Streams near real-time processing is possible and can be used for time critical operations with latencies of <10ms. Kafka's diverse use cases as either a messaging layer or for both message queuing and stream processing offer a flexibility in use: It can be integrated with other streaming platforms, including Spark Streaming, making it a good way to combine the benefits of the distributed system with the needs of the individual use case. Here, it largely depends on what the latency and scalability requirements are. A pure Kafka architecture makes sense for real-time analyses, but if certain latencies are not a critical point for the specific use case, a Kafka-Spark Streaming combination can also appear useful, since micro-batching can still be used for comparatively fast analysis.

BACKMATTER

LIST OF REFERENCES

Akil, B., Zhou, Y. & Rohm, U. (2018), O*n the usability of Hadoop MapReduce, Apache Spark & Apache Flink for Data Science*.

Apache Software Foundation. (2022a). *Apache Spark*. Retrieved from https://spark.apache.org/

Apache Software Foundation. (2022b). *PySpark Streaming*. Retrieved from https://spark.apache.org/docs/latest/api/python/reference/pyspark.streaming.html

Apache Software Foundation. (2022c). *Streaming Programming Guide*. Retrieved from https://spark.apache.org/docs/latest/streaming-programming-guide.html#discretized-streams-dstreams

Azarmi, B. (2016), *Scalable Big Data Architecture*. Apress.

Chambers, B., Zaharia, M. (2018): Spark. The definitive guide: Big data processing made simple. 1st Edition. Sebastapol, CA: O'Reilly.

Chapter 8. Data Types. Retrieved from https://www.postgresql.org/docs/current/datatype.html

Countrymeters. (2021). *Population of the world and countries*. Retrieved September 3, 2021, from https://countrymeters.info/

Damji, J. S., Wenig, B., Das, T., & Lee, D. (2020). *Learning Spark, Lightning-Fast Data Analytics. 2***nd** *Ed.* Sebastopol: O'Reilly Media.

Daniel, J.C. (2019): *Data Science at scale with Python and Dask.* Sebastopol, CA: O'Reilly.

Dean, J. & Ghemawat, S. (2004). *MapReduce: Simplified Data Processing on Large Clusters* (USENIX Association, Publ.). https://www.usenix.org/legacy/publications/library/proceedings/osdi04/tech/full_papers/dean/dean.pdf

Department of Earth and Environmental Sciences - Frequently Asked Questions. Retrieved from https://xraylab.esci.umn.edu/frequently-asked-questions

Garillot, F., van Maasakkers, G., & Maas, G. (2019). *Learning Spark Streaming. Mastering structured streaming and Spark Streaming.* Sebastopol: O'Reilly Media.

Goldberg, D. (1991). What Every Computer Scientist Should Know About Floating-Point Arithmetic. *Computing Surveys*(March 1991).

GridFS. (2021). Retrieved from https://docs.mongodb.com/manual/core/gridfs/

How is Unstructured Data Used in a Database? (2021). Retrieved from https://www.mongodb.com/unstructured-data/database

INSPIRE. (2022). INSPIRE. https://inspire.ec.europa.eu/

Interface Age Staff. (1980). *Interface age: General purpose software* (Vol. 2). Dilithium Press.

John, T., & Misra, P. (2017). *Data Lake for Enterprises [electronic resource]* (1st ed ed.): Packt Publishing.

JSON and BSON. (2021). Retrieved from https://www.mongodb.com/json-and-bson

Karau, H., Konwinski, A., Wendell, P. & Zaharia, P. (2015), Learning Spark. 1st Edition. O'Reilly.

Kilbourne, J., & Williams, T. (2003). Unicode, UTF-8, ASCII, and SNOMED CT. *AMIA ... Annual Symposium proceedings. AMIA Symposium, 2003*, 892-892.

Kitchin, R. (2021). The Data Revolution: A Critical Analysis of Big Data, Open Data and Data Infrastructures (2. Aufl.). SAGE Publications Ltd.

Kitchin, R. & McArdle, G. (2016). What makes Big Data, Big Data? Exploring the ontological characteristics of 26 datasets. Big Data & Society, 3(1). https://doi.org/10.1177/2053951716631130

Kleppmann, M. (2017): *Designing data-intensive applications. The big ideas behind reliable, scalable, and maintainable systems.* 1st Edition. Sebastopol, CA: O'Reilly.

Lammel, R. (2008), *Google's MapReduce programming model*. Science of Computer Programming Volume 70, Issue 1, 1

Luntovskyy, A., & Globa, L. (2019, 9-13 Sept. 2019). *Big Data: Sources and Best Practices for Analytics.* Paper presented at the 2019 International Conference on Information and Telecommunication Technologies and Radio Electronics (UkrMiCo).

Maas, G., & Garillot, F. (2019). *Stream Processing with Apache Spark.* Sebastopol: O'Reilly Media.

Most popular social networks worldwide as of January 2022, ranked by number of monthly active users. (2022). Retrieved from https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/

n.a. (n.d.) Dask Project Webpage (URL: https://Dask.org/ [last accessed: 08.03.2021])

Our World in Data. (o. D.). GitHub. https://github.com/owid

Pai-Dhungat, J. (2020). Invention of CT-Scan. *J Assoc Physicians India, 68*(5), 53.

PostGIS - Spatial and Geographic objects for PostgreSQL. Retrieved from https://postgis.net

The Python Software Foundation. (n.d.). *The ElementTree XML API*. https://docs.python.org/3/library/xml.etree.elementtree.html

SQL Injection. Retrieved from https://www.w3schools.com/sql/sql_injection.asp

Team, T. H. (2015). Publish & Subscribe - MQTT Essentials: Part 2. *MQTT Essentials.* Retrieved from https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe/

typing — Support for type hints. (2022, 2022-03-16). Retrieved from https://docs.python.org/3/library/typing.html

Walters, R. (2017, 2021-06-23). Getting Started with Python and MongoDB. Retrieved from https://www.mongodb.com/blog/post/getting-started-with-python-and-mongodb

White, T. (2015), *Hadoop. The Definitive Guide.* 4th Edition. Sebastopol, CA: O'Reilly.

Zhang, J., Porwal, S., & Eaton, T. V. (2020). Data preparation for CPAs: Extract, transform, and load; ETL processes unearth the fuel needed to power the analytics and visualizations that unlock business insights*(6),* 50. Retrieved from http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=edsgsb&AN=edsgsb.A644557535&site=eds-live&scope=site

LIST OF TABLES AND FIGURES

IU Internationale Hochschule GmbH
IU International University of Applied Sciences
Juri-Gagarin-Ring 152
D-99084 Erfurt

Mailing Address
Albert-Proeller-Straße 15-19
D-86675 Buchdorf

media@iu.org
www.iu.org

Help & Contacts (FAQ)
On myCampus you can always find answers
to questions concerning your studies.