Course Book

BIG DATA TECHNOLOGIES

DLMDSBDT01

iu
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

DLMDSBDT01
Version No.: 001-2023-0921
N. N.

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

Date, C. J. (2003). *An introduction to database systems*. Pearson.

Kleppmann, M. (2017). *Designing data-intensive applications.* O'Reilly.

Wiese, L. (2015). *Advanced data management.* De Gruyter.

REQUIRED READING

UNIT 1

Buhl, H. U., Röglinger, M., Moser, F., & Heidemann, J. (2018). Big data: A fashionable topic with(out) sustainable relevance for research and practice? *Business & Information Systems Engineering, 5*, 65—69.

UNIT 2

Khasawneh, T. N., Al-Sahlee, M. H., & Safia, A. A. (2020). SQL, NewSQL, and NOSQL databases: A comparative survey. *Proceedings of the 2020 11th International Conference on Information and Communication Systems (ICICS)* (pp. 13—21). IEEE.

UNIT 3

Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. *Proceedings of the Sixth Symposium on Operating System Design and Implementation* (pp.137—150). Google Research.

UNIT 4

Sharma, K., Marjit, U., & Biswas, U. (2018). Efficiently processing and storing library linked data using Apache Spark and Parquet. *Information Technology and Libraries, 37*(3), 29—49.

UNIT 5

Farcic, V. (2016). *The DevOps 2.0 toolkit : Automating the continuous deployment pipeline with containerized microservices*. Packt.

FURTHER READING

UNIT 1

Ellingwood, J. (2016, September 28). *An introduction to big data concepts and terminology*. DigitalOcean.

UNIT 2

Cattell, R. (2011). Scalable SQL and NoSQL data stores. *ACM SIGMOD Record, 39*(4), 12—27.

Molinaro, A. (2006). *SQL cookbook: Query solutions and techniques for database developers*. O'Reilly.

Pavlo, A., & Aslett, M. (2016). What's really new with NewSQL? *ACMSIGMOD Record, 45*(2), 45—55.

Perkins, L., Redmond, E., & Wilson, J. (2018). *Seven databases in seven weeks: A guide to modern databases and the NoSQL movement* (2nd ed.). Pragmatic Bookshelf.

UNIT 3

van Dongen, G., & van den Poel, D. (2020). Evaluation of stream processing frameworks. *IEEE Transactions on Parallel and Distributed Systems, 31*(8), 1845—1858.

UNIT 4

van Leeuwen, L. (2019). *High-throughput big data analytics through accelerated Parquet to Arrow conversion* [Master's thesis, Delft University of Technology]. TU Delft Repository.

UNIT 5

Ekka, S., & Jayapandian, N. (2020). Big data analytics tools and applications for modern business world. *2020 International Conference on Electronics and Sustainable Communication Systems (ICESC) (*pp. 587—592). IEEE.

LEARNING OBJECTIVES

The focus of this course book is to introduce you to the subject of big data from a technical perspective.

The first unit introduces you to several key aspects of big data, namely its volume, velocity, variety, and veracity. You will also learn about different data sources and types, as well as how new sources can be integrated into a system.

The second unit is all about databases, from very basic data structures and the relational database model and its usage, to more complicated and modern NoSQL and NewSQL databases.

In the third unit we do a deep dive into data formats, going from simple text formats to highly optimized columnar storage systems and sophisticated serialization formats.

The following unit is devoted to the topic of processing frameworks for big data. This includes processing large amounts of data all at once, systems that deal with continuous streams of data, and retrieving the data you want.

The final unit teaches you how scalable infrastructures for big data systems are built. You will learn how containers are used to build clusters of computers, what architectures are used to deal with big data, and how to bring such systems to the Cloud.

UNIT 1

DATA TYPES AND SOURCES

STUDY GOALS

On completion of this unit, you will have learned…

- the definition of big data.
- what kind of data is processed in big data applications.
- how data can be integrated from different sources into one target system using an ETL approach.
- the difference between structured, unstructured, and semi-structured data.

1. DATA TYPES AND SOURCES

Introduction

In this unit, you will be introduced to the defining characteristics of big data and how it is stored, managed, and analyzed. You will also learn what distinguishes big data systems from traditional database systems, and what historical developments led to their existence.

While it's hard to capture precisely what makes a big data system "big" in general terms, you will learn that the concepts of volume, velocity, variety, and veracity—often dubbed the "four Vs"—are central to identifying such systems.

This unit will also introduce you to various data sources that naturally come up in modern business contexts. Large datasets need to be collected, stored, and integrated so that businesses can make use of them, extract useful information, and build complex analytics applications on top of them.

Finally, you will learn that data come in various types, and that depending on the data's origin and nature, must be stored in different formats. Storage and aggregation of such heterogenous data types can pose a challenge to businesses. This challenge can be addressed by leveraging modern big data infrastructures and following industry best practices.

1.1 The Four Vs of Big Data

The term "big data" has been in active use since the late 1990s, but its definition has evolved in the last decades. Like other hyped terms, "big data" has become a buzzword that is often over- or misused, which is why you see certain circles avoid it altogether.

Nevertheless, modern data systems in the twenty-first century are different in nature from what were seen decades ago. One important difference is that we're collectively creating and storing ever more data. There's no doubt that larger and larger datasets have contributed to the rise of big data, but size is only one part of the equation. Viewed more holistically, what we've coined "big data" is our solution to the need to store, manage, and process large, complicated, and varying data that come in at a fast pace. To capture the complexity of big data, literature written on the topic often refers to the four Vs: volume, velocity, variety, and veracity.

In the literature, big data is sometimes defined by just three Vs, namely volume, velocity, and variety. In other definitions, even more Vs are mentioned. One example is "value", meaning that big data should serve to extract meaningful value from data, e.g., by means of applying machine learning algorithms.

Volume

Enterprises like Google, Facebook, Amazon, and Netflix store multiple petabytes of data (sometimes even more). For storing and processing such large datasets, hundreds of thousands of machines are necessary.

The scales of large datasets

Bytes are the fundamental digital storage units. While personal hard drives are mostly measured in terabytes these days, the volume of data that large companies have to handle easily goes into peta-, exa-, or even zettabytes. There is no indication that our need to store ever-increasing amounts of data will slow down any time soon.

It is a challenge for big data technologies to manage these vast amounts of data across multiple storage nodes so that it can be retrieved, analyzed, and written with high performance.

Replication to avoid data loss

When machines fail and you don't back up your data, you might incur partial or even complete data loss. Because some data (like bank account information) are highly critical, losing data is usually a bad for a company. To tackle this problem, it is standard practice to replicate data accross several nodes, i.e., store multiple copies of the exact same data on different machines. While this introduces redundancy and higher storage cost, it also minimizes the risk of data loss. On top of that, it improves high availability and fault tolerance of your applications.

Hardware failures happen all the time, and it's important that these failures be compensated by the system. Data replication is one technique to avoid loss of data and compensate for machine failures. When one storage node in a cluster fails, another node can continue its work because it has a redundant copy of the data. Other techniques for compensating failures include logging information, like states and operations, or a redundant execution of computing tasks on multiple nodes.

Velocity

Velocity refers to speed. Typical big data systems store and manage large amounts of data at increasingly high rates. The speed at which new data are generated, changed, and processed is a challenge. Users of social networks like Twitter, Facebook, or YouTube continuously produce new content. This not only includes millions of Tweets posted every hour, but it also includes tracking information, for example, view counts or user GPS data.

One approach for managing these high amounts of incoming data is to collect a batch of new data first, and only further processing these batches at regular time intervals, and not at first contact. In the application area of data streams, the time intervals in which you collect the batches are often called windows. By slight abuse of notation, we also sometimes call the batch itself a window. With this approach, instead of always being up to date, the number of views per video, for example, is updated only once per hour. Consequently, the

view counter in this example might often show an outdated, and hence inaccurate, value. While you could argue that a slightly outdated video view counter is acceptable, in other scenarios (for example, your current bank account balance) such staleness would lead to confusion for users and might even be harmful. When a new post on social media is created, at least the content-creator themselves should see this new post. This is why in many applications this approach is not accepted anymore. There is a need for real-time (or at least near real-time) data availability. Using such real-time analytics, customer behavior data, like the scrolling speed in a social-media app, cannot only be used for cross-customer analytics, but also for real-time predictions. The probability of whether the user closes the app or click on a specific link or video can continuously be predicted. Optimization algorithms can then choose best-fitting items and show them to the user to control the probability in the desired direction.

Data streams
A data stream is a continuous flow of signals which send or receive data. A data stream is a continuous sequence of data from a provider.

Another area with high data velocity is sensor systems. Each sensor, e.g., a weather sensor, or a sensor in a car or a machine, produces a continuous **data stream**. In many of those applications, it is not feasible for a complete data stream to be stored in a database. Instead, filters and aggregations must first be applied to reduce the amount of data. It is a challenge to find the correct tradeoff between throwing away too much useful data and storing more data than can be handled appropriately. While you could always throw away unneeded data later (which rarely happens), your system needs to ensure that it can handle the throughput of data as well. A system should not slow—or in the worst case, shut down altogether—when it hits peaks of data load. For instance, Amazon needs to be prepared for Black Friday onslaughts without compromising user experience or transaction speed. Also note that in this example the users of the site are primarily responsible for the velocity aspect of big data.

Variety

Schema
The schema of a database defines which data can be stored in it. It is also called the database metadata. In a relational database, the schema consists of tables and their columns.

Big data often has heterogeneous data structures. The opposite of this is homogenous data with a fixed **schema**, as is found in typical relational databases. Think of an offline store with fixed inventory which can be stored in a static set of tables curated and used by accounting. In this example, the schema (names, types, and meanings of the values in your table) likely never change. But sometimes data do not follow a specific schema. For example, sometimes a schema can change over time, and in some cases (for instance, messages sent over a social media network) data do not have a schema at all. A challenge in big data management is to deal with this heterogeneous data. Algorithms have to be flexible in order to work with arbitrary structures. Furthermore, it is often necessary to extract features from the data first, as it is not feasible to analyze large texts, images, and multimedia data in real-time.

Veracity

Data that are produced by humans can be unreliable. Posts on social networks or blogs can contain incorrect information, contradictions, or just plain typos. All this makes it hard for algorithms to extract value from the data. The challenge is to detect which data are trustworthy and which are not. Algorithms are used to measure data quality and perform data-cleaning steps.

Another layer of unreliability comes when several records exist of the same entity. How does an algorithm determine which entry is correct? For example, let's say a user tried to update their status on a social media platform and didn't get a success notification. The same user might switch from their desktop to their mobile phone to repeat the operation, but maybe they enter a slightly different status message this time. The backend of the social media platform now has two very recent status events associated with the same user but sent from different devices. Depending on when these data points were registered in the app, the timestamp may or may not help the platform to decide which status to display.

Veracity is one of the reasons Twitter does not allow users to edit tweets, a topic of agitated discussions and easily the top feature request on their platform. The rationale is that if you retweet or otherwise endorse a tweet that gets altered in hindsight to contain questionable content, it might shine a bad light on you, too. Needless to say, it is technically challenging to modify data at such scale, but there's also an ethical component when it comes to displaying the "right" data to your users.

Lastly, you should consider the possibility that truth or veracity can sometimes not be properly assessed. Many websites try to categorize their content—Amazon, for instance, has a complex hierarchy of product categorizations. There's usually not much of a moral dilemma when it comes to inanimate objects, but problems arise when humans and their behavior become the target of categorization. At worst, algorithms will amplify biases inherent in data, such as racial stereotypes. In many cases, it simply makes no sense to want a single "true" prediction. Since big data systems not only store the original, raw data, but also derived data such as predictions from machine learning algorithms, you need to be careful to assess which parts of the data are truly trustworthy.

1.2 Data Sources

In traditional information systems (like those used in banks or insurance companies), data were mainly collected by the staff of the company. In big data applications, data have their origin in more versatile sources. Examples are customer relationship management systems, user-generated input on social media, or sensor data. Each individual data source presents challenges with regard to the four Vs introduced in the last section: volume, velocity, variety, and veracity. But the combined use of different data sources is another big challenge. They have heterogeneous schemas that may contain duplicate or contradictory data items, and it is often not easy to find correspondences across different data sources.

Web Information Systems

An information system is software that collects, stores, processes, changes, analyzes, and shows data. A web information system runs programs on a web server. For interaction and presentation on the client side, a web browser is used. This makes applications platform-independent and easy to access. Websites such as social networks, communities, online shops, or search engines use this approach.

The term Web 2.0 refers to internet content with emphasis on user-generated content. Before the Web 2.0 era, most websites were more or less static. Internet users were just consumers. Today, the main content of many websites is created by the users themselves. On social networks, users create their profiles, add friends, and write posts and comments. The huge amount of data generated on social networking websites is used by the platforms themselves in order to analyze user behavior and optimize monetization through advertising. Developers can also access APIs (short for "application programming interface") to retrieve data from social media and use them for versatile purposes. For example, a product vendor can use Twitter data to analyze what customers think of their products, what they expect and prefer, and how to optimize marketing campaigns. There are a lot of text-mining algorithms used to gain insights from user-generated text data. For example, sentiment analysis is a popular approach used to detect the emotions behind a text. Simply put, sentiment analysis can analyze a text to tell whether an expressed attitude is either positive or negative. More advanced systems can distinguish a variety of sentiments, even within one sentence.

The challenges in managing and analyzing user-generated data stem from the four Vs of big data. The amount of data created by users is so large and growing so fast that it is not feasible to check and moderate every single entry by a human. As some posts do not have a clear structure, this makes it hard to extract their meaning or value. Posts can also be written in different languages, use slang and abbreviations, or have typos. Lastly, it is not always clear what is true and what is not.

Multimedia Data

Videos, music, voice recordings, and multimedia documents such as presentation slides are even more difficult to analyze than textual posts. Proper preprocessing of these input formats is one of the most important steps required for storing such data. Simple algorithms for image preprocessing can be used to extract the size or the main color of an image. More complex algorithms that use machine learning techniques can detect what is shown on an image or who people in an image are, and "sense" whether those people look happy or sad. Music files can be analyzed by extracting their melody progression, speed, style, key, and the band or composer. In videos, speech recognition techniques can extract the spoken text. Image-mining algorithms applied to individual frames or keyframes of a video to extract information about the content can be used here as well.

Whenever media data are used as a source for big data technologies, it is important to identify the features that should be extracted from them. After feature extraction, a multimedia document can be thought of as a row in a table. For example, after recognizing that an image shows a dog, the object description "dog" can be stored in the image metadata as a string value. Later, it can be used in search queries and analytics. Keep in mind that because images can contain multiple objects (and these objects can differ greatly in their attributes), using relational model tables with a fixed schema can become difficult to justify, and perhaps even impossible to implement. In general, multimedia data are highly variated and can only be handled with modern big data technology solutions.

Log Data, Monitoring Data, and Sensor Data

Sensors return measurement data at certain points in time, either periodically or when triggered by events. For example, a weather sensor could write the current temperature, the humidity, and its current GPS position together with the current timestamp into a log file every five seconds. This log file can either be analyzed in real-time or in specific time intervals (for example, only once per day). Real-time analytics enable you to react to specific events almost immediately after they occur. For example, an alarm sound can be played as soon as the weather sensor reports a temperature value below or above a specific threshold. To avoid a false alarm, it may be better to only play that alarm when the temperature is out of this range for longer than 60 seconds, i.e., in more than 12 consecutive log entries. Stream-processing engines support real-time, near-real-time, and window-based analytics on data streams. An alternative way of analyzing log data is loading the full log file into a database. Data processing frameworks allow for directly working on these files. Instead of having to load the full log file, the database will just load the difference, also called the delta, from a previous loading process.

Servers, smartphones, and many other devices also produce log entries similar to those of our hypothetical weather sensor. A web server logs every single request of a web page. Such a log entry contains a lot of information about the client: their IP address, country, city, browser, operating system, monitor resolution, and much more. This makes it possible to analyze click behavior, time spent on specific websites, and whether a visitor is a recurring visitor or a new one. In smartphones there are sensors to collect data like GPS position or battery status. A proximity sensor and the gyroscope can be used in combination to detect whether the phone is in a pocket, or the user is holding it in their hand, or whether it is resting on a desk. These sensors are primarily used for tracking fitness activities, but are also useful for predictive analytics. As an example, a bonus rewards app can send a push notification with a special offer when the user passes a specific store.

Web Pages

HTML pages from the World Wide Web are another important data source for big data applications like search engines. A search engine allows users to find items in a large set of documents that match specific search terms. The most prominent examples are web search engines like Google, Bing, or Yahoo. The documents that are searched consist of trillions of web pages. Each web page has a title, text, a set of outgoing links to other pages, and many more properties. A good search engine is not only able to find documents that match the given search terms, but it should also present them in an optimized order according to the intended meaning of your search query and your current preferences.

A fundamental component of each web search engine is the web crawler. The job of this software is to scan the internet for new web pages, as well as for changes in existing pages. A web crawler takes as an input the URL of a web page, then it opens and **parses** it, storing all relevant information about this site in an index. Furthermore, it scans the document for outgoing links to other documents. The URLs of these links are then used to recursively call the web crawler again to traverse the web graph page after page.

Parsing
A parser for websites extracts the texts, outgoing links, and other information from HTML pages.

The order in which the search results are displayed to the user is very essential. Particularly when there are many results, a good search engine should display the most relevant ones at the very top. One criterion for sorting is the similarity between your search terms and the documents that are returned by the search engine. Note that in this context it is standard to use “search results” and “documents” interchangeably, as to the search engine a web page is ultimately just an HTML document that should relate to whatever you’re searching for.

To formalize one popular way to measure similarity, let’s introduce the following notation. For sake of simplicity let’s say there is a total of N possible words you could type into a search engine. If we define and fix an order of these words, say alphabetically, then we can define a vector of length N (called the search vector) to encode any query into zeros and ones. Specifically, let t be our search query consisting of k terms in total, and let q be the search vector of length N in which the entry q_i corresponding to the search term $t_i$of the query is 1. For all other possible terms, q_i is 0. Analogously, for each document, a document vector d holds the number of occurrences d_i of the term t_i in the document. The following formula is called the cosine similarity, and it returns a number between 0 and 1. The larger this value is, the closer a document d matches the search query q. Here, the product in the nominator is the dot product between vectors, which is normalized by the product of the Euclidean norms of search and document vector, respectively.

$$sim(q, d) = \frac{q \cdot d}{|q||d|}$$

Another criterion for sorting search results is the relevance of a document. The idea behind Google’s PageRank algorithm is that a document has a high relevance when many other documents with a high relevance have an outgoing link to it. A simplified version of the PageRank algorithm looks as follows:

$$PR(p_i) = \sum_{p_j \in in(p_i)} \frac{PR\big(p_j\big)}{\big|out\big(p_j\big)\big|}$$

where $in(p_i)$ is the set of pages that link to p_i and $out(p_j)$ are the outbound links on page p_j.

As the function calls itself, PageRank is a recursive algorithm. After starting with an initial PageRank value which is equal for every page, the values converge after a few iterations of the algorithm. While PageRank is not used in Google’s search engine anymore, it has been the backbone of the company’s operation for many years, and the efficient, web-scale implementation of PageRank set Google apart from other search engines at the time, leading to its massive success.

1.3 Data Types

Data are often categorized into three forms: structured, unstructured, and semi-structured. Multimedia data can be mentioned as another form, which includes images, audio, and video.

Structured Data

In a relational database, tables have a fixed pre-defined schema. Before actually loading or storing data, one has to first create a table, specify its columns, and choose appropriate data types. This way, application developers, administrators, and data analysts can rely on that schema. They must provide data in that structure to make it fit, and they should know which attributes the data have and which they don't. A structured database also has performance benefits. As the database management system knows the columns and their data types, it can choose an optimized internal storage structure and appropriate data-compression methods, and optimize queries against your data for faster data retrieval. Database administrators and developers can further optimize performance by creating indexes.

The schema of a relational database is often modeled with entity-relationship diagrams (ERD) or the Unified Modeling Language (UML). Using one of these techniques, a conceptual model is created in an early phase of database development. Administrators, developers, and clients can discuss the conceptual model and improve it according to their individual requirements. After the conceptual model is completed, it can be transformed into a logical model. In a relational database, this logical model consists of tables that are connected by foreign-key relationships.

A fixed schema does not mean that it cannot be altered afterwards. Changing the schema by adding, renaming, or dropping columns is called schema evolution. Query languages like SQL support commands for changing the metadata of a table, but in cases where a table already contains a lot of rows, this can be very expensive. Dropping a column or a later change of a column's datatype must be applied on every single row, which can also be expensive. Adding a new column results in setting a NULL value or a default value in this new column for each existing row. Evolving a schema in such a way is usually called a database migration.

Unstructured Data

Big data from social media, web crawlers, or other sources often have no fixed schema. Plain text is an example of unstructured data. It does not follow a pre-defined structure and is thus hard to handle. Search algorithms on texts are more complex than exact-match queries or range queries on structured data. While not always possible, it is usually beneficial to think about ways to first preprocess unstructured data by extracting features and storing these features in a structured format, for example in a table or by using a full-text index. Partially extracting structured data from unstructured, raw data can be helpful in building efficient data structures.

To exemplify such preprocessing, let's have a look at common preprocessing steps in the domain of natural text data. Not all of these steps have to be carried out, but you often find several of them combined.

- Tokenization splits a text into its elements: words, numbers, dates, and so on.
- Stop-word elimination removes the most common words of a language, like "a", "the", "with", and "at", as they are unlikely to carry much information.

- Stemming transforms a word into its word stem. For example, “cats” is reduced to “cat” and “playing” is reduced to “play”.

Text preprocessing can improve the effectiveness and performance of search algorithms, but in some cases, it can also cause problems. The name of the band “The Who” or the sentence “to be or not to be” is completely empty after stop-word elimination. Algorithms have to detect this and should not mark the combination of these words as stop words in this case.

Fuzzy search
In contrast to an exact search, a fuzzy search doesn’t simply rank matches as “true” or “false”. Rather, it returns a value between 0 and 1, with higher numbers being better matches.

The preprocessing steps from above support a **fuzzy search** on full-text data. When a user searches for texts with the keywords “cats that are playing”, texts are found which contain the words “cat”, “play”, “plays”, and so on. If multiple matching documents are found, algorithms must sort them so that the users see the best-fitting results first. One algorithm which does this is computing the term frequency-inverse document frequency (TF-IDF) for each search term and document. A simplified version of this measure can be calculated by dividing the number of term occurrences in the given document by the total number of its occurrences in all documents:

$$tf\text{-}idf\Big(t, d\Big) = \frac{n_{t,\,d}}{\sum_{k\,\in\,D} n_{k,\,d}}$$

Even this simplified version of the TF-IDF formula shows how it can measure search term relevance that makes sense intuitively. Namely, a search term for a document has a high TF-IDF if, and only if, it occurs frequently within the document and the search term is found rarely in the corpus of all documents. To give an example, if you search a document for the term “cats”, you may find documents that contain that word often; however, since the internet is full of cats, you will still get a relatively small TF-IDF, simply because the word “cats” has such a low specificity. It’s difficult to set yourself apart from other documents in a search engine if the topic is ubiquitous. In contrast, if you were to search for “nuclear fusion”, any document that contains these terms several times will likely be a website about this topic, due to the high specificity of this concept. All things considered, there aren’t that many websites devoted to nuclear reactions.

You’ve been shown a simplified version of the TF-IDF formula, as it captures the basic idea of this algorithm quite well. In general, the “term frequency” term remains the same, but the inverse document frequency is slightly modified to make more sense in practice. Specifically, we use

$$\log\Big(N \,/\, \textstyle\sum_{k\,\in\,D} n_{t,\,d}\Big)$$

as an IDF term, where N is the total number of documents in our corpus. The natural logarithm is applied to this quotient to prevent IDF values from becoming too large. Without applying the logarithm, a very scarce search term only occurring in a single document would lead to an IDF of N. On the other hand, we use N in the nominator of IDF to make sure our TF-IDF measure doesn’t become too small for frequent search items. We still want meaningful results for frequently queried terms.

Semi-Structured Data

Data are called semi-structured when they follow a certain structure, but this structure is not fixed and not pre-defined. Instead, the metadata describing the structure is part of the data itself.

Examples here are **JSON** or **XML** documents. It's difficult to give a complete definition of these two description languages here, so instead we'll focus on giving an example of a JSON document and discuss some of its properties. The following JSON document describes one product in an online shop database:

JavaScript Object Notation (JSON)
A syntax for describing documents whose attributes can be strings, numbers, Booleans, nested objects, arrays, or null.

EXtensible Markup Language (XML)
Language for describing machine- and human-readable documents that form a tree structure.

Code

```
{"_id": 123,
 "description": "T-shirt",
 "price": 15.95,
 "color": "green",
 "size": "M"
}
```

It is not necessary to define a schema to define a JSON object. The attribute names are self-contained within the document. Other products could have the same or many other additional attribute names. You can also create nested JSON objects to model more complex relationships. For instance, the JSON model of a citizen might have an attribute called "previous residences", which is a JSON itself containing all the addresses at which a person has ever lived. Here's an example of such a document, showcasing that JSON also features lists:

Code

```
{"_id": 42,
 "name": "John Doe",
 "previous_residences": [
  "Chicago",
  "Hamburg",
  "Beijing"
 ]
}
```

This flexible schema makes it possible to store heterogeneous data with arbitrary attributes. If you were to fit a dataset of JSON objects into a table, you would have to have a row for each attribute. This comes with many problems. First, there will likely be a lot of rows and many of them will be NULL. Maintenance and extendability also become a nightmare, as each time a new attribute is added, it must be added to all existing rows. This approach is often not feasible, as most classical database management systems have a limit on the number of columns.

The data types in semi-structured data are also not pre-defined. In the document above, the attribute “size” is a string. In other documents, it can also be a number or an array. Many NoSQL databases follow the approach of flexible schemas. But the lack of a fixed schema is a challenge for application developers and analysts. Your application code cannot rely on the existence of a specific schema, because you won’t know which attributes are present in any given JSON object until you inspect it. So your application either has to support arbitrary documents, or it has to restrict the supported documents to those which match a required schema. Only if all documents do so will the application work properly. Developers must be very careful to guarantee that all data are structured in such a way that they can be retrieved and analyzed properly. When the schema changes, either all existing documents have to be migrated to this new schema, or applications have to support both the new and the old schemas.

1.4 Data Integration

Information integration is the process of merging information from different data sources to create a unified data view. For example, say telecommunication company A acquires another company, B. All customers and contracts from B must therefore be integrated into A’s database. Some customers may already exist there, others are new. Maybe B’s customer table does not have a gender column, but this field is mandatory in the target table. These and other challenges have to be mastered when integrating data items from one system into another.

This type of integration work also frequently happens within a company’s database systems (e.g., when a company decides to work with previously untapped data). In the early days of online shops, it wasn’t necessary for companies to track their users beyond the limits of their own domain. Nowadays it’s mandatory for online shops to do so if they want to gain and keep an edge over the competition. This requires them to integrate new user data with existing data all the time.

Schema Mapping

Schema mapping is the first step to take when two data sources that describe the same kind of data are being combined. This can be done manually or semi-automatically. A semi-automatic schema mapping approach analyzes the schemas of two data sources and finds matching attributes by comparing their names, data types, and/or instance values. The output of this algorithm is a list of correspondences. For example, the column “academic_title” in A’s customer table corresponds to the column “title” in B’s customer table. It would be a wrong correspondence if a customer’s title (“Prof.”, “Dr.”, etc.) was matched with a book title, for example. The mapping algorithm can compare typical string lengths to avoid this mistake, or it can increase the probability of a match when other columns of the same two tables have a match as well.

Forms of Heterogeneity

Six forms of heterogeneity between different data sources have to be mastered in typical information-integration tasks.

- Technical heterogeneity: The ways to access data in the different data sources differ. One source may be accessed with SQL queries, while the other source accepts REST API calls. Wrappers can be used to overcome technical heterogeneity. These programs can offer uniform access by delegating requests in the source-specific access method.
- Data model heterogeneity: A data model describes possible data structures. For example, the relational data model supports tables that consist of atomic columns. The JSON data model supports key-value pairs, nested documents, and lists. When two data models differ, it is necessary to transform data from one model into another.
- Syntactical heterogeneity: Differences in syntactic representations of data are often very easy to solve because in most cases, functions can be used to convert values from one syntax into another. Transforming UTF8 texts into another encoding or converting a date value like 28.10.2025 into 2025-10-28 can easily be automated.
- Structural heterogeneity: Two schemas are structurally heterogeneous when they use two different data model concepts for the same real-world concept. For example, in one online shop, books are stored in a book table, games are stored in a game table, and so on. So the shop uses the concept of tables for organizing products into categories. In another online shop, the concept of column values is used. Here, all products are stored in one single product table. A value “book” in a category column indicates that a specific product is a book. With traditional query languages, it is easy to transform one schema into another, but this is only possible when the list of categories is fixed. When they are unknown, or when they can change over time, it is more difficult to develop a generic and flexible transformation.
- Schematic heterogeneity: Transforming one schema into another is easier when the same data modeling concepts are used in both data sources. In one online shop, the name of a product’s category is stored in a “category” column in the product table, as in the example above, e. g., “category = book”. In another online shop, the category names are stored in a separate category table, which only carries categories and their IDs, but no product information. These two tables can be joined on a common category ID to bring category names and other product information together.
- Semantic heterogeneity: On both the metadata and data levels, it is very hard to automatically detect semantic correspondences or mismatches. Different schemas use synonyms, abbreviations, or different languages for the same columns (for example, the English “academic title” and the German “Titel”). To master this form of semantic heterogeneity one can use schema matching algorithms, dictionaries, and thesauruses.

The examples above show that integrating information from different sources is often very complex. It is even harder when data integration is not just a one-time job (e.g., company A needs to import all customers from company B) but an ongoing task. This is the case when different systems continuously produce individual data. Whenever data from one system are required in another, or when a uniform view across different sources is needed —for example, data analytics—data integration has to be performed.

Virtual Integration Versus Materialized Integration

One approach to integrating data from one system into another is not actually moving or copying the data, but rather simply creating virtual access to them. To users it looks as if all data from different sources are present in one single schema, but the data remain in their original format. Whenever a query is sent to the virtual schema, it is forwarded to the source systems. This approach can be a bit slow, but it gives users a view of the data which is always up to date.

Data warehouse
A database that brings together data from different sources and is optimized for analytical purposes is a data warehouse.

Online analytical processing (OLAP)
Complex analytical queries on a multidimensional data model for business analytic or reporting purposes is an online analytical process.

Online transaction processing (OLTP)
For simple queries that affect only a few number of rows, OLTP databases are optimized for handling thousands of those queries at the same time.

In contrast, materialized integration is a very typical approach in **data warehouses**. Data are loaded from their source into the data warehouse. If the data warehouse is optimized for **OLAP**, complex analytical queries, machine learning algorithms, and predictive analytics can be more easily performed compared to the same queries executed directly on the source system, which is typically an **OLTP** database. Furthermore, working with data within the data warehouse and running complex queries does not affect the performance of ongoing queries in the source systems. When a big "orders" table for an online shop database is loaded into the data warehouse, product recommendations, inventory-holding predictions, and other complex analytics queries can be executed there without slowing the performance of the online shopl.

ETL: Extract, Transform, Load

Populating a data warehouse is often done via an ETL approach. There are popular ETL tools or frameworks which orchestrate an entire ETL process. The individual phases of this approach are as follows:

- Extract. The data are extracted from the source. During this phase, data-quality checks can be executed. If a data item does not fulfill specific criteria, it is rejected and reported back for further analysis. Data checks can include type checks, range checks, pattern matching, outlier detection, and more. For example, each price of a product must be a number in the range between 0 and 99999.99, denominated in US dollars. If all constraints are fulfilled, the item is passed to the transform phase.
- Transform. A data item is transformed into a structure that matches the target schema. One important part of this phase is data cleansing. A set of specific rules and functions is applied for filling up missing values, dropping attributes, correcting errors, encoding, decoding, calculating, aggregating, and more. After the transformation phase, a data item is ready to be loaded into the target system.
- Loadset. It is simply inserted with the current timestamp. The timestamp of the former version will then be updated to mark a row as invalid.

SUMMARY

The 4Vs are used to define the term big data. They stand for volume, velocity, variety, and veracity. These four characteristics are at the core of why big data management and big data analytics are so complex. In big data scenarios, data come from many different sources and have dif-

ferent types. Integrating different sources, making sense of them, and gaining insights into the data is a big challenge across many application domains, which explains the need for new specialized technologies for solving big data problems that cannot be solved with traditional approaches.

UNIT 2

DATABASES

STUDY GOALS

On completion of this unit, you will have learned …

- what a relational database is.
- the basics of the query language SQL.
- the different kinds, characteristics, and use cases of NoSQL databases.
- how NewSQL databases differ from both SQL and NoSQL databases.
- when to use timeseries databases for representing data.

2. DATABASES

Introduction

Databases were introduced to offer a software layer for managing data that is accessed by applications. Before there were databases, data were typically stored in files. This requires significant effort to develop efficient data access, and it can lead to data inconsistency. Databases offer an easy-to-use and powerful query language, support indexes and optimizers to enable high performance, and guarantee consistent and protected access to the data.

Over time, many different types of databases have been developed to cater to different use cases. Traditionally speaking, SQL-based relational database management systems (RDBMS) have dominated the database market. They are still widely used today, but some types of applications have shifted towards more specialized forms of database systems. For example, the **Internet of Things**, with its huge number of sensors and other data-collecting devices, lends itself to utilizing the timeseries databases which will be introduced in this unit.

Internet of Things
In the most general sense, the Internet of Things (IoT) refers to everything connected to the internet. More specifically, the IoT is a trend towards connecting more and more devices to the Internet.

In this unit, we will look at relational databases and the query language SQL. Furthermore, younger databases are introduced, namely NoSQL systems and time-series databases.

2.1 Database Structures

The most popular **database management systems** (DBMS) are relational DBMS (RDBMS). They use the relational model (Codd, 1970; Date, 2007) and can be queried using the structured query language, SQL.

Database management system (DBMS)
A database management system is a software tool to work with a database (which itself is just a collection of stored data and metadata). The combination of the database (DB) and the database management system (DBMS) is called a database system (DBS).

The Relational Model

A table in a relational database is called a relation. It consists of rows and columns. The table name, its columns, and the column types belong to the database metadata. The rows, which are also called instances, are on the data level. A relation can be seen as a set of tuples. Its columns are specified when a table is created, each column has a specific data type, and each row in a table has exactly one value in each column. This value can also be “null”, which represents the absence of a value. The following example shows a “products” table with columns titled “product number”, “description”, “price”, and “manufacturer”.

	Description	Price	Manufacturer
2	Pineapple	4.99	Fruitstar
5	Comic book	NULL	Bookbook

The “null” value in the price column of product number 5 could mean that the price is unknown or that the article is out of stock. A price of “null” does not mean that a product is free of charge. If this were the case the price would be 0. You will also have to account for missing or null values when doing analytics—you wouldn’t want these values to lead to mathematical errors when computing average prices, for instance. In general, you have to be careful with null values and avoid them, if possible, by introducing more meaningful default values.

In the example table above, the column “product_number” is underlined. We use this convention to define this column as the primary key of the table. A primary key consists of one or more columns, the values of which uniquely identify each row of the table. Primary key columns must not be null, implying that a primary key should be unique. This means that the product number is never null, and there are no two products that have the same product number. If the primary key were a compound key (combining the “product_number” and “manufacturer” columns), it would be permissible to have two products with the same product number, but only if they are from a different manufacturer. For example, (5, Fruitstar) is a valid primary key of a product row if and only if there is no other product with a product number of 5 manufactured by Fruitstar. The combination must be unique throughout the whole table.

In the following example, a second table is introduced. In this new “manufacturers” table, the primary key is the company column.

	Description	**Price**	**Manufacturer**
2	Pineapple	4.99	Fruitstar
5	Comic book	NULL	Bookbook
11	Gift voucher	10.00	NULL

	Country
Fruitstar	Italy
Bookbook	Australia

It is now possible to define a foreign key on the manufacturer column in the products table which references the company column in the manufacturer table (and which is that table’s primary key). Foreign keys allow you to relate data across several tables. We will see how to connect tables in practical terms in the next sections. For now, note that using foreign keys is how you associate additional information with a given row in a table. For instance, the product listed as product number 2 is from the manufacturer Fruitstar, and you can check that this company’s country of origin is in fact Italy by following the foreign key relation.

A foreign key column only accepts values that are present in the referenced column. So in this example, it is not possible to insert a product with the manufacturer Moviemaster into the products table when there is no such company in the company column of the manufacturers table. Nevertheless, manufacturer values can be null.

Transactions and the ACID Properties

A transaction in databases consists of a series of operations that fulfills the four ACID properties (Haerder & Reuter, 1983). These properties are atomicity, consistency, isolation, and durability.

- Atomicity: A transaction is either executed completely or not at all. Partial execution of only some of the operations of the transaction is not allowed. When an error occurs within a transaction, or when the user decides to abort a transaction, all work that has already been executed has to be rolled back and undone. A successful transaction is always completed with a commit operation. For example, let's say a user inserts two rows in the products table above. After inserting the second row, the system crashes. No row should be inserted in this case because the transaction was not yet committed. Also, if the insertion of the second product fails, either because the product number already exists in another row, or because the user manually aborts the transaction, neither of the two rows should be inserted. Only if both rows can be inserted successfully and the transaction committed will the two rows will be written to the table.
- Consistency: Each transaction brings the database from one consistent state to another. It is not accepted that a transaction leaves the database in an inconsistent state. As you've seen in the example from above, the insertion of a product with an already existing product number is not accepted. If it were, the database would be in an inconsistent state.
- Isolation: As long as a transaction is ongoing and not yet complete, the effects of this transaction will not be visible to other transactions running simultaneously. As an example, when a transaction changes the price of a product, this new price will not be visible for other transactions until this transaction commits. Otherwise, it would result in what's known as a dirty read. The transaction management system must guarantee that all multi-user anomalies, like dirty reads or lost updates, are avoided.
- Durability: A commit durably persists the data that are written by a transaction. After a commit, there is no possibility to roll back the effect of this transaction anymore. These effects have to be written in such a way that are never lost, even in cases like system crashes or hardware failures. This means data must be written on a non-volatile storage medium like a solid-state drive (SSD) or a hard disk. Data only stored in RAM will be lost during a power outage. Furthermore, techniques like replication help avoid a loss of committed data even when the storage medium breaks or the whole data center becomes unavailable.

When working with a database, users, developers, analysts, and administrators can make use of commands like commit or rollback to control a transaction. In typical drivers for database management systems, it is also possible to use a so-called auto-commit setting. Here, every single command is automatically committed, so it forms a separate transaction. If multiple operations should form a single, larger transaction, manual commits have to be used. For example, in a money-transfer application, the two update operations

which reduce or increase the balance of two bank accounts should be executed within one transaction. This way, it is guaranteed that either both updates are executed or neither are.

2.2 Introduction to SQL

SQL is a standardized language supported by all popular relational database management systems. It consists of three main parts: DDL, DML, and DQL.

- Data Definition Language (DDL): This part of the language is used to define the schema of a database. New tables are created with `CREATE TABLE`, they can be dropped with `DROP TABLE`, and modified with `ALTER TABLE`. All DDL commands affect the database metadata which is stored in the DB catalog.
- Data Manipulation Language (DML): These commands are used for manipulating the data stored in the database belonging to the DML. Common commands include `INSERT`, UPDATE, and `DELETE`. `SELECT` is also a DML command—it does not modify the data in-place, but is used to retrieve data in a specific format.
- Data Control Language (DCL): Administrators can use commands like `CREATE USER` or `GRANT` to manage database users and control their privileges. These commands are part of the `DCL`.

Schema Design

A database consists of schemas. Each schema stores an arbitrary number of tables. Both schemas and tables are database objects and both have a name. A table name is unique within the schema it belongs to. When creating a new table, the columns of this new table have to be specified. Each column has a column name, a data type, and optional properties. The following example shows a `CREATE TABLE` command to create a manufacturers table with two columns: company and country.

Code

```
CREATE TABLE manufacturers (company VARCHAR(100) PRIMARY KEY,
country VARCHAR(100));
```

The company column has the data type `VARCHAR(100)` and the `PRIMARY KEY` property. This means it can store string values up to a length of 100 characters and it uniquely identifies a manufacturer. The following list shows the most important SQL data types.

- `INT / INTEGER`: A whole number that can be positive or negative. In many DBMSs, an integer consumes four bytes of space, so it can store numbers between -2,147,483,648 and 2,147,483,647.
- `DECIMAL`: An exact decimal number with a fixed number of total digits and fractional digits. E.g., `DECIMAL(5,2)` can store numbers with up to five digits, two of them after the decimal point and up to three before the point. This means the largest possible

`DECIMAL(5,2)` number is 999.99, the smallest is -999.99, and any number in between is allowed as long as it doesn't have more than two decimal places. So `DECIMAL(N, M)` denotes numbers with at most N digits, at most M of which are decimal places.

- `FLOAT / DOUBLE`: An approximate floating-point number. Typically, `FLOAT` uses four bytes, and `DOUBLE` eight bytes, but that depends on the concrete `DBMS` specifications. As these numbers are approximate, calculations and comparisons may lead to problems.
- `CHAR`: This command indicates a character string with a fixed size. `CHAR(5)`, for example, has a fixed size of five characters. It is not possible to store strings longer than that, but it is possible to store shorter strings as the remaining characters are filled up with trailing spaces. So when storing the value `'Bob'` in a column of type `CHAR(5)`, the value `'Bob '` (including two spaces) will be stored there.
- `VARCHAR`: These are strings with variable lengths up the specified number of characters. In the example above, the `VARCHAR(100)` column can store company names with a maximum length of 100 characters.
- `DATE`: A date value consists of a year, a month, and a day. The typical SQL notation is YYYY-MM-DD, for example, `'2020-12-31'`.
- TIMESTAMP: This is a point in time consisting of a year, month, date, hour, minute, and second. Typically, the seconds are decimal numbers to support timestamps with a precision of one millisecond or less, e.g., `'2020-12-31 23:59:59.999'`.
- `BLOB / CLOB`: Binary Large Objects (`BLOB`) are used to store large amounts of arbitrary data, such as images or audio. Character Large Objects (`CLOB`) can be used to store long text values.
- `GEOMETRY`: This is a data type for storing geometric objects like points, lines, or polygons. Functions for these kinds of data can be used to compute distances, sizes, overlaps, and more.
- `JSON / XML`: This is an arbitrary structured document in XML or JSON format.

Some database management systems support further data types like BOOLEAN, while some do not support `GEOMETRY`,`JSON`, or `XML`, and some databases divide specific data types into multiple types, for example distinguishing between `INTEGER`, `TINYINT`, `MEDIUMINT`, and `BIGINT`. In general, there are a lot of aliases for datatypes. For example, `VARCHAR` is an alias for `CHARACTER VARYING`.

Here's a list of the most important column properties of a DBS.

- `NULL`: This property allows storing `NULL` values. This is the default setting in most relational database management systems.
- `NOT NULL`: With this property, `NULL` values are forbidden. It specifies that a column must be mandatory.
- `UNIQUE`: When a column is specified as `UNIQUE`, there cannot be duplicate values in the column. So for example, `emailVARCHAR(200) UNIQUE` guarantees that each email address is present only once. Nevertheless, an arbitrary number of `NULL` values are allowed in that column.
- `PRIMARY KEY`: The `PRIMARY KEY` property can be written next to that column to specify this column as the primary key of the table. A primary key is always `UNIQUE` and not `NULL`.

- CHECK : A Boolean predicate can be specified to restrict the domain of allowed values in the column. E.g., price DECIMAL(9,2) CHECK (price>=0) forbids storing negative price values.
- DEFAULT: If no value is set, the specified DEFAULT value is taken. E.g., price DECIMAL(9,2) DEFAULT 0 means that the default product price is 0 dollars. If no DEFAULT were set, it would be NULL.
- REFERENCES: This is a foreign-key reference to a specific table and column. Only values that are present in the referenced column can be stored. An example of a reference would be manufacturerVARCHAR(100)REFERENCESmanufacturers(company) in which the column manufacturer of the current table is referenced with the company column of the manufacturers table.

Insert, Update, and Delete

The DML commands INSERT, UPDATE, and DELETE are used to modify rows within a table. INSERT adds a new row by specifying its column values. Let's say we have a products table with four columns, namely, product_number, description, price, and manufacturer. The following two example queries insert new products:

Code

```
INSERT INTO products (product_number, description) VALUES (22, 'Tomato');
INSERT INTO products VALUES (23, 'Banana', 0.80, 'Fruitstar');
```

As you can see in this example, to insert a product you can either explicitly provide a column specification or omit it. If you work with a column specification, you only have to provide the specified values (in that order). In the case that you omit it, values for all columns of the table must be given in the original column order.

An UPDATE query consists of a SET clause and a WHERE clause. The WHERE clause is optional and specifies which rows should be updated. If it is omitted, it affects all rows. The new column values are given in the SET clause. The following two examples show how all product prices can be increased by five percent, and how the manufacturer and price of one specific product can be changed:

Code

```
UPDATE products SET price = price * 1.05;
UPDATE products SET price = 0, manufacturer = 'Yellowfruit' WHERE
product_number = 23;
```

For deleting rows, the WHERE predicate in the DELETE command specifies which rows to delete. The following command will delete the previously inserted banana product by its product number 23:

Code

```
DELETE FROM products WHERE product_number = 23;
```

The SQL standard and all popular database management systems offer further DML commands for manipulating the data within a table. For example, a TRUNCATE command empties a whole table, and the MERGE command combines data of two tables.

Select

The SELECT command is very powerful as it supports multiple clauses for different purposes. The simplest form of a SELECT query is shown in the following example, which returns a full table:

Code
```
SELECT * FROM products;
```

As no WHERE predicate is used here, all rows are returned. It uses a star (*) to select all columns from the products table. Instead of this star, it is possible to restrict the number of columns to a custom subset, apply computations, or introduce column aliases. Here's an example demonstrating these possibilities:

Code
```
SELECT description, price * 0.85 AS price_euro FROM products WHERE price
IS NOT NULL;
```

An arithmetic or comparison operation (+, –, =, >, ...) that is applied on at least one NULL value always returns NULL. And in Boolean predicates, NULL is evaluated to false.

In the examples above, there was only one table in the FROM clause. When two tables are specified and separated with a comma, the cartesian product or cross join of those two tables is computed. This means every row of the first table is combined with every row of the second table. For this to work there needs to be a foreign key relationship between these tables.

A more typical use case is the application of the JOIN operator within the FROM clause. A join connects two rows of two tables when they match a given join predicate. The following query shows a product description together with the country where the product manufacturer is from:

Code
```
SELECT P.description, M.country FROM products P JOIN manufacturer M
WHERE P.manufacturer = M.company;
```

The table aliases P and M are introduced here to simplify the query. Mind that products which have the manufacturerNULL are filtered out here because they do not find a join partner in the manufacturers table. When NULL values are supposed to be part of the results, a LEFT JOIN can be used. This join variant guarantees that each row of the left

table (which here is `products`) will be present in the query result. Similarly, a `RIGHT JOIN` would guarantee that all right table elements are present (which here is `manufacturers`).

Aggregation functions can be used to count values or compute the sum, minimum (`min`), maximum (`max`) or average (`avg`) of multiple values within a group. By default, all rows of a table belong to one single group so that the following query returns just one row:

Code

```
SELECT count(*), sum(price), min(price), max(price), avg(price)
FROM products;
```

count(*)	sum(price)	min(price)	max(price)	avg(price)
122	209.23	0	49.95	1.715

As soon as an aggregation function is used, it is not allowed to use other columns without aggregation functions in the `SELECT` clause anymore, as it wouldn't make sense. For example,

Code

```
SELECT manufacturer, count(*) FROM products;
```

is invalid as the manufacturer is a set of manufacturer values after grouping. This can be solved by using a GROUP BY clause:

Code

```
SELECT manufacturer, count(*) as num_products
FROM products GROUP BY manufacturer;
```

The query now returns as many rows as there are distinct values in the manufacturer column. It shows for every single manufacturer its name and the number of its products. The result could look as follows:

manufacturer	num_products
Fruitstar	34
Yellowfruit	1
NULL	11
…	…

To only restrict the result to those manufacturers that have at least two products, a HAVING predicate is helpful. While WHERE is executed before grouping, HAVING is executed after grouping and typically consists of aggregation functions. The following example uses a HAVING predicate and furthermore an ORDER BY clause to sort the result by the manufacturer name:

Code

```
SELECT manufacturer, count(*)
FROM products
GROUP BY manufacturer
HAVING count(*) >= 2
ORDER BY manufacturer;
```

The default order is an ascending order. This means the manufacturer Bookbook appears before the manufacturer Foodstar. When using ORDER BY manufacturer DESC, the results are sorted in descending order.

For more complex analytical queries, SQL supports window functions, grouping sets, or recursive queries. Each DBMSs extends the SQL language by using extensions and built-in functions, and allows users to develop their own user-defined functions and stored procedures.

2.3 Relational Databases

Relational database management systems (RDBMS) were developed in the 1960s. Before that, there were hierarchical databases and the network model. Due to its simple data model, an easy-to-use query language, extensibility, and performance, relational databases are still the most common databases in the industry. In this section, the most popular systems are introduced.

- Oracle is the worldwide leader of commercial database management systems. There is a free Express version, but it has a limited set of features and can only use 1 GB of memory. Other Oracle database products have a very large feature set and can even be run on a cluster of multiple database nodes as a distributed database.
- MySQL is the most popular open-source database. MySQL has belonged to Oracle since 2010. As it is pre-installed on many web servers, it's often used for dynamic websites such as blog systems or online shops. Using the XAMPP package, it is very easy to install a pre-configured software stack on a web server or personal computer. XAMPP stands for cross-platform, Apache, MySQL, Perl, and PHP. It also contains the web-based database administration tool phpMyAdmin. Database administrators can open this tool in a web browser to connect to the MySQL database, create and manage tables graphically, and write SQL queries. MySQL supports various storage engines. The choice for a specific storage engine impacts not only the internal storage format of a table but also the supported features and database performance. For example, the storage engine MyISAM does not check foreign-key constraints. This makes this engine faster than other engines like InnoDB in some scenarios, but it can lead to inconsistencies in the data.

- MariaDB is an open-source system and almost fully compatible with MySQL. Developers can use the same drivers for connecting applications to a MySQL or MariaDB database. MariaDB is a fork of MySQL and offers more features and more storage engine choices. On big databases, MariaDB often performs better than MySQL.
- Microsoft SQL Server is the commercial RDBMS from Microsoft. SQL Server exists in multiple versions for different purposes; for example, there is one version for big data warehouses. It uses an SQL dialect called Transact SQL (TSQL). This language extends SQL by variables, procedures, exception handling, and more.
- PostgreSQL is the most advanced open-source database. The PostgreSQL database supported advanced features like triggers much earlier than MySQL and other open-source RDBMSs. Many features of PostgreSQL are still not supported by others, and it's also a surprisingly robust and fast choice for medium-sized data applications.
- DB2 is the commercial RDBMS from IBM. DB2 is the successor of IBM System R, the very first relational database management system. It exists in a LUW version (for Linux, Unix, and Windows) and a version for z/OS, which is an operating system for mainframe computers by IBM.
- SQLite is a DBMS in form of a library, available for a variety of programming languages. All database management systems listed above run as a server process. Applications can connect to the database via a port (e.g., 3306 is the default port for MySQL). An SQLite database is just a file. Via programming libraries, application developers can access this file by writing SQL queries. The set of features in SQLite is very limited, and SQLite is not very fast. Thus, it is not well-suited for storing big datasets. It's best for simple applications that need a small database, for example, a mobile-phone app or a simple web application.

Every database management system offers administration tools, either via a GUI or via command line. Examples are the graphical SQL editors phpMyAdmin or MySQL workbench for MySQL and MariaDB, or universal database clients like DBeaver, DB Visualizer, or Datagrip. The latter can connect to different systems from different database vendors.

2.4 NoSQL Databases

In some applications, relational databases have specific weaknesses, motivating developers to create a new family of database management systems called NoSQL databases (Connolly & Begg, 2005). NoSQL, originally meaning "non-SQL", as a convention for database systems that go beyond the relational model and its query language SQL, nowadays is mostly interpreted as "Not only SQL". NoSQL databases can have traits of relational database systems, but will generally use a different data model and query language (Sadalage & Fowler, 2013). One weakness of relational databases that also fits the four Vs of big data (volume, variety, velocity, and veracity) is that they have a fixed schema. Big data often needs heterogeneous or flexible schemas ("variety").

Another objective for NoSQL databases is a demand for complex storage structures like graphs or JSON documents. And lastly, one of the most important properties of NoSQL databases is scalability. Many NoSQL databases use a data model that is well-suited for distributing data items across a cluster of multiple nodes. So when the amount of data

grows or the performance is not sufficient anymore, it is easy to fix by simply adding more nodes to a cluster. Especially in the cloud, this is very easy to do. That being said, many modern relational databases also scale horizontally.

The following four categories are often used to categorize typical NoSQL databases (Perkins et al., 2018), each of which we will discuss in more depth later in this unit.

- Key-Value Stores: The database is a set of key-value pairs, like a map in programming languages. The values can be atomic values or complex values like a list or a set. The underlying system is optimized for key-based access. Example systems include Redis, Riak, Amazon DynamoDB (DeCandia et al., 2007).
- Wide-Column Stores: A table in a wide-column store has no fixed column schema. Each row in such a table can have arbitrary columns. Example systems include HBase, Cassandra, Microsoft Azure Table Storage, Google Cloud Bigtable.
- Document Stores: A collection in a document database can store arbitrarily structured documents, often using the JSON format. These documents consist of attribute-value pairs, and values can be an atomic value or a list or sub-document. Example systems include MongoDB, Couchbase, CouchDB, Google Cloud Firestore.
- Graph Stores: A graph database stores nodes and edges between these nodes. Both nodes and edges can have labels, and both can save so-called properties. A property is an attribute-value pair. Via programming APIs and graph-based query languages, it is possible to efficiently traverse through the graph and perform graph algorithms like breadth-first search or finding shortest paths. Example systems include Neo4J and TinkerGraph.

There are many database management systems that belong to multiple categories. They are called multi-model databases. Some examples of these systems are ArangoDB, Microsoft Azure Cosmos DB, or OrientDB. They can store graphs, documents, key-value pairs, and more. Most NoSQL databases are either open-source or part of a cloud platform like Amazon AWS, Microsoft Azure, or Google Cloud.

When we concentrate on the first three categories of NoSQL databases—key-value, wide-column, and document stores—what they all have in common is that a data item does not relate to other items in the same or another collection. In a graph database, this is not the case because nodes are connected via edges. In a relational database, this is also not true because a row in a table can have foreign-key references to other rows. In the literature, the three categories of key-value, wide-column, and document stores are also called aggregate-oriented systems. Aggregate means that an item is stored together with everything that belongs to that item. There is always an ID that uniquely identifies an item. In key-value stores, this ID is the key; in wide-column stores, there is a row ID; and in document stores, there is a document id. This concept makes it easy to distribute items across multiple machines in a cluster, and therefore most NoSQL databases are distributed databases.

Partitioning

Partitioning is the process of distributing data items across multiple machines. Some systems, mostly relational databases, call it sharding.

When there are, for example, one million key-value pairs in a key-value store, and there are five nodes in a cluster, each node will store 200,000 items on average. There are two popular approaches to how partitioning can be done, namely range partitioning and hash partitioning.

- Range partitioning: Each node in the cluster is responsible for a specific key range. When we want to store the key-value pair ("KKK", 5), and node 2 is responsible for the key range [J, N], then this node 2 will store that key-value pair. And when a search query wants to find the value of the key-value pair with key "CXY", this query can be routed to node 1, because this node is responsible for all keys in the range [C, I]. Repartitioning is done when a new node is added to the cluster, or a node is removed, or when one node stores many more items than the other. This can be done by adjusting the key ranges for a node and then simply moving the rows with the specific keys to another node.
- Hash partitioning: A hash function takes as an input the ID of a data item and returns the number of the node where this item should be stored. For example, let $h(x = x \ mod \ 5)$ be the hash function and a document with document ID 207 should be stored, $h(207) = 2$, which means that this document is stored on node 2. This hash function is not a good example, because nearly all items have to be moved to a different node when the number of nodes in the cluster increases or decreases. As a solution, consistent hashing (DeCandia et al., 2007) is an approach where the range of the hash function is much higher than the number of nodes, e.g., $h(x) = 82261x \ mod \ 2^{32}$. The same hash function can now be used to compute a hash of each node itself, for example with x being the MAC address of the ethernet adapter of that node. When a new key-value pair should be saved, it is stored in that node which has the closest hash value.

BASE and the CAP Theorem

The consistency model BASE is often used as an opposite to the ACID paradigm introduced earlier. BASE stands for "basically available, soft state, and eventually consistent" (Bernstein & Das 2013). It states that there is no guarantee that the system is always responding to each query, and that the values within the database can be in an intermediate and inconsistent state. Eventually consistent means that after some time, a consistent state is reached again, but we don't get the same guarantees as in transactional systems.

The CAP theorem is often used to classify database management systems based on their consistency model (Fox & Brewer, 1999). The theorem says that each system can only provide at most two of the following three properties: C, A, or P.

- Consistency: Each read operation is answered with the most current value or an error.
- Availability: Each query is answered, but not with an error.
- Partition tolerance: The system is still working even if there is a node failure or package loss in the network.

Traditionally (and in some cases still today), relational database management systems simply ran on one single server, so partition tolerance was not an issue. Therefore, these systems offer consistency and availability (CA).

In contrast, a distributed database (NoSQL or relational database), is never safe from network or node failures. In case of such a failure, a system like that must decide whether to guarantee consistency or availability—you can't have both.

A CP system wants to ensure to never return values that are out of date. But when the most recent value is stored on a node which is currently unavailable, the whole system is unavailable. An AP system does not tolerate unavailability. In the same scenario where a network failure happens, the system would simply return an outdated value, but would do so immediately.

Most NoSQL databases implement either CP or AP. In some systems, users can configure this setting depending on their application requirements. For example, in a social network or a video platform, availability is more important than always returning the most recent and consistent values. In more critical applications, the opposite might be the case.

Key-Value Stores

One example system of the category key-value stores is the open-source database Redis. It was published in 2009 and is one of the most popular NoSQL DBMSs. The Redis data model is a large database of key-value pairs. Both the key and the value can be any binary data, for example, a number, a string, or an image. Applications interact with the database using the two main operations: GET and SET.

```
SET x 5

GET x
```

The second command returns the value 5, which was initially set in the first command. Like maps or associative arrays in programming languages, the SET command updates a value if the key already exists and inserts a new key-value pair if not.

In addition to simple binary values, Redis offers complex data types that can be used as values as well. Here are some examples:

- Lists are an ordered collection of elements. Each element has a position. `LPUSH` can be used to push a new element on the left side of a list, `RPUSH` adds something on the right. At the same time, `LPOP` and `RPOP` remove elements from the list and return their values. Lists in Redis are often used as queues, e.g., in message queueing systems.
- Sets are an unordered collection of elements without duplicates. `SADD` adds an element to a list and returns 1 when it was successfully added or 0 if this value was already there. Sets can be used, for example, on a video platform to store all IP addresses of users who watched a specific video in order to avoid a view being counted twice.
- Sorted sets are sets of elements where each element is attached with a score. `ZADD` adds a new element with a given score; `ZINC` increments the score. `ZRANK` returns the rank of an element. Rank 0 means that there are no other elements with a lower score. `ZRANGE` returns the elements whose ranks are in a specific range. Sorted sets can be used, for example, in a web shop, to efficiently find the ten most popular items in each category.

- Geodata is a set of elements where each element has a geo-position consisting of a longitude and latitude value. `GEORADIUS` finds all elements within a given circle; `GEODIST` can compute the distance between two elements.

Due to optimized storage structures, most Redis commands can be executed in constant or logarithmic time. This makes Redis high performant even if there are billions or trillions of key-value pairs and millions of elements within a list, set, or sorted set.

Redis is typically not used as a full alternative to a relational database, but rather as an addition for special use cases. For example, items on an online shop can be stored in a PostgreSQL database, and the click counters for each item are within a Redis database. Another typical use case is using Redis as a cache. Adding an item into the cache, checking whether an item with a specific key is in the cache, and returning that item can be done in Redis in constant time.

Another factor that makes Redis very fast is the fact that Redis is an in-memory database. The whole database is always in RAM and must fit into memory. To avoid data loss when the system crashes, Redis can be configured to store snapshots of the whole database in regular time intervals on disk (for example, every minute), but only if there are more than 1,000 changes. Another persistency setting is using an append-only file. In such a file, it is possible to log every single command to definitively avoid data loss. When the system crashes and restarts again, the most recent snapshot together with the append-only file is used to restore the database state.

Redis supports replication and sharding. Replication means that one node stores a full copy of the data of another node. In Redis, one machine is the replication master, and the others are slaves. A write operation is always sent to a master node, which forwards this to its slaves. When applications require strong consistency, they also need to read from the master. When eventual consistency is enough, they can also read from a slave, which is often faster as they are not as utilized as much as the master. But it can happen that the slaves are not yet up to date and return **stale data**.

Stale data
This is when read access to data returns an outdated value.

Sharding is realized in Redis via query routing. Each Redis node in a cluster is responsible for a specific key range. As each read and write access to the data always happens key-wise, a query routing mechanism can forward each query to the corresponding node.

Wide-Column Stores

Wide-column stores often look quite similar to relational databases. Their data models are tables which consist of rows and columns. The wide-column store Cassandra uses a query language that is very similar to SQL.

The wide-column store HBase uses a PUT/GET interface which is similar to the Redis examples from above. Each PUT or GET request contains a row-id. Here are some examples:

Code

```
CREATE 'products', 'details', 'similar_products'
PUT 'products', '5', 'details:description', 'Comic book'
PUT 'products', '5', 'details:price', '8.95'
PUT 'products', '5', 'details:pages', '104'
PUT 'products', 5', 'similar_products:77', ''
```

The table that is created with the one inserted row looks as follows:

row-id	details			similar_products
5	description	price	pages	77
	Comic book	8.95	104	

Other rows in the product table can have different columns. The only fixed thing at table-creation time are the column families. The example table from above has the two column families, `details` and `similar_products`. In the `details` family, arbitrary properties of a product can be stored, for example, the number of pages of a book or the weight of a chocolate bar. Each product row can have different columns here. The column family `similar_products` can be used to store references to other products. Google used their wide-column store Bigtable to store details about websites in one column family, and the list of outgoing links to other websites in another column family. Algorithms can use these data as an input to perform complex big data analytics.

HBase supports advanced configurations that can be set for every column family. Versioning automatically stores a history of older column values. This makes it possible to analyze changes or access former values. Each column value in HBase is always attached with a timestamp of when it was written. Another setting is called `TTL`, which stands for time-to-live. All data in a column family with the setting `TTL=3600` will be automatically deleted after one hour, which has 3600 seconds.

HBase only offers one single generic data type, namely byte arrays. Each row-id, column name, and value are just arbitrary arrays of bytes. The applications that work with an HBase database have to convert data into byte arrays first before they write it, and they have to convert it back into the actual data types when reading it.

HBase uses the Hadoop Distributed File System (HDFS) to store its data on a cluster of storage nodes. Range partitioning is used to distribute the rows across the nodes according to their row-ids. A master node called HMaster accepts queries and forwards the requests to what is known as a RegionServer, which is responsible for a given row-id.

Document Stores

Document stores can be used as a full alternative to a relational database. They allow for storing arbitrary structured data, typically in the form of JSON documents, and they offer powerful query mechanisms, indexes, and high scalability.

The most popular document database—and also the most popular NoSQL database—is MongoDB. A MongoDB database consists of what are called collections. Each collection stores documents which are similar to JSON. Internally, MongoDB uses an optimized storage format called BSON (binary JSON) which enables a quicker search.

The following example shows how a document in MongoDB is inserted into a collection and how to update and retrieve it.

Code

```
db.products.insert(
 { "_id": 5,
   "description": "Comic book",
   "price": 8.95,
   "pages": 104,
   "publisher": {
          "name": "Bookboy",
          "country": "USA"
   },
   "similar_products": [77, 108, 12]
 }
)
db.products.update( {"_id": 5 },
                    {"$set": {"price": 9.95}} )
db.products.find( { "_id": 5 } )
```

The document that is stored here consists of six attributes: `_id`, `description`, `price`, `pages`, `publisher`, and `similar_products`. The `_id` is the identifier for each object. It is unique within a collection. If it is not set when inserting a document, MongoDB automatically generates a unique ObjectID. The other five attributes in the example document use the JSON data types string, number, document, and list. Other accepted value data types are Boolean, null, and some special MongoDB types for dates, timestamps, geo data, and more.

Nesting data via lists and sub-documents makes it possible to store everything that belongs together within one single document. In the example above, it would also be possible to store an array of ratings for each product within the document. So, instead of using foreign keys and joins—as it is done in relational databases—the best practice in MongoDB is embedding data within the document itself.

A MongoDB cluster consists of a replica set. This is the set of nodes that stores a redundant copy of the full database. When reading or writing documents, applications can set a read preference and a write concern. The read preference “primary” guarantees strong consistency as it only reads from the primary node. The primary is analog to the replication master in HBase. But in MongoDB, any node within a replica set can become the primary. When the primary node fails, one of the secondary nodes becomes the new primary.

Sharding in MongoDB uses range partitioning by default, but it can also be configured to apply hash partitioning. The concept of a shard key allows users to specify the attribute by which MongoDB distributes documents across the nodes. The default shard key is the `_id`, but if we used the field "`publisher.name`" in the example collection from above, all products that have the same publisher name would be stored on the same node. This would be useful when product searches filter by this field, or when aggregations and groupings are done per publisher.

The MongoDB aggregation pipeline is a feature that allows analysts to define a pipeline that consists of multiple data-processing steps. The following aggregation-pipeline expression counts the number of products for each publisher that cost more than five dollars:

Code

```
db.products.aggregate( [
  { "$match": { "price" : { "$gt":5} } },
  { "$group": { "_id": "$publisher",
                "num_products": {"$sum":1} } }
])
```

Here, pipeline consists of the two steps `$match` and `$group`. Other steps are `$unwind` for unwinding lists, `$sort`, `$limit`, and more. MongoDB utilizes the full cluster performance when executing aggregation-pipeline jobs.

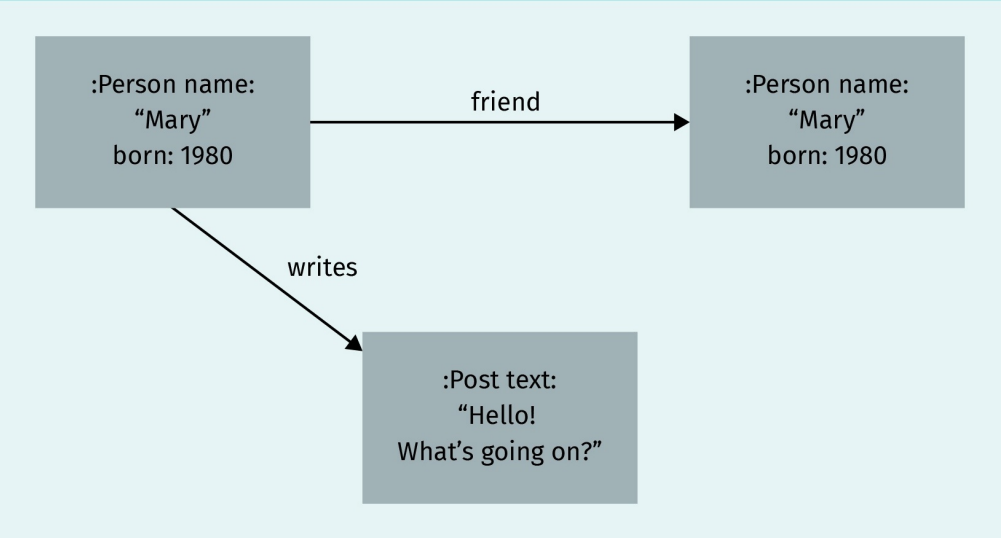
Graph Stores

The data model of graph databases is very complex due to the connections between items. This also makes it hard to distribute a graph across multiple nodes in a cluster.

In math, a graph consists of a set of nodes and a set of edges. An edge is called a directed edge when its direction matters, otherwise it is an undirected edge. For example, "Peter is the father of Mary" describes a directed edge. "Peter is married to Conny" is undirected because it is also true in the opposite direction, as marriage is bidirectional. Many graph databases only support directed edges. Here, the best practice is storing the marriage relationship in any direction, and when writing the query "Who is Peter's wife?" both incoming and outgoing edges are considered.

The most popular graph database management system is Neo4J. It is available as an open-source version and a commercial version with advanced features. Neo4J supports transactions and it is fully ACID compliant. The data model of Neo4J are property graphs. Each node in the graph can have one or more labels, also called node types, and arbitrary properties. Properties are key-value pairs that store the data within the nodes. Two nodes can be connected by a directed edge. Edges have a label, and they can also store properties. The following example shows a property graph that consists of three nodes and three edges.

Figure 1: Property Graph with Three Nodes and Three Edges



Source: Pumperla (2020).

Neo4J’s query language is called Cypher. Commands for creating the graph above in Cypher look as follows:

Code

```
CREATE (a:Person {name:"Mary", born:1980}),
(b:Person {name:"Conny"}),
(c:Post {text: "Hello! What's going on?" }),
(a)-[:friend]->(b),
(a)-[:writes]->(c)
```

Parentheses in Cypher indicate nodes, with lines and square brackets used for edges, and properties written in curly brackets. Before a colon, an optional variable can be introduced which is not stored in the graph, it is only valid within the Cypher command. After the colon symbol, the node type or edge label is written.

The following Cypher query finds all friends of Mary:

Code

```
MATCH (a:Person) -[:friend] - (b:Person)
WHERE a.name = 'Mary'
RETURN b;
```

The `MATCH` clause in Cypher searches for a specific pattern within the graph. The pattern here is two “`person`” nodes connected by a “`friend`” edge. As no direction of the edge is specified, both incoming and outgoing edges are considered. For each match that is found in the graph, the `WHERE` predicate is evaluated, so that in the example above, the variable

b is a “person” node that we are looking for. The clause RETURN b will output this result set of nodes. If we wrote RETURNb.name, the query result would be a table with just the name column of Mary’s friends.

As an alternative to Cypher, the generic graph query language Gremlin can be used (Rodriguez, 2015). While the idea behind Cypher is a pattern-matching approach, Gremlin queries consist of graph-traversal steps. The query from above, which finds all friends of Mary, looks as follows in Gremlin:

Code

```
g.V().has("Person", "name", "Mary")
    .both("friend")
    .hasLabel("Person")
```

The query starts with the set of all nodes in the graph g.V(), where the “V” stands for vertices, an alternative name for nodes. This set is then filtered with a “has” predicate on both the node label and the “name” property. Up to this point, we are in the “person” node that represents “Mary”. The “both” step navigates to neighboring nodes via “friend” edges. An “out” or “in” step would consider only outgoing and incoming edges, but “both” traverses in both directions. As a result of this “both” step, nodes are found which we finally filter with a “hasLabel” predicate to guarantee that the result nodes have the “person” label. Chaining predicates as in Gremlin often results in more human-readable queries, especially when queries are more complex.

2.5 NewSQL Databases

There is a common misconception regarding relational and NoSQL databases. Many sources report that relational databases are ACID compliant and only scale vertically, while NoSQL databases implement BASE guarantees and are horizontally scalable. There are historical reasons for this misconception.

When relational databases were first developed, databases were typically run on individual servers. Consequently, ACID guarantees were provided and these servers were vertically scalable through advanced hardware. When NoSQL databases were developed a few years ago, this standard architecture changed. Instead of running databases on individual servers, the data was typically distributed across multiple machines. This brought the advantages of being able store larger amounts of data and to process data in parallel on multiple machines as well as a shorter geographical distance between application and data storage, leading to shorter latencies and higher availability due to a global distribution of servers. In addition, these systems could be scaled horizontally by adding more machines, i.e. nodes, to the cluster. This distributed architecture has been the standard in almost all NoSQL solutions from the beginning. However, distributed architectures are not exclusive to NoSQL databases. Most relational databases today are also operated as distributed systems.

To illustrate that a relational database runs on a cluster with multiple nodes, the term NewSQL was used for a while. One of the first databases to introduce this principle was VoltDB. However, the term NewSQL has not gained widespread acceptance. Since almost all relational as well as NoSQL databases today can be operated as single server setups (running on single machines) as well as distributed systems, the distinction between traditional relational databases and the so-called NewSQL databases has become superfluous.

The boundaries between relational and NoSQL databases are also blurring increasingly in other respects. For example, multi-model databases can be addressed with multiple APIs regardless of their internal database model. Thus, we can query one and the same database system with SQL or store our data as key-value pairs or documents.

In terms of the CAP theorem, most distributed systems today, be it a relational or NoSQL database, can be set to different levels between availability and consistency. Since in distributed systems partition tolerance must be given by definition, in practice, the only choice is between maximum availability or strict consistency. When deploying a database, we choose between these two extremes.

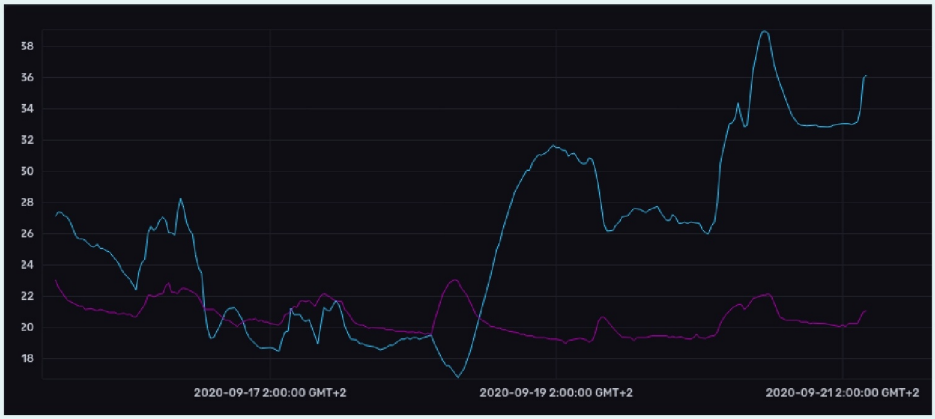
2.6 Time Series Databases

A time series is a sequence of data points captured from one or more sources (e.g., temperature and humidity sensors) over a defined time period. Each data point represents a measurement taken at a certain point in time. In most cases, the data points are spaced equally throughout time. In other words, the temporal distance between two successive measurements is constant.

Time series data come in different shapes and sizes. Datasets consisting of only one value per point in time are called univariate. On the other hand, multivariate time series contain two or more values per data point. Multivariate datasets offer the possibility of finding correlations across multiple different dimensions of measurement over the course of time (e.g., finding a correlation between temperature and humidity over a given time period) (Dunning & Friedman, 2014).

When visualized, time series data give us a graph where one axis (usually the X-axis) represents the flow of time while the other axis represents the actual measurements. For example, the following graph shows a multivariate time series:

Figure 2: Graphical Plot of a Multivariate Time Series



Source: Pumperla (2020).

The purple time series represents the temperature of a specific room over the course of one week, while the blue line displays the humidity level. This dataset will serve as an example dataset for working with the real-world time series database InfluxDB.

Use Cases

Generally speaking, time series are useful for applications where the timing of measured events is of particular importance. In such applications, time-range queries tend to be the most prevalent way of retrieving information from the underlying data store. For example, one might be interested in the average value of all measurements captured within the last 24 hours or the last calendar year. Both of these queries are based on time ranges.

Besides storing and summarizing time series data, another primary area of working with time series consists in analyzing data from the past in order to make predictions about the future. For example, meteorologists use techniques of time series analysis to forecast the weather based on historical data, while seismologists apply the same methods to mathematically model elastic waves propagating through the Earth (Dunning & Friedman, 2014).

Additional real-life applications can be found in other fields where the measurement and interpretation of trends and tendencies is of vital importance. For instance, the black box installed in aircrafts is essentially a device to capture multivariate time series data generated by an airplane's multitude of sensors.

Representing Time Series

The choice of a data store solution for a given time series dataset should be based on several questions that characterize the specific properties of the dataset. The following questions are a good starting point to better understand the data at hand:

- What data types need to be stored?

- How many separate time series need to be stored?
- At what pace are data being acquired? In other words, what volume of data per time unit must be handled by the data store?

File-based formats

There is only a limited number of use cases for which storing time series data in a file-based format is useful. Two criteria should be met for flat file storage to be practical. First, the number of time series at hand should be limited to only a few. Second, the time ranges to be analyzed should be comparatively wide with regards to the partitioning size of the file storing the data.

If these criteria do not apply, it inevitably means that only a small percentage of the available data inside a single file is queried at any given time. Trying to address this problem by reducing the number of time series per file or lowering the file partition size usually results in performance issues, since handling a growing volume of small files becomes increasingly inefficient.

Even though many newly created databases are initially implemented using flat files, especially during early development, you are well-advised to switch to a real database solution in the long run. This becomes especially evident when we look at the requirements of a constantly growing database, which will eventually exceed the constraints of flat file database systems (Yusof & Man, 2018).

Figure 3: Example Excerpt of a CSV Flat File Storing Time Series Measurements

flat_file.csv

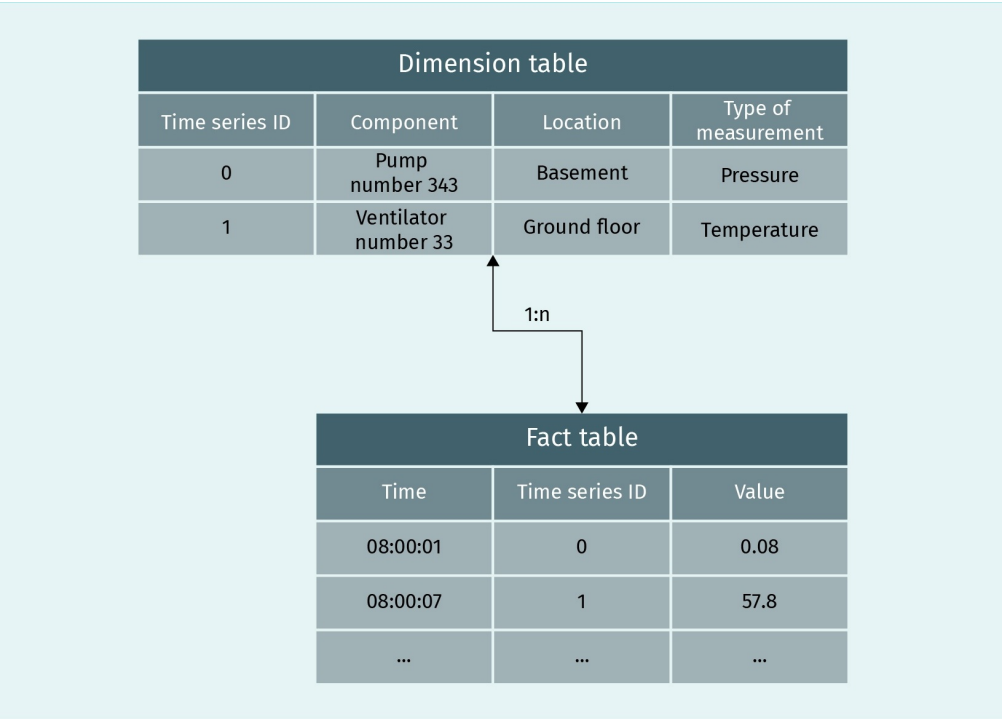
```
“Time”, “Time series ID”, “Value”
“08:00:12”, “0”, 0.01”
“08:00:15”, “1”, “18.17”
“08:00:17”, “1”, “18.13”
...
```

Source: Pumperla (2020).

Time series data in relational databases

Storing multiple time series in a single relational database can be achieved by using a star schema, which consists of one fact table and several dimension tables connected to it. The name comes from its geometric configuration, with the fact table in the middle, surrounded by the dimension tables, schematically resembling a star. In this setup, the fact table is responsible for storing the times and corresponding measurements of a time series in a chronological order.

Figure 4: Example of a Relational Database Scheme for Time Series



Source: Pumperla (2020).

The fact table stores measurements taken by sensors inside different components in a factory. The measurements in the fact table are associated with their respective components by a time series ID, while the relatively unchanging dimension table stores further information about each component.

In addition, the fact table also stores the IDs of the time series that each measurement belongs to. These IDs are essentially foreign keys to the dimension tables, which store further information about the time series that are represented by a given ID. In this way, each measurement that is stored in the fact table can be attributed to a specific time series.

The star schema can store data points up to a magnitude of hundreds of millions or even billions. However, serious performance problems arise once data are being fetched or processed, because modern applications (e.g., machine-learning systems) require instant access to millions of data points per second. While the lower end of these fast-paced database operations is in fact achievable for relational database systems, the complexity increases fast with growing amounts of data (Dunning & Friedman, 2014).

Figure 5: Example of a Fact Table in a Relational Time-Series Database

Example of a Fact Table in a Relational Time-Series Database		
Time	Time series ID	Value
08:00:12	0	0.01
08:00:15	1	18.17
08:00:17	1	18.13
08:00:21	0	0.02
08:00:22	2	50.67
08:00:24	1	18.16
08:00:30	0	0.01

Source: Pumperla (2020).

The fact table stores all measurements along with their respective time series ID, which represents the sensor or device that took the measurement. Further information about each time series ID (respective to each sensor) is stored in a dimension table.

Time series databases

Time series have to be stored in a way that is specifically optimized for time range queries. The properties and requirements of time series datasets narrow the list of database systems best suited for the job. These requirements can be boiled down to volume and flexibility. Regarding volume, time-series data are often characterized by a vast and ever-increasing volume of data that must be stored and processed, especially since the explosive proliferation of sensors in times of the Internet of Things (IoT). This immediately becomes evident when we look at an example application for time series data, like public transport forecasts or planning, where huge amounts of data are collected by Automatic Data Collection (ADC) systems, e.g., Automatic Passenger Count, Automatic Fare Collection, and Automatic Vehicle Location systems. These data are then used to create prediction models based on different approaches, like machine learning.

For the second requirement, flexibility, we just need to recall emerging techniques and methods of data analytics which promise unheard-of insights, and which sometimes require unstructured or semi-structured data (e.g., social media posts). These requirements suggest the use of NoSQL database solutions over traditional relational databases, because they best fit the requirements of horizontal scalability coupled with high performance, while trading off features like automatic enforcement of referential integrity (i.e., strict foreign-key relationships) and transactions, which are core features of SQL.

The flexibility aspect of NoSQL is a result of its dynamic schemas, which stand in contrast to the static schemas found in relational databases. These dynamic schemas allow the use of unstructured or denormalized data and make it easy to modify schemas in a database

when faced with changing data and requirements. Consequently, NoSQL databases can be anything from documents to graph databases and can be easily modified, while the stricter, table-based structure of relational databases makes it an extremely time-intensive task to modify the schemas. Further flexibility is added through horizontal scaling, which allows us to scale over cheap, commodity servers, instead of vertically scaling by upgrade an existing server or buying a better one, as is typical in relational database systems.

Moreover, some NoSQL database systems, like Apache HBase, support the use of wide tables, which essentially means that a database table can have a virtually unlimited number of columns. This feature offers a solution to one of the core issues associated with the star schema, which is its excessive use of rows, namely, one entire row for each measurement. With the help of wide tables, each row can store a multitude of measurements.

Conventionally, each row represents a time window that stores 100 to 1,000 measurements occurring within that interval. This leads to a performance boost, since the rate at which data can be fetched is partially dependent on the number of rows that are being read. And since some database systems, like HBase, have a disk storage policy that strings together data on disk based on its primary key, the retrieval of data from any given time series becomes even more efficient, because fragmentation of data is avoided.

Figure 6: Example of a Wide Table in a NoSQL Database

Example of a Wide Table in a NoSQL Database

Time window	Time series ID	Time offset within time window with corresponding values					
		"+1"	"+4"	"+10"	"+18"	"+22"	"+33"
08:00:00	0	0.01			0.03		
08:01:00	0		0.02				
08:00:00	1	18.45		22.33		23.43	
08:03:00	2						104.56

Source: Pumperla (2020).

The use of a wide table with time windows instead of a single row for each measurement reduces the number of rows significantly.

A further performance enhancement can be achieved by using hybrid-style time series databases. In these database systems, the measurements of all columns are compressed into one super-column, commonly referred to as a **blob**. This compression operation is done by the hourly accumulator (blob maker) whenever a time window ends. By compressing many columns into a blob, the amount of data that have to be fetched is reduced significantly. Moreover, a reduction of columns effectively reduces the per-column overhead that must be considered when using HBase and its on-disk format (Dunning & Friedman, 2014).

Another refinement of hybrid-style database design can be achieved by using the direct blob insertion design. The idea behind direct blob insertion is to cache newly arriving measurements in memory instead of intermediately storing each measurement in a separate database column. In this way, the bottleneck of storing measurements in database columns is bypassed, and the blob is generated directly from memory after each time window ends.

Blob
The term blob refers to binary data stored within some data management system such as a database. Usually, the data store is not aware of the exact contents of a blob, making it impossible to query for information stored within a blob.

Time series databases are a special case of NoSQL databases that can be defined as a way to store multiple time series such that queries to retrieve data from one or more time series for a given time range are particularly performant. Some common characteristics and features of time series databases include

- write once read many (WORM). Saved objects typically don't change after they have been added to the database.
- compression. Time series databases typically utilize compression algorithms to manage the data efficiently.
- efficient data summarization. Time series databases are optimized for requesting summaries of data spanning long periods of time.
- data lifecycle management. Many time series databases can transparently down-sample or invalidate old data that are no longer useful or keep high-precision data around for a limited amount of time. With conventional databases, managing old or high-precision data has to be implemented manually by the application developer.

Several database systems optimized for handling time series data have been developed by the open-source community. The following sections will focus on InfluxDB, arguably the most widespread solution in use today.

InfluxDB

InfluxDB operates on a data structure composed of buckets, measurements, series, and data points (InfluxDB, 2020). A measurement is like a table in an RDBMS, where the primary index is pre-configured to a **UNIX epoch** timestamp. Each data point comprises a timestamp and several key-value pairs collectively referred to as the "fieldset". In addition to the fieldset, each data point optionally contains a number of tags called the "tagset". Like fields, tags also consist of a key and value component. Field and tag values can be integers, floating-point numbers, strings, and Boolean values.

UNIX epoch
A UNIX epoch refers to the earliest possible time value in the UNIX Epoch time system. In this system, a timestamp is the number of seconds that have passed since the UNIX epoch on January 1, 1970.

The major technical difference between fields and tags is that fields aren't indexed. This means that queries that filter a dataset based on field values must sequentially scan all data points to match them against the query conditions. Although tags are indexed and therefore offer faster filtering, they have the downside of increased RAM requirements. As a rule of thumb, you should persist variable classes of information as fields and more limited value spaces as tags.

A series is a set of data points defined by grouping together points based on their tagsets. That is, a series can be viewed as a container for several data points that belong together. In turn, a measurement serves as a container for a set of series and is defined by a simple string identifier. A measurement's string identifier should accurately represent the domain of the data.

Hands-On Use Case: Occupancy Detection

Occupancy detection refers to the task of detecting whether a room is occupied or not based on a series of measurements derived from sensors in the room. Multiple research teams have released datasets for developing and testing binary classification systems for predicting room occupancy from temperature, humidity, light, CO2 levels, and other sensor data.

In this section you will see the process of importing an occupancy dataset into an InfluxDB instance. In order to run this section's code on your system, you have to install Python 3.

Development setup

Start by signing up for an InfluxDB 2.0 Cloud account. Enter your credentials or authenticate with your Google account and then choose the free plan. When asked to select a provider and region, go for the Amazon Web Services (AWS) option based in Frankfurt.

After logging into your new account, click on the "Data" icon to the left. Next, click on the "Token" tab and generate an all-access token called "Occupancy Detection". Copy the token to your clipboard for later use. Then, switch to the "Buckets" tab and create a new bucket called "Occupancy Detection" as well.

Proceed by creating a new directory on your computer. Open this directory in a terminal and run `pip install influxdb-client` to install the InfluxDB Python client on your system. After the installation has finished, go ahead and download the UCI Occupancy Detection dataset.

Unzip the dataset archive to the directory you just created. Change to the resulting directory and open the `datatraining.txt` file to inspect its format. As you can see, the format consists of eight columns of comma-separated values. Each data point has an ID (first column), a timestamp, temperature value in degrees Celsius, and other sensor measurement values.

Importing the data into InfluxDB

Create a file called load_data.py and open it in a text editor. Paste the following code into the file and adjust the connection details (token, org and bucket) as needed (we're using Python version 3.6 or higher throughout this example):

Code

```
from datetime import datetime, timedelta

from influxdb_client import InfluxDBClient, Point, WritePrecision
from influxdb_client.client.write_api import SYNCHRONOUS


def extract_cols(row):
   cols = row.strip().split(",")
   cols[0] = int(cols[0].strip('"'))
   cols[2:7] = [float(col) for col in cols[2:7]]
   cols[7] = bool(int(cols[7]))
   return cols


def create_data_point(row, time):
   cols = extract_cols(row)
   data_point = Point("occupancy") \
       .field("temperature", cols[2]) \
       .field("humidity", cols[3]) \
       .field("light", cols[4]) \
       .field("co2", cols[5]) \
       .field("humidity_ratio", cols[6]) \
       .field("occupancy", cols[7]) \
       .time(time, WritePrecision.NS)
   return data_point


def load_data(client):
   one_week_ago = datetime.now() - timedelta(weeks=1)
   with open('occupancy_data/datatraining.txt') as occupancy_data_fp:
       next(occupancy_data_fp) # skip header row
       all_data_points = []
       for i, row in enumerate(occupancy_data_fp):
           time = one_week_ago + timedelta(minutes=i)
           all_data_points.append(create_data_point(row, time))
       client.write_api(write_options=SYNCHRONOUS).write(bucket,
org, all_data_points)

if __name__ == '__main__':
   org = "<your-org>"
   bucket = "Occupancy Detection"
   token = "<your-token>"
   url = "https://eu-central-1-1.aws.cloud2.influxdata.com"

   client = InfluxDBClient(url=url, token=token)
   load_data(client)
```

After you have adapted the script by filling in your connection details, run the script by executing 'python load_data.py' on the terminal. The script should complete its operation very quickly thanks to using the batch write operation.

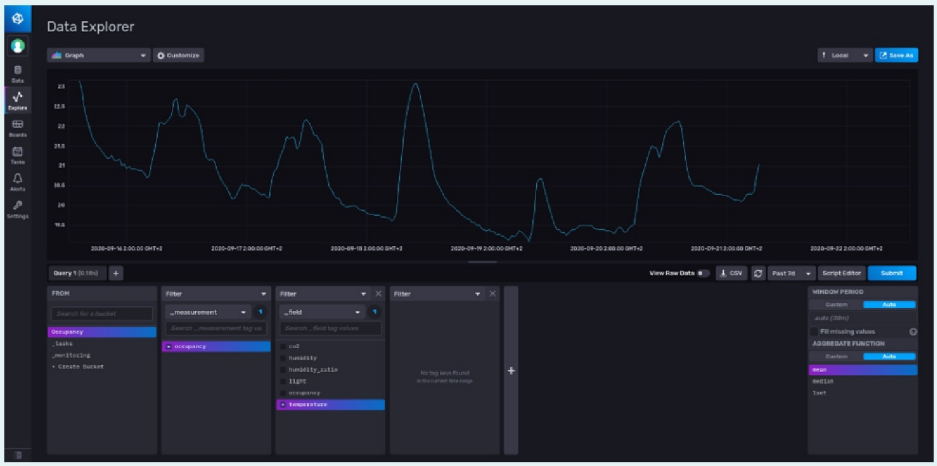
The script first connects to the InfluxDB server and reads the occupancy detection dataset from the local storage device. For each row (i.e., data point) in the dataset file, the script then generates a Point instance provided by the InfluxDB Python client. All generated Point instances are collected in a list. Lastly, this list of data points gets sent to the database server in a single batch. Inserting all data points as part of one batch drastically reduces the transmission time compared to sending each data point by itself.

Note that we are generating our own minute timestamps instead of taking the timestamps in the data file. This is done in order to prevent InfluxDB from automatically deleting the data points right after inserting them. InfluxDB does this because the default retention period for data points is 30 days. That is, data points with a timestamp older than 30 days are periodically deleted.

Interacting with data in InfluxDB

InfluxDB Cloud offers an exploratory time data visualization interface. It automatically plots one or multiple time series into a graph and is a valuable tool to better understand a new dataset. To start using it, log into your InfluxDB Cloud account and select the "Explore" item in the sidebar. Set a time range and additional filters and then press the "Submit" button to visualize the filtered data points:

Figure 7: Screenshot of the InfluxDB Data Explorer Interface



Source: Pumperla (2020).

Besides the visual interface, InfluxDB data can be accessed by means of Flux, a functional data scripting language created for querying InfluxDB databases. The syntax of Flux is largely inspired by JavaScript. Although it looks very different from SQL and other query languages, much of the functionality of SQL is also available in Flux. The following Flux query fetches the 100 most recent data points from last week's data:

Code

```
from(bucket:"Occupancy Detection") \
       |> range(start: -7d) \
       |> filter(fn:(r) => r._measurement == "occupancy") \
       |> filter(fn:(r) => r._field == "temperature") \
       |> tail(n:100) \
       |> yield()
```

In order to run the above query, create a new script file called query_data.py and paste the below code into it:

Code

```
from influxdb_client import InfluxDBClient

def query_data(client, org, bucket):
   query = f'from(bucket:"{bucket}") \
       |> range(start: -7d) \
       |> filter(fn:(r) => r._measurement == "occupancy") \
       |> filter(fn:(r) => r._field == "temperature") \
       |> tail(n:100) \
       |> yield()'
   result = client.query_api().query(org=org, query=query)
   for table in result:
       for record in table:
           print(record)

if __name__ == '__main__':
   org = "<your-org>"
   bucket = "Occupancy Detection"
   token = "<your-token>"
   url = "https://eu-central-1-1.aws.cloud2.influxdata.com"

   client = InfluxDBClient(url=url, token=token)
   query_data(client, org, bucket)
```

Besides the basic functionality such as filtering data points, Flux also features a range of familiar aggregation functions which can be used to summarize a dataset. For example, the Flux mean() function computes the mean or average value of records in a given input table. The below query results in an average temperature of 20.61 degrees Celsius for the entire occupancy detection dataset:

Code

```
from(bucket:"{bucket}")
       |> range(start: -7d)
       |> filter(fn:(r) => r._measurement == "occupancy")
       |> filter(fn:(r) => r._field == "temperature")
       |> mean()
```

Timeseries analysis

With more and more data becoming available thanks to technologies such as InfluxDB, machine learning systems have become viable tools for businesses to derive valuable insights from their data or automate repetitive business processes. As mentioned, time series lend themselves to being analyzed for the purpose of forecasting future developments. One of the most well-known examples would be predicting the weather. Besides forecasting future values of a time series, another task of analyzing time series involves finding outlier data points. In this context, anomaly detection refers to the identification of rare events or other deviant observations represented as data points in a time series (Dunning & Friedman, 2014).

SUMMARY

Relational databases are well-suited for many applications, but especially when facing big data problems, traditional approaches quickly come to their limits. SQL is a powerful language which evolved over time and was extended and optimized by different vendors. But the biggest weaknesses when managing big data are its fixed schema and often bad scalability.

NoSQL databases solve these problems by using flexible schemas and different data models than SQL. These specialized data models also make NoSQL well-suited as a distributed database. This increased degree of flexibility is achieved by loosening some characteristics of relational databases. For example, most NoSQL databases have lower consistency guarantees than relational databases, which means that the same query can lead to different results when issued from a different client.

NewSQL approaches try to combine the benefits of both worlds. They share the same or similar transactional properties with relational databases, are accessed via SQL, and scale better than traditional relational database management systems. As a rule, NoSQL and NewSQL databases achieve higher scalability by sharding each data item across multiple nodes in the cluster.

Time series data has its special requirements, and specialized NoSQL databases, like InfluxDB, handle them quite well.

UNIT 3

DATA FORMATS

STUDY GOALS

On completion of this unit, you will have learned …

- the basics of various text-based and binary data formats.
- the difference between traditional and modern data formats.
- the advantages of column-based formats for modern big data applications.
- the strengths of binary communication protocols for scalable applications.

3. DATA FORMATS

Introduction

The way data are stored has an impact on a variety of aspects in data analysis. Depending on what your priorities are, you might opt to store data in a human-readable fashion or find a very efficient way of compressing data to take up the least amount of storage possible, thereby sacrificing readability and other aspects.

File formats are many and diverse, but the most basic distinction you can make is between text and binary files. Both file types are stored as bytes, ultimately a series of zeros and ones on your machine, but are encoded and represented differently. In text data the bytes represent characters; binary data also include custom data points that don't map to characters at all.

Text files

Text files are written on disk row by row, similarly to how data is stored in traditional databases. A plain text file, often having a ".txt" file extension, can be described as a sequence of characters, where newline characters, also called end-of-line characters (EOL), mark the end of separate lines. To denote the end of a file, plain text uses an end-of-file character (EOF). In the rich text format, often denoted with an ".rtf" file extension, text styles like italics are also part of the format, whereas plain text just contains the text itself.

It is important to understand that since text files are also just sequences of bytes on your computer, they need character encoding so that your applications can interpret and display the correct character representations to you. A widely used character encoding is UTF-8, whose name is derived from Unicode Transformation Format and the fact that it is an 8-bit encoding scheme. UTF-8 powers roughly 95 percent of all web pages (W3Techs, 2020) and is vastly more powerful than simplistic encodings like ASCII, which are not capable of representing any non-standard characters. UTF-8 is backward compatible with ASCII though, meaning that each valid ASCII character is also a UTF-8 character.

UTF-8 uses between one and four bytes (a byte corresponding to eight bits), choosing the minimal number of bits needed to represent a character whenever possible. The more frequent characters in the English language, like basic letters and numbers, are smartly represented with fewer bytes to save space. In total, UTF-8 can represent 1,112,064 characters, and we call each of them a code point. For example, the Unicode code point for the Euro sign ("€") is `U+20AC`, which can be represented by just three bytes in the following binary representation in UTF-8:

Code

```
11101101 10010101 10011100
```

Text files can typically be opened by a text editor and don't need a dedicated application to be read. They are generally difficult to corrupt, meaning that you can manually alter and store them and will still be able to open them again. Text files are very convenient as they don't need special technology to be accessed, but the convenience hides quite a glaring pitfall: If we need to access even the smallest bit of information in the file, the entire file needs to be opened. This means that the entire file has to be loaded into memory. While this is not an issue for small files, it puts a hard limit on the maximum size of a file that can be opened to the amount of RAM of a system, requiring you to split up large files into several smaller ones. This can be seen as a minor, first example of the problems that can arise from working with big data. We'll look into several ubiquitous text file formats in this unit.

Binary files

As opposed to text files, binary files also contain custom bytes of data that don't have any textual representation. Prominent examples of binary formats are any kind of audio, video, or image formats. The following image of a screen of code has been encoded with the JPG image format:

Figure 8: Screen of Code Encoded with JPG

Source: Spiske (2018).

When you try to open this image with a text editor, you will see something like this:

Code

```
ˇÿˇ‡JFIFHHˇ,ICC_PROFILE lcmsmntrRGB XYZ ‹)9acspAPPLˆ÷”-lcmsdesc¸^cprt\
wtpthbkpt|rXYZêgXYZ§bXYZ∏rTRCÃ@gTRCÃ@bTRCÃ@descc2textIXXYZ ˆ÷”-XYZ
3§XYZ o¢8ıêXYZ bô∑Ö´XYZ $†Ñ∂œcurvÀ…cíkˆ?Q4!Ò)ê2;íFQw]Ìkpzâ±ö|¨ıø}”√È0
ˇˇˇ€Ñ
```

The textual representation of the image gives you a seemingly chaotic mess of garbled characters. That's your text editor trying to make sense of custom binary data that can't be read as text. But you can also notice that some parts of the data are in fact textual, for instance, the word `ICC_PROFILE` above.

While binary files require extra effort to be handled, they open potentially unlimited ways to remove the limitations of text files. Binary files allow us to store all sorts of data that are not text and are generally more compact than text files. They also allow us to organize the data in different ways than line-by-line. This allows for, among other things, more efficient storage and a more complex file structure, which can overcome the reading limitation of text files.

File header
A file header is a section of a file, usually at the very beginning, that includes various information on the file. This information is also known as metadata.

Binary files usually include a **header** which contains all sorts of information about the file. The above snippet showing the JPG image in a text editor corresponds to that file's header.

Binary formats in the context of big data

This is particularly useful, as the header usually contains a map of the file structure, so that only specific parts of the file need to be accessed when certain information is needed. This can also help for accessing different parts of a single file simultaneously, therefore helping with read speed. The development of data formats for big data has initially focused on increasing read and write speed, reducing storage size, and storing complex data.

More recently, the focus has shifted to columnar formats. Columnar formats, as the name implies, focus on storing data columns of a table in a contiguous way, instead of focusing on individual rows or records the way relational databases do. In the context of big data, this helps when querying few columns of a very wide table, as only the data of interest (usually a small subset) need to be read. This in turn yields much higher read speed than row-based formats. Apache Arrow and Apache Parquet are two formats that follow this philosophy and that we will investigate more closely in this unit.

Another notable movement in the context of big data is the usage of binary communication protocols, as opposed to text-based protocols, which allow you to store and also transmit data efficiently between several computers. These protocols usually come with a description language which makes it relatively easy to access data stored with them from several programming languages and on any platform. Examples of such binary protocols are Apache Thrift, Apache Avro, and Protocol Buffers.

3.1 Traditional Data Exchange Formats

Before we delve deeper into complex, modern file formats, let's devote some time to traditional formats that are still very relevant today, including both text-based and binary formats.

CSV and TSV

Apart from plain text, comma-separated values (CSV) are among the most widely used text formats you can find. CSV files use commas to separate values in each row of data, hence the name. Although it's not required, each row in a CSV file usually has the same number of values separated by a comma, which makes this format eligible to represent tabular data. Often the first row of data is reserved as a header, which describes columns corresponding to each position in a row. Here's an example of the first three lines of a CSV file containing car data:

Code

```
Year,Make,Model,Description,Price
2004, Mercedes,"A52","car description",34000.00
2011,BMW,"Extra, ""Limited Edition""","",49000.00
```

As you can see, while simple at first glance, there are certain rules a CSV file has to adhere to. For instance, any text field may be in quotation marks, but some of them have to be. If your text field contains a comma or quotation marks, you need to quote the field. Additionally, if your field contains quotation marks you need to represent them with the double-quote character (""). In the above example, to store the text field Extra, `"Limited Edition"` in our CSV file, we need to insert two double-quote characters and quote the whole field. Also note that any whitespace, including spaces, tabs, and line break characters, is part of the field. In the example note the leading space in front of `Mercedes`, which is likely a typo, so this space should be stripped to avoid confusion.

CSVs are so widely used because they have the benefit of usually being very human-readable (although it quickly gets tricky once you have too many "columns" or fields that are very long, spanning multiple lines in your text editor) and the fact that they can represent tabular data. Most relational databases allow you to export their tables as CSV, and you can usually also import them. Commercial software like the spreadsheet tool Excel is based on tabular data, too, and can import and export CSV files as well. Reading tables and spreadsheets is a regular task across most business units in practically any company, which makes CSV an ideal candidate for data exchange and communication. One of its drawbacks is that CSV is not stored very efficiently. Database dumps of large tables can quickly go into several hundred megabytes, which removes many of CSV's advantages again, because computers start to struggle to seamlessly open such large text files.

There are many formats closely related to CSV, all distinguished by which separator they use. One example is that of tab-separated values (TSV), which is functionally equivalent to CSV, but has the benefit of being very readable in the case where all fields in a "column" are of comparable length, so that the tabs roughly align. The above example written in TSV format would look as follows:

Code

```
Year   Make   Model   Description   Price
2004   Mercedes   "A52"   "car description"   34000.00
2011   BMW   "Extra, ""Limited Edition"""  ""   49000.00
```

As files become larger, it usually becomes difficult to "spot" the tabs that separate fields, unless they're extremely standardized, which is one of the reasons why you see CSV files more often in practice.

Textual Markup and Serialization Languages

While CSV files can be great for simple tabular data, they fail to encode more complex data. For instance, if we wanted to include a list of all car owners with their contact details in the example above, the only practical way of doing so would be to create an "owners" column, which squeezes the contact list information into one field, which is not ideal. In general, you want to be able to represent complex, nested structures with your data formats.

The three textual formats we're looking into in this section are XML, JSON, and YAML. While each of those formats were created for different purposes, and see themselves as different in nature, for us they're effectively just text-based data formats in the context of this unit. A complete introduction to the specifics of each format would be beyond the scope of this text, but we can consider crucial aspects of their design

XML

The Extensible Markup Language (XML) format is the oldest of the three formats we're looking at. XML is a **markup language** that was designed for the internet and is a general text-based data format to describe arbitrary data structures.

Markup Language
In text processing, a markup language is a system to annotate text, for instance to style the content or otherwise attribute values to it. HTML is an example of such a language.

Here's an example of an XML file describing a person, which we will use as a running example to compare the different formats:

Code

```
<?xml version="1.0" encoding="UTF-8"?>
<!---This is a comment--->
<person>   <firstName>Jane</firstName>
<lastName>Doe</lastName>   <age>20</age>   <address>
<streetAddress>24 5th Street</streetAddress>
<city>New York</city>        <state>NY</state>
<postalCode>10023</postalCode>   </address>
```

```
<phoneNumbers>       <phoneNumber>
<type>home</type>           <number>213 555-4321</number>
</phoneNumber>       <phoneNumber>
<type>fax</type>           <number>646 555-1234</number>
</phoneNumber>   </phoneNumbers>   <sex>
<type>female</type>   </sex></person>
```

This example displays a few key aspects of XML, as outlined below.

- XML declaration: All XML files start with a declaration as part of the metadata, in this example, stating that we use version 1.0 of the XML language and UTF-8 for encoding purposes.
- XML tags: These tags start with a < and end with a >. You have start-tags like `<age>` and end-tags like `</age>`. There is also a third kind of tag, the empty element tag, which is self-closing, e.g., `</lineBreak>`, to denote a line break in a text document.
- XML elements: The textual data between start and end tags is called an element. For instance, in the above example `female` is the element of the `type` tag.
- Nesting: XML documents can be arbitrarily nested. In the example, you see that we can have several phone numbers within the `phoneNumbers` tag and each `phoneNumber` tag can in itself have different kinds of tags and elements.
- Comments: XML can contain comments, which you have to wrap into a tag with an exclamation mark `<!---->`.
- XML attributes: Within a tag you can specify one or many attributes. For example, the encoding part of the XML declaration, `encoding="UTF-8"`, is an attribute.

Whether you put a value for your data structure into the element of a tag or into an attribute is left up to the application developer. The above example can be equivalently written as follows by moving elements into attributes:

Code

```
<person firstName="Jane" lastName="Doe" age="20">
<address streetAddress="24 5th Street" city="New York" state="NY"
postalCode="10023" /> <phoneNumbers>
<phoneNumber type="home" number="213 555-4321" />
<phoneNumber type="fax" number="646 555-1234" />
</phoneNumbers>   <sex type="female" /></person>
```

This version is more concise, but it is generally recommended not to overuse attributes, as it is said to decrease readability of XML documents, especially when documents become large. It is also important to stress that all indentations used in these examples are purely optional and only used to help you read them. Computers can cope equally well with XML files that are stripped of any whitespace, sacrificing readability for minor improvements in compression.

XML is not as popular as it used to be, but it is still widely used. In particular, XML has established itself as the de-facto standard for advanced word processing system documents such as Apache OpenOffice, LibreOffice, and Microsoft Office, which uses the Open Office XML standard.

JSON

The JavaScript Object Notation (JSON) file format was specified first in the early 2000s but was formalized by ECMA in 2013. JSON originated from the JavaScript programming language, but is designed to be a language-independent exchange format. In JSON, much like in XML, white space and formatting does not matter, but well-formated examples make for better readability. In JSON, an example encoding for a person reads as follows:

Code

```
{
   "first_name": "Jane",
   "last_name": "Doe",
   "age": 20,
   "address": {
       "street_address": "25 5th Street",
       "city": "New York",
       "state": "NY",
       "postal_code": "10023"
   },
   "phone_numbers": [
       {
           "type": "home",
           "number": "213 555-4321"
       },
       {
           "type": "fax",
           "number": "646 555-1234"
       }
   ],
   "sex": {
       "type": "female"
   }
}
```

You can see that JSON does not build on advanced features such as attributes or tags, and instead uses a handful of basic structures to describe data. The basic data types are numbers (integers, or floating-point), strings, boolean (“true” or “false”), or `null` to indicate absence of a value. Additionally, the following advanced data types are allowed:

- Objects are collections of key-value pairs of various types. Keys, like `last_name`, are always text strings, whereas the values can be any of the data types allowed by JSON, including objects (allowing arbitrary nesting of objects). Objects are the cornerstone of JSON and the above example describing a person is a JSON object.

- Arrays are ordered lists of values of heterogenous type. Arrays are described using square brackets, and the elements of an array are separated by commas, like the `phone_numbers` array above, which has two objects in it.

JSON is a ubiquitous format that is used in all sorts of applications and programming languages. To name just two examples, Python's dictionary type seamlessly maps to JSON, and every JSON object can be parsed into a Python dictionary, providing a simple yet effective serialization method for Python practitioners. Also, while NoSQL databases tend to have JSON support anyway, classical relational databases also often come with a JSON type for their fields, which allows them to store JSON data as text fields in a relational schema.

One common criticism of JSON is that it doesn't allow you to write any comments into its documents. When manually creating or inspecting JSON documents, it can be helpful to have comments in it to clarify the content. On the other hand, when you merely use JSON as a data exchange format, humans will never touch these files directly and comments would be of little use anyway.

YAML

The YAML format, first introduced as "Yet Another Markup Language" in 2001, was later renamed to the recursive definition "YAML Ain't Markup Language" for version 1.2, to make the point that it's not just a markup language, but rather a general-purpose data exchange format. Nevertheless, it is comparable in functionality and usage to other markup languages like XML, and version 1.2 of YAML is a strict superset of JSON, meaning that everything that can be encoded in JSON can be realized in YAML as well.

From a practitioner's point of view the biggest advantage of YAML is its high readability—it has a strictly defined indentation system that closely resembles Python, which allows it to do without any braces or brackets. In other words, unlike XML or JSON, formatting does matter with YAML. At the same time, this strict system is one of the most notable disadvantages of YAML, too, since writing in this format can become error prone. Indenting a single line incorrectly can corrupt an entire YAML file. So, if you're working with deeply nested data structures, validating a YAML file can become a tedious task. Below you find our running example of our "person" data structure in YAML:

Code

```
first_name: Jane
last_name: Doe
age: 20
address:
   street_address 24 5th Street
   city: New York
   state: NY
   postal code: '10023'
phone_numbers:
   - type: home
     number: 213 555-4321
```

```
   - type: fax
     number: 646 555-1234
sex:
   type: female
```

The format is so intuitive that it barely needs any explanation. Simply note how closely it resembles the JSON example from earlier, but without the visual clutter (braces, brackets, etc.). Also, note that lists are implemented using dashes ("-").

Traditional Binary Formats

Moving on from text-based formats, we should also mention a few binary formats that are relevant in the context of big data.

BSON

Binary JSON (BSON) has been introduced by the MongoDB team to address a few drawbacks of the JSON file format:

- As a text-based format, reading large JSON files can be slow.
- JSON is human readable, but that comes at the cost of having relatively large file sizes, which can be an issue when the data volume becomes too large in big data scenarios.
- JSON only supports a few basic data types, but we might want to store more complex data, e.g., multi-media data, like images.

BSON sets out to solve all these problems by introducing a binary variant of JSON. BSON and JSON are structurally very close to each other, but "look" different, as humans can't easily inspect binary formats. Let's have a look at a very simple JSON document:

Code

```
{"hello":"world"}
```

In BSON, this would look as follows:

Code

```
\x16\x00\x00\x00         // total document size
\x02                     // type String
hello\x00                // field name "hello"
\x06\x00\x00\x00world\x00     // value and properties
\x00                     // type EOO ('end of object')
```

As you can see, the field names and values ("hello" and "world") are still represented as text in BSON, just the metadata gets transformed into binary. That means when working with text-only data, BSON does not represent much of an uplift in terms of reading speed from an application or when it comes to file sizes. However, when storing binary data in

fields, JSON is not sufficient. It is common for modern big data systems to have a wide variety of data, including binary formats, and BSON is one good way to store your complex data structures, especially when using it with a NoSQL database.

Hierarchical data format

Another important example of an efficient binary data format is the Hierarchical Data Format (HDF). Its latest iteration is HDF5, which has been designed to store large amounts of data. All HDF tools and libraries are available under the open BSD license, and the format is supported by many programming languages, including Python and R.

HDF5 is structured like a folder and file system on your computer, but there are some unique differences in terminology:

- Groups loosely correspond to a folder in our file system analogy. A group either contains datasets or other groups.
- Datasets correspond to files on your file system and contain the actual data. Datasets can be standalone or stored within a group.

HDF has various advantages that make it a worthwhile option to consider as a storage format, including self-describing metadata, efficient compression, and data slicing capabilities.

3.2 Columnar Data Formats

Let's now illustrate the benefits of columnar storage formats by investigating two prime examples and comparing them to each other: Apache Parquet and Apache Arrow.

Apache Parquet

Apache Parquet is a columnar data storage format. It is free and open source. It was built from the ground up to work with fast and efficient big data applications. It was created to support efficient compression, i.e., finding the best possible compromise between storage size and computational effort for decompression (which requires time). It supports nested data structures, multiple data types, and is designed for optimal read speed. It was designed to be compatible with big data analytics frameworks, such as Apache Spark. Furthermore, it is highly compatible with programming languages—it can be used with C++, Java, Python, and more.

Parquet originally started as a joint open-source project between Twitter and Cloudera and was released in 2013. It is built to deal with complex nested data structures and uses the record shredding and assembly algorithm, a technology originally developed by Google. The name Parquet is inspired by the wooden flooring technique parquet, originating from the French word for "small compartment", in reference to Parquet's ability to efficiently store columns next to each other.

Column-based storage

Parquet is defined as column-based since its main selling point is being able to access a single column very efficiently, without having to touch the rest of the file. This results in efficient reading speed when compared to other formats that are row-based. In row-based storage systems, a full row needs to be read whenever any value from the row needs to be accessed. This approach proves to be very wasteful if the user only needs to read a single column from a table. In this case all the rows are first read, and then all values of the row, except the one needed, are discarded. Let's say you want to compute summary statistics like mean, standard deviation, and some percentiles of the prices of a product database. To do that, all you need is the price column of the product table. In a column-based storage system you have direct access to this column, can read it quickly, and do advanced analytics on it.

One of the trends in big data is the increasing number of attributes that are stored, which usually translates to an increasing number of columns in tables. The more columns, the longer the rows of the table will be, and the longer it will take to read a single row. In this context, designing a format such as Parquet solves the issue. Using data compression also reduces the bandwidth required to transfer the data. These features make remote storage a good use case for the Parquet format. One of the strong features of Parquet is that **compression** is implemented on a per-column basis. If all data within one column have the same data type, we say the data are homogenous, which makes it possible to efficiently compress it. Think of a product database again and assume we have ten categories of products in our database which are stored as strings (e.g., "software" or "livestock"). Even if you have several million products, storing category data can be done very efficiently. All you need to do is keep a list of length ten, one item for each category, and store the row indices of the products with that category. In a relational database model, unless you redesign and normalize your data models, you will have to store several million strings for each product alongside other attributes in the same table.

Compression
Data compression algorithms can encode and rewrite binary data in a more compact way while keeping the amount of information intact.

Column-based compression means that different compression techniques can be used for different data types, which increases overall efficiency. Parquet is agnostic when it comes to compression algorithms, which leaves room for new and better compression algorithms that might become popular in the future.

File format specification

Parquet supports several data types: Boolean, signed integers, floating-point values of various lengths, and byte arrays of arbitrary length. The structure of a Parquet file consists of two main parts: the data block and the metadata block. The data block is written first, and the metadata second. At read time, the metadata is read first, so that only the necessary data is read once the file structure is known. The metadata contains information on the data types, compression, nesting levels, etc. This way the user can access the schema of the data before having read any data, in a similar way to what one would do with a database.

The data block is organized in contiguous chunks called row groups. Each row group contains one or more columns, which in turn are divided into pages, which contain the actual values. All parts of the data blocks contain headers which specify the structure and location of its subcomponents.

This complex structure is very useful at read time, when specific columns can be selected while ignoring the rest of the data. Parquet also supports filtering (depending on data types), so that the user can, for example, decide to read only values that are greater than five. These and other features make for very high reading speeds.

Apache Arrow

Released in 2016, Apache Arrow attempts to simplify system architecture, improve interoperability, and reduce ecosystem fragmentation in order to promote the reusability of libraries and algorithms across systems.

An open standard

Apache Arrow defines a language-agnostic format specification for in-memory storage of flat and hierarchical data. The focus is on efficient operation on modern CPU and GPU architectures. In practice, there are Arrow libraries that implement this format for several programming languages. Arrow is not strictly speaking a file format, as it's meant for storing data onto memory, or RAM, which separates it from the formats described so far in this unit. However, its specification is not restricted to memory and can be used for data streams and fast on-disk storage. It was designed to overcome the limitations of in-memory storage that were found in several analytics software and take advantage of most recent hardware advancements such as **GPUs** (Dinsmore, 2016).

GPUs
A graphical processing unit is the main processing unit found in graphics and video cards. They primarily used for computational applications, most notably deep learning.

Interoperability

Normally an application has its own in-memory format for table-like structures such as data frames. If the user needs to pass data from one application to another, the data usually needs to be converted to the new format. This means that the data needs to be copied at least once, possibly needing more memory space, and probably it will have to be copied to disk using an intermediate format that is compatible with both applications. This will require more time and more disk space. Arrow aims to solve all this. Using Arrow, a table created by an application could be passed to another application without needing extra work or copies, since both applications understand the format.

The main use case for Arrow is fast and efficient data analytics. Most notably, Arrow is compatible with data analytics frameworks such as Apache Spark and Pandas. It is also the basis for other file and data exchange formats for data analytics.

Columnar format

Apache Arrow is a columnar format that takes the philosophy of Apache Parquet and extends it. It is designed to represent table-like datasets in memory. It is especially efficient for large quantities of data and takes advantage of modern processor **computing instructions**.

Computing instructions
Central processing units (CPUs) ship with a standard set of instructions, of which the most basics are add, subtract, multiply, and divide. More advanced instructions allow vectorized operations, i.e., on multiple values at the same time.

Arrow works as a shared standard that different libraries can use to save data. This allows for seamless data exchange between different applications without reverting to different internal formats, which would require processing time to convert between each other. Libraries implementing the Arrow format exist in several programming languages and are all free and open source.

Comparing Arrow and Parquet

When we compare Arrow with Parquet, there are a few differences due to the separate use cases they were created for.

While they are both column-based, Arrow has very little support for compression. In general, compression is used to reduce the size of data that have to be transferred from disk to memory, making access faster. Arrow is focused on in-memory data, so this is not an issue.

The internal architecture of the storage format is also different, which is necessary in order to accommodate for the differences between storage disks and RAM.

In a generic workflow, the user would read data from Parquet files. The data would then be converted to Arrow for further analysis and sharing between different analytics applications. It is possible to use the Arrow format for creating files, even though that's not its intended use. It may be convenient when needing to quickly save temporary files on a local disk, but it should not be regarded as a long-term storage solution. Apache Parquet would be the correct choice for this use case.

3.3 Advanced Serialization Frameworks

So far, we've seen many examples of what we coined "traditional" file formats. Some of them, like HDF5, were self-describing, meaning that their description or schema as a data format is built into the format. Now we will look at frameworks that take this idea one step further. These formats use language-agnostic protocols to define what your data will have to look like and then provide you with code, often auto-generated, to send and receive such data. This way of transmitting data is not only very safe, since the generated code gives you schema validation, i.e., it tells you exactly how data has to look, but it can also be extremely fast, since data can be compressed very efficiently. Moreover, these formats are ideal candidates for cross-platform and cross-language communication because the schema specification is independently defined. These serialization formats share some common traits:

- They all use an Interface Description Language (IDL) to specify how to transmit and receive data defined by them.
- They are all binary file formats.
- They are very fast for data transmission when compared to other binary formats, and easily beat text-based formats.
- They can be versioned, i.e., they allow for schema evolution, which means that when your requirements change over time, your data can evolve with them.

Apache Thrift

As a first example, let's have a look at Apache Thrift, which was originally developed at Facebook and has since been publicly released under the Apache 2.0 license. We'll look at Thrift in a bit more detail than the other two formats that follow, as they conceptually share so many traits.

Each Thrift project starts with defining a `.thrift` specification file which you can use not only to define data, but also complex services that interact with that data. We will not discuss Thrift's extensive Interface Description Language (IDL) in detail here and will only focus on giving you a first example that can be used in a service for data exchange. The basic type you have to know to define data structures in Thrift is the `struct`. Defining a struct, and writing a Thrift interface in general, looks a bit like writing C code, but as we mentioned before, Thrift is completely language agnostic. Imagine you're running a company that needs to handle important personal information for your clients. We're going to build a simple interface for defining "person" data and validating it with Thrift. Here's the content of a file we'll call `person.thrift`, which uses an enum type for enumerations (listing properties), structs for data, and a service type to define the interaction with the data:

Code

```
enum PhoneType {
   HOME,
   FAX
}
enum Sex {
   MALE,
   FEMALE
}
struct Person {
   1: string name
   2: i32 age
   3: Sex sex
   4: PhoneType phoneType
   5: i32 phoneNumber
   6: bool validated
}
service PersonService {
```

```
   void ping()
   Person validatePerson(1: Person toValidate)
}
```

We modeled `PhoneType` and Sex after the properties for the "person" objects modeled in several formats earlier, and `Person` is a subset of what we've defined there. You'll notice that to define a `struct` you not only need to provide types like `string`, `bool` or the numerical type `i32`, but you also have to prefix each entry with a number, which Thrift uses internally for its data schemas. Additionally, we defined a service interface called `PersonService`, which you can ping to see if the service is up. You can also use it to send a `Person` data object from a client to a server instance for validation, which we'll get to in just a moment.

With this `person.thrift` interface description file, you can now auto-generate code that allows you to build proper applications around the Person data type and its associated service. Thrift supports many different languages (C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml and Delphi, among others). For now, we're going to focus on Python only. To generate Python code for the above interface file, you need to install Thrift and then run the following command:

Code

```
thrift --gen py person.thrift
```

This will generate a folder called `gen-py`, which contains a Python module called `person` with several files for you. You can test these by creating a Person class in Python, the implementation of which Thrift has done for you:

Code

```
from person.ttypes import Person
person = Person(name="Jane Doe", age=20)
```

Further, you also have a full client for the `ping` and `validatePerson` functions of service already. The only thing you need to do to use this functionality (thereby using Thrift for communication of data) is to provide implementations for these functions. To do so, we create a file called `server.py` with the following contents:

Code

```
import sys
sys.path.append('gen-py') # load the person module
from person import PersonService
from person.ttypes import Person, Sex, PhoneType
from thrift.transport import TSocket, TTransport
from thrift.protocol import TBinaryProtocol
from thrift.server import TServer

class PersonHandler:
```

```
    def ping(self):
        print('ping()')
    def validatePerson(self, person):
        person.validated = True
        return person

if __name__ == '__main__':
    handler = PersonHandler()
    processor = PersonService.Processor(handler)
    transport = TSocket.TServerSocket(host='127.0.0.1', port=9090)
    tfactory = TTransport.TBufferedTransportFactory()
    pfactory = TBinaryProtocol.TBinaryProtocolFactory()

    server = TServer.TSimpleServer(processor, transport, tfactory, pfactory)
    print('Starting the server...')
    server.serve()
    print('done.')
```

While this might look like a lot at first glance, most of it is boilerplate code necessary to start the service. The `validatePerson` method we define in the dedicated `PersonHandler` is very straightforward: we simply accept a `Person` object, set its validated property to `True` and return it again. In a more realistic use case, much more would happen in a data validation step, but this is just an introductory example. You can start this service with

```
python server.py
```

which is enough to try out this service. We do so by creating a file called `client.py` as follows:

Code

```
import sys
sys.path.append('gen-py')

from person import PersonService
from person.ttypes import Person, Sex, PhoneType
from thrift import Thrift
from thrift.transport import TSocket
from thrift.transport import TTransport
from thrift.protocol import TBinaryProtocol

def main():
    # Create socket and use buffers
    transport = TSocket.TSocket('localhost', 9090)
    transport = TTransport.TBufferedTransport(transport)

    # Wrap in a binary protocol
    protocol =TBinaryProtocol.TBinaryProtocol(transport)
```

```
    client = PersonService.Client(protocol)
    transport.open() # Connect to the server
    client.ping()

    person = Person(name="Jane Doe", age=20, sex=Sex.FEMALE,
phoneType=PhoneType.HOME, phoneNumber=2135554321)
    response = client.validatePerson(person)
    print(response.validated)
    # True!
```

In this example we first create a client using Thrift's TBinaryProtocol, connect to the server that is already running, ping the server, and then validate the "Jane Doe.". You might argue that to quickly define data validation functionality, what we just did is overkill. But the point is that systems like Thrift shine when applications scale massively. For one, we defined both our server and our client in Python, but we could have used Java for the client and Go for the server, or any other combination of supported languages, and the above example would still work. This is immensely useful for companies working with heterogenous services written in various languages. Thrift allows you to define Remote Procedure Calls (RPCs), i.e., communication between clients and servers on different hosts in the same network, and does so seamlessly. Given that Thrift's communication is very fast, it becomes an attractive choice for service communication. Note that you can also easily swap the roles between client and server to send data bidirectionally, too.

As a last point worth mentioning when discussing the benefits of Thrift, you can't send or receive the wrong data by accident. Only data that pass the internal schema validation are allowed. To give an example, let's say that in the above example, instead of "Jane Doe", we'd tried to send data for "John Doe" whose age we wrongfully format as string "30", not as an integer:

Code

```
person = Person(name="John Doe", age="30")
validated = client.validatePerson(person)
```

This is immediately caught as an argument error with proper type hint by Thrift.

Code

```
error: required argument is not an integer
```

To summarize the process of the example we've just investigated, the following steps are involved when working with Thrift as protocol for communication:

- Define a `.thrift` file specifying data and functionality of a service you want to use.
- After downloading and installing Thrift, generate code in the languages you want.
- Use the languages you chose in the last step to implement one or several servers to be provided in services.

- Use the generated clients to send data and utilize the functionality implemented in the server. Thrift has different binary protocols to choose from for data serialization.

Following this process, you see that Thrift is more than just a data format. It is also an extensive IDL that generates code for complex client-server interaction and a system for RPCs.

Apache Avro

Avro is a serialization framework developed within the Apache Hadoop project and can be used for RPC, just like Thrift. It also uses protocols defined in a configuration file and encodes its data in a binary format. You can (but do not have to) run code-generation similar to what we've seen for Thrift. Avro has a description language called Avro IDL, but you can also describe schemas in a JSON format, which we will do in an example below. There are many languages that have official support for Avro, including C, C++, C#, Go, Haskell, Java, Javascript, Perl, PHP, Python, Ruby, Rust, and Scala.

To define a schema of a person like that which we defined for Thrift, you create a file called `person.avsc` which has the following JSON data:

Code

```
{
  "namespace": "person.avro",
  "type": "record",
  "name": "Person",
  "fields": [
     {"name": "name", "type": "string"},
     {"name": "age", "type": ["null", "int"]},
  ]
}
```

As you can see, Avro allows you to define records, the equivalent of structs in Thrift, which has fields with names and types. Of note is that the type can be a list and must include a null value to make the respective field optional. To use this schema to write example data in Avro's binary format, we can do the following in Python:

Code

```
import avro.schema
from avro.datafile import DataFileReader, DataFileWriter
from avro.io import DatumReader, DatumWriter

schema = avro.schema.Parse(open("person.avsc", "rb").read())
writer = DataFileWriter(open("users.avro", "wb"), DatumWriter(), schema)
writer.append({"name": "Jane Doe", "age": 20})
writer.append({"name": "John Doe"})
writer.close()
```

Essentially, we parse the schema and open a writer for the file `users.avro`, then append sample data and close the writer when done. To read this binary data format back into memory and print it in human-readable form, you use the following code:

Code

```
reader = DataFileReader(open("users.avro", "rb"), DatumReader())
for user in reader:
   print(user)
reader.close()
```

Doing so in fact gives us back the data we entered initially, in JSON format (note the None value for the example with missing age entry):

Code

```
{u'age': 20, u'name': u'Jane Doe'}
{u'age': None, u'name': u'John Doe'}
```

In comparison to this JSON format, Avro IDL is not very aesthetically appealing. As you can see from this example, in Avro you don't need the more complex "service" setup to use it as data format. In terms of features, Avro is comparable to Thrift, but when it comes to adoption, Avro is mainly used within Hadoop, while Thrift enjoys wider adoption.

Protocol Buffers

The last serialization framework we're considering is probably the most popular today. Protocol buffers, or protobuf for short, were built by Google almost 20 years ago and are still being used, not only within Google itself, but also in open-source tools like Google's TensorFlow framework. Protobuf is similar to Thrift in many aspects, in particular its ability to be integrated within an RPC application using Google's gRPC protocol. As a language specification, protobuf is more lightweight than XML, but faster. Protobuf natively has support for code generation in C++, Java, C#, Python, Go, Ruby, Objective-C, JavaScript, and PHP, but there are third-party implementations available for many other languages.

The basic data structures you define in protobuf are called messages, which unlike other structures we've seen, are not self-describing. You define your messages in .proto files, which for our running example of a person could look as follows:

Code

```
syntax = "proto2";
message Person {
 required string name = 1;
 optional int32 age = 2;
}
```

Protobuf allows you to explicitly mark fields in a message as `required` or `optional` and has many basic and advanced data types that we've seen for the other protocols before. Note that you need to specify which major version of protobuf's syntax you want to use, and we're using "2" for this example. Instead of a prefix to fields, in protocol buffers you assign numbers to fields after their name. After installing the protobuf compiler `protoc` on your system and saving the above content into a file called `person.proto` you can run code generation for Python like so:

Code

```
protoc --python_out=. person.proto
```

This will generate a file called `person_pb2.py` which you can use to conveniently define a person in object-oriented style as specified in `person.proto`, including checks for data types and fields, for instance:

Code

```
from person_pb2 import Person
person = Person()
person.name = "Jane Doe"
person.age = 20

person.age = "20"
# ----> 1 person.age = "20"
# TypeError: '20' has type str, but expected one of: int, long

person.fail = 42
# ----> 1 person.fail = 42
# AttributeError: 'Person' object has no attribute 'fail'
```

We're not going into specifics of using protobuf for RPC, but note that the gRPC protocol looks and feels very similar to the example we discussed for Thrift earlier in this section, including a `service` definition within the `.proto` file.

SUMMARY

The evolution of data formats has followed and adapted to the evolution of data. Text files have become largely obsolete for big data application, and binary file formats have become the de-facto standard. Using compression techniques and a complex internal structure, binary files have overcome the limits of text files concerning storage size, read, and write speed.

We've discussed traditional file formats that are still relevant today, such as CSV, XML, JSON, YAML, BSON, and HDF5.

Apache Parquet is a format optimized for on-disk and remote storage. It is a column-based format and supports compression. Its columnar format is designed to optimize read speed. The use of compression minimizes storage size and transfer bandwidth.

Apache Arrow is a columnar format designed for in-memory usage. It draws from the same philosophy of Apache Parquet. Its focus is interoperability between big data analytics ecosystems, such as pandas and PySpark. Arrow is designed to store table-like objects found in big data software, so that the same object stored in memory can be used by multiple applications. This way, when switching between them, there is no need to copy or convert the data, which optimizes memory size and speed.

Finally, we looked at advanced serialization frameworks like Apache Thrift, Apache Avro, and Google's Protocol Buffers. These frameworks can be used to specify data and also include sophisticated message passing systems used in RPC applications.

UNIT 4

MODERN BIG DATA PROCESSING FRAMEWORKS

STUDY GOALS

On completion of this unit, you will have learned …

- the differences between batch processing and stream processing frameworks.
- the importance of the MapReduce framework for processing large-scale datasets.
- why Hadoop has been superseded by Spark in terms of processing speed.
- how distributed query engines can be used to make a data infrastructure more homogeneous to work with.

4. MODERN BIG DATA PROCESSING FRAMEWORKS

Introduction

The rapid expansion of the internet has led to an enormous increase in publicly-available data. The amount of data being produced daily has long outgrown the processing power of any single computer. It therefore has become necessary to find ways to break big data analytics tasks into smaller subtasks that can be solved by single machines working in orchestration. Working with large amounts of data involves considering two components: a way to store the data and a way to process them, i.e., a way to create new insights from analysis. What follows is an overview of some technologies that aim to solve big data problems with respect to the second part: processing.

We'll look at two fundamental ways of processing data, starting with batch processing with the MapReduce compute paradigm (and its most popular implementation, Hadoop). Then we'll introduce the Spark framework, which in many ways is a successor to Hadoop. After that, we'll look into streaming data (another way of processing data) by investigating systems like Flink and Spark streaming. Then we will get to know the idea of distributed query engines and how they can help us consolidate a heterogeneous data infrastructure. In the last section we will look into other big data processing tools, specifically for the Python ecosystem.

4.1 Batch Processing Frameworks

Batch processing refers to running a computer program, without user interaction, for which all necessary data are available when launching the program. This type of processing gets its name from the fact that the dataset is processed in chunks, or batches, during execution. Batch processing is usually used to gain insights from massive datasets that have been created or collected over extended periods of time. A common theme in batch processing is that the large amounts of data have to be spread and stored across several machines. In classical programming paradigms, you would write a program that loads data and computes something using that data ("bringing data to compute"), whereas in big data systems you usually have to bring the compute instructions to the data somehow. Doing so in a distributed data setup is not easy, and you need dedicated algorithms. Generally speaking, designing and implementing complex computation frameworks (and performant storage solutions) is one of the core challenges of the big data age.

For instance, a company might be interested in performing sentiment analysis on millions of product reviews publicly available on the web to gain insight into how their products are perceived by their customers. Because new reviews are written all the time, such a company might take a snapshot of its product review data first, then run a suitable sentiment analysis algorithm on this fixed dataset (i.e., a batch process), and then store the

resulting sentiments back into a database of their choice. Note that the original data source from which this company takes snapshots and the database to which they store sentiment analysis results may have nothing in common with one another. When this company reaches a certain size and their product catalog becomes large enough, storing the respective product and sentiment data becomes a challenging problem, as does batch processing it.

Hadoop and MapReduce

Hadoop is a framework for implementing software systems for processing large amounts of data in a parallel manner on many nodes of commodity hardware. It is an implementation of MapReduce, a programming model for processing datasets with a distributed algorithm on a cluster of computers. Hadoop features mechanisms to ensure a reliable and fault-tolerant mode of operation.

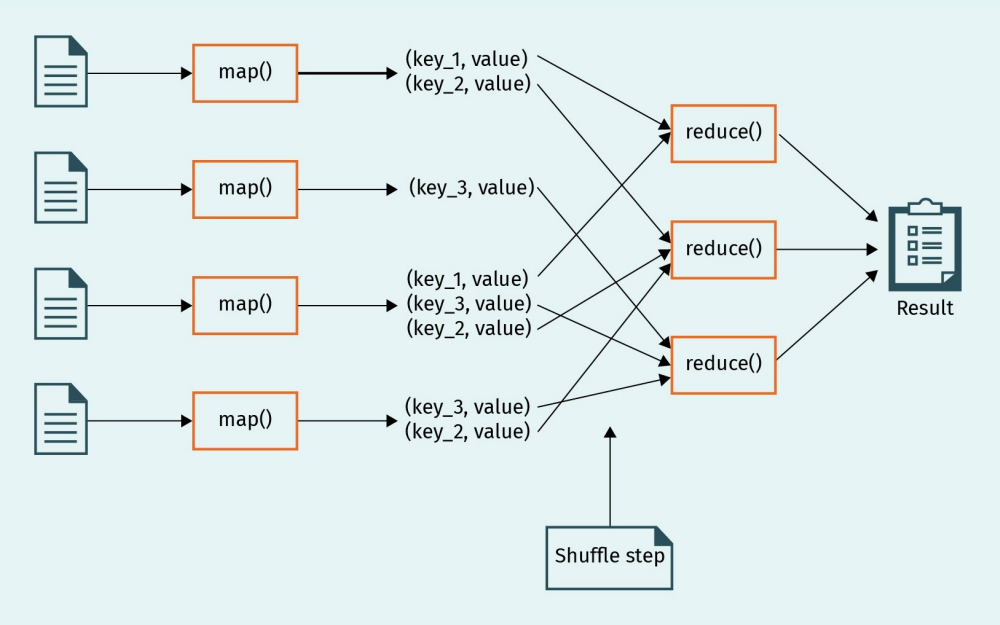
Suppose that we have one million books and would like to compute the distribution of words found in all of them. In other words, we would like to know how many times each word was used throughout those books. In the past, when most computing was done on a single processor, this task would take a very long time, even on the fastest of processors. One way to deal with this problem is to divide a large task into several subtasks. In this example, a key insight is that you can count all words and their occurrences for each individual book first, and then sum up your counts for all the books you are considering. The first step, counting the words in a single book, is independent of how many books you have and what their word distribution looks like. This means we can parallelize this task. Once you have counted all words for each book, what you end up with is essentially a table with word occurrences for each book. To further proceed with your task, you can now forget about the original books and just work with your occurrences tables. In other words, you've transformed, or "mapped", a large dataset (namely, books) into a much smaller data set by processing it in parallel. Now all that's left is to combine the tables in a straightforward way: When two tables have the same word, sum up their occurrences, and if only one has the word, just take that occurrence. This way you can slowly aggregate the one million tables into a single table, i.e., by "reducing" all tables into one. Before carrying out this reduction step, it might make sense to sort your table items lexicographically, so that it's easier to look up individual items. While this set of tasks is tedious for humans, it can be efficiently done by a computer. The schematic procedure of transforming a million books into a word distribution, by first mapping the data in parallel, sorting it, and then reducing the dataset to the result you want, is appropriately called "MapReduce".

To be more precise, a MapReduce program consists of two fundamental steps: a map function and a reduce function. But, concrete implementations of this paradigm usually come with auxiliary steps in the process. The terms "map"and "reduce" are derived from the area of **functional programming**. In the context of MapReduce, `map` implements a simple operation representing a subtask of a problem. Simply put, this operation is applied to all elements of a dataset in parallel by the framework (e.g., Hadoop), which transforms ("maps") each data point into a desired output. This output has a special structure, namely, each element of it consists of a key and a value. In the word-count example above, the keys are the words, and the values are the counts. Afterwards, the MapReduce framework groups all individual map outputs by their key in what's called a shuffle-and-

Functional programming and higher-order functions
In functional programming, map and reduce refer to two instances of higher-order functions. Higher-order functions can either take other functions as arguments or return them as results.

sort step. Finally, the results of this step are handed over to the `reduce` function. This function performs a summary operation over all the individual key-value sets produced by the shuffle-and-sort step. The following diagram illustrates how this algorithm works schematically.

Figure 9: Graphical Representation of a MapReduce Program



Source: Pumperla (2020).

The following code sample implements a non-parallel and very simplified variant of a MapReduce framework in Python:

Code

```
# Define a map function that returns (word, 1) as key-value pair for
each word in the text.
def map_func(text):
    for word in text.lower().split():
        yield word, 1

# Define a reduce function that sums up the counts of words.
def reduce_func(key, values):
    total = 0
    for count in values:
        total += count
    return key, total

# Create a toy data set to run the program on.
text_1 = "Lorem ipsum dolor sit amet, consetetur et sadipscing elitr."
text_2 = "At vero lorem et accusam et justo duo ipsum et ea rebum."
```

```
dataset = [text_1, text_2]

# Apply map function all elements in the dataset
map_results = []
for element in dataset:
   for map_result in map_func(element):
       map_results.append(map_result)
print(map_results)
# Output: [('lorem', 1), ('ipsum', 1), ('dolor', 1), ('sit', 1), …]

# Group the key-value pairs produced by the map step by their key.
This is the sort-and-shuffle step.
groups = dict()
for key, value in map_results:
   if key not in groups:
       groups[key] = []
   groups[key].append(value)
print(groups)
# Output: {'lorem': [1, 1], 'ipsum': [1, 1], 'et': [1, 1, 1, 1],
'dolor': [1], …}

# Apply the reduce function to each key-value pair
reduce_results = dict()
for key, values in groups.items():
   reduce_results[key] = reduce_func(key, values)
print(reduce_results)
# Output: {'lorem': ('lorem', 2), 'ipsum': ('ipsum', 2), 'et':
('et', 4),'dolor': ('dolor', 1), …}
```

From a developer's perspective, Hadoop and other MapReduce implementations manage the entire infrastructure necessary to reliably and efficiently work with large amounts of data. All that is left to do for the developer is implement both the `map` and `reduce` functions (called `map_func` and `reduce_func` in the above code example to avoid confusion, because `map` and `reduce` have been built-in functions of the Python programming language for a while). Managing communications and data transfers between the different parts of the system and providing fault tolerance are taken care of by the underlying framework. This makes the developer's job much easier, as complex functionality such as re-scheduling tasks or jobs in case of hardware failure is provided out-of-the-box by the framework.

Hadoop Distributed File System

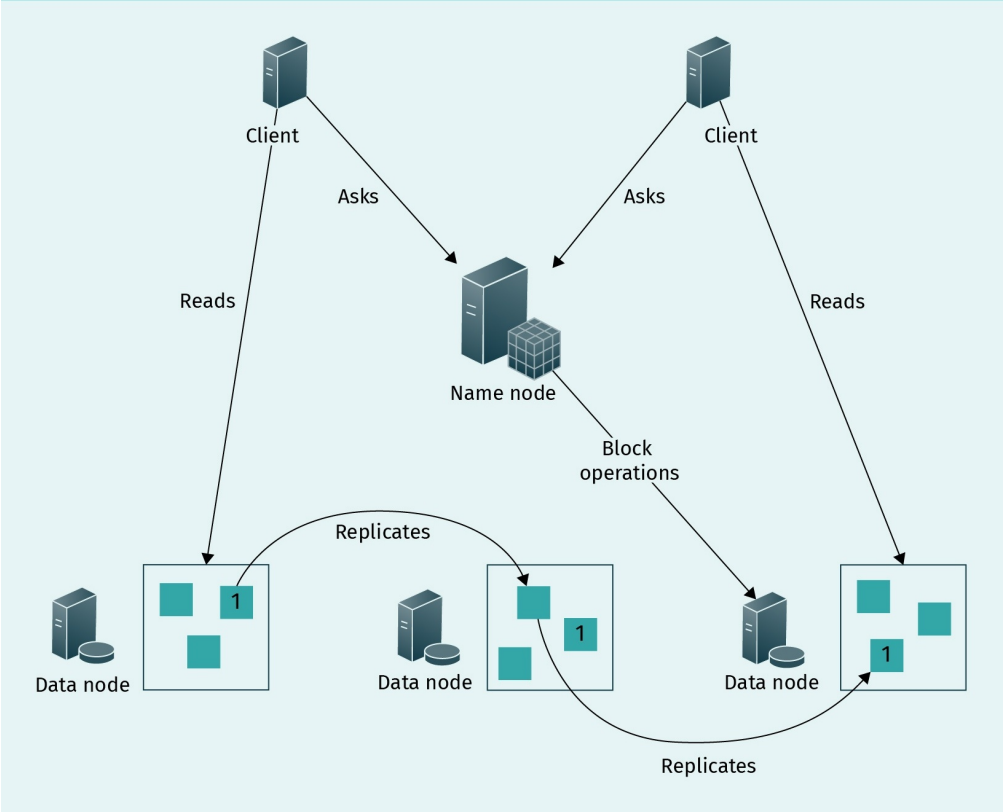
Hadoop reads data, writes intermediate results between steps, and stores final results to the Hadoop Distributed File System (HDFS). HDFS is a scalable file system distributed over multiple nodes in a Hadoop cluster. The motivation for developing the HDFS was that most large-scale data analysis was run on oftentimes very large files. Moreover, machine

learning tasks, computing statistics, and other data mining methods usually access and process all available data in their entirety, i.e., the focus is on batch processing, and querying for information is secondary.

HDFS divides files into chunks or blocks, usually between 16 and 64 megabytes in size. Each of the chunks is replicated a certain number of times on different machines. The default number of replications in HDFS is three. The chunking and discovery of files is orchestrated by a name node (historically also referred to as a master node, following a culturally-insensitive master-slave naming convention that has been prevalent in distributed computing, but has largely vanished in the last years) that manages a directory with the location of all copies of a file. By contacting the name node, all participants using the distributed file system know where the files and replicas are located. The name node is more aptly called the primary name node, as there is a secondary name node in Hadoop as well. This secondary name node is not a fallback or drop-in replacement for the primary name node, but rather a helper node that stores metadata, etc.

Besides the name node, HDFS also consists of data nodes whose purpose is to store the data blocks. This kind of node is also responsible for serving read and write requests from the clients. Name nodes perform block creation, deletion, and replication upon instruction from the name node. The following diagram shows how Hadoop is schematically set up for storage and how client applications interact with it.

Figure 10: Structure of a Hadoop Distributed File System



Source: Pumperla (2020).

Apache Spark

Since its inception in 2014, Spark has become the new standard for scaling out batch processing tasks. The main reason for Spark's success is speed. Its high performance results mainly from the fact that it keeps the data in main memory in-between processing steps. By doing so, Hadoop's costly writes to disk and the resulting reads between the steps can be avoided (Samadi et al., 2016). Spark can also be deployed on various modern cluster computing infrastructures.

In addition to better performance, Spark offers other features that set it apart from Hadoop. Most prominently, Spark doesn't impose restrictions on the sequence of computational steps. That is, with Spark it is possible to define flows other than "first map, then reduce". From that perspective, a Spark program is simply a pipeline of operations on Resilient Distributed Datasets (RDD). Spark calls these operations "transformations". Spark works with Java, Scala, Python, and R, and also has a special query language called Spark SQL that closely resembles SQL. On top of the core Spark library, there are currently four additional components available, namely:

- Spark SQL, the already mentioned query language for Spark

- Spark GraphX, a set of tools specifically designed to work with graphs, i.e., nodes and edges, at scale
- Spark MLlib, a machine learning library on top of Spark
- Spark Streaming, for analyzing continuously incoming flows of data

An RDD is a read-only collection of objects partitioned across the machines in the cluster in a fault-tolerant manner. RDDs can, for instance, be created by reading data from a file system such as HDFS, and you can apply arbitrary transformations to an RDD to process your data.

To give you a concrete example of the Spark RDD API, let's revisit our word count example we created in our MapReduce implementation. With Spark in Python this boils down to just a few lines of code. Each Spark program starts off by creating a `SparkSession`, which you can think of as the coordinator of the computation that needs to be carried out on a compute cluster. The great thing about Spark is that you don't need a cluster of several machines to test it; you can create a `SparkSession` on your laptop or desktop machine, which emulates the full cluster setup:

Code

```
from pyspark.sql import SparkSession

spark = SparkSession.builder\
   .master(“local”)\
   .appName("PythonWordCount")\
   .getOrCreate()
```

Now, assuming you have a plain text file called `words.txt` on your machine next to where you execute this Python code, you can now compute the word count of this document with Spark:

Code

```
# Read the text as lines into an RDD.
lines = spark.sparkContext.textFile("words.txt")

# Split lines into words individually.
split_lines = lines.flatMap(lambda x: x.split(' '))

# Map each word to (word, 1)
words = split_lines.map(lambda x: (x, 1))

# Sum up all word occurrences.
counts = words.reduceByKey(lambda a, b: a + b)

# Collect returns the result on "master"
output = counts.collect()

# Print the resulting counts
```

```
for (word, count) in output:
print(f"{word}: {count}")

# Stop the Spark session
spark.stop()
```

Note how similar this program looks to our naïve MapReduce implementation on a high level. On the other hand, you can also see how Spark is not confined to the rather strict paradigm of MapReduce. In fact, you could have applied any other set of transformations to the initially created RDD called `lines`.

4.2 Streaming Frameworks

Data streams can be characterized in multiple ways. By common definition, a data stream is a continuous flow of data items following a particular temporal order. From a more formal perspective, a data stream can be seen as a continuous and theoretically infinite process in which events occur independently from one another.

Especially in times of big data and the Internet of Things, many data streams produce huge amounts of data. For example, the Large Hadron Collider (LHC) built by CERN produces up to one billion particle collisions each second while running experiments. As it is not possible to record all these events, a filtering system is in place to reduce the amount of data and select only those events deemed interesting enough for further analysis. Despite the data reduction efforts, CERN still needs to store about 90 petabytes (90 million gigabytes) each year.

The LHC is a great example of an application best modeled in terms of data streams. A common characteristic of stream-processing software systems is that, in most circumstances, they only have one opportunity to process each data item. This characteristic is also referred to as single scan. Once an element has been processed, it is discarded or archived. Although in the latter case it is in theory possible to process the archived data again, this is usually an expensive operation that should be avoided as much as possible. Another characteristic of stream-processing systems is that they have no influence on the order in which the data arrive, or when they arrive.

There are multiple ways to deal with the issue of finding a trade-off between storage space requirements and being able to provide precise answers to the questions the system is designed to answer. In cases such as the LHC, it is impractical or even impossible to store all generated data. Solutions to this problem include storing a kind of summary of previously-seen data by intelligently reducing the data or simply forgetting old or out-of-date data.

In the context of stream-processing systems, sampling refers to selecting a subset of the data elements as they arrive in the system. What might sound quite straightforward at first glance becomes more of a challenge when obtaining a representative sample is a require-

ment. Besides data sampling, other methods for reducing the amount of data include data summarization (e.g., by means of data element clusters grouping similar objects seen in the past) and **histograms** to approximate the distribution of values over time.

Histogram
A histogram is a tool to approximate continuous distributions by "binning" the space of possible values into non-overlapping intervals and then counting the values or data elements per interval.

Many stream-processing frameworks have been developed to tackle real-world big data problems. The next section will introduce the basic concepts of Apache Flink. Besides being used by many companies dealing with large amounts of data, Flink has been shown to outperform other stream-processing frameworks in academic benchmarks (van Dongen & van den Poel, 2020).

Flink

Flink is built around the concept of dataflow graphs. A dataflow graph is a way to describe the flow of data between different operations. Since there are usually sequential dependencies between multiple operations (i.e., one operation has to be performed before another), Flink models dataflows as directed graphs. In these graphs, nodes are called operators and represent computations, while edges represent data dependencies. Special cases include operators without inputs (also called data sources) and operators without outputs (called data sinks).

As an example, consider a system that analyzes new tweets as they are published by counting how many hashtags they contain (Hueske & Kalavri, 2019). This system could logically be broken down into several sequentially-executed operators: A `TweetSource` operator would serve as the data source and would send the incoming tweets to an `ExtractHashtags` operator. Then, a `Count` operator could proceed to count the hashtags extracted in the previous step. Finally, a sink operator could exploit the processed data, e.g., by saving them to a database.

It is important to note that this logical view of the entire streaming pipeline is an idealized version of what really happens when Flink processes events. During the startup process, Flink converts this logical graph into a physical dataflow graph. In such a graph, each operator can potentially be executed within several parallel tasks being processed on different machines. This enables the framework to parallelize operations over multiple machines, making the processing of large-scale streaming data possible.

All tasks are run by a `TaskManager` instance, i.e., a representation of a compute node within a Flink cluster. A `TaskManager` can run tasks of different operators and can share working on one task together with other `TaskManager` instances. Each `TaskManager` can accept a certain maximum number of tasks which are distributed by the `JobManager`.

The `JobManager` is in charge of the execution of orchestrating an entire Flink application. A Flink application can't continue to work if the `JobManager` fails for some reason. This makes the `JobManager` a **single point of failure**. In this sense, a `JobManager` is similar to the master node of the Hadoop Distributed File System, which also represents a single point of failure. For many high-availability applications, a single point of failure is not acceptable. For such cases, Flink features a high-availability runtime mode.

Flink's Python API, called `PyFlink`, has two main facets, namely, the Table API and the DataStreams API. While the first gives you access to a query language similar to SQL (hence, the name "Table", as in database table), the latter provides lower-level access to Flink's Streaming engine.

Single point of failure
In computing, a single point of failure is an obligatory part of a system without which the system as a whole will stop working. As such, single points of failure should be avoided in a system architecture as much as possible.

For an example of the Table API, let's consider yet another word count example. Flink has the concept of data sources, which is precisely what you expect them to be, and data sinks, where data are stored again after processing. To use the Table API, we first need to define one source and one sink, simple files in our example, and equip them with a schema, and then register both as tables in our Flink execution environment:

Code

```
from pyflink.dataset import ExecutionEnvironment
from pyflink.table import TableConfig, DataTypes, BatchTableEnvironment
from pyflink.table.descriptors import Schema, OldCsv, FileSystem
from pyflink.table.expressions import lit

exec_env = ExecutionEnvironment.get_execution_environment()
exec_env.set_parallelism(1)
t_config = TableConfig()
t_env = BatchTableEnvironment.create(exec_env, t_config)

t_env.connect(FileSystem().path('/tmp/input')) \
    .with_format(OldCsv()
        .field('word', DataTypes.STRING())) \
    .with_schema(Schema()
        .field('word', DataTypes.STRING())) \
    .create_temporary_table('mySource')

t_env.connect(FileSystem().path('/tmp/output')) \
    .with_format(OldCsv()
        .field_delimiter('\t')
        .field('word', DataTypes.STRING())
        .field('count', DataTypes.BIGINT())) \
    .with_schema(Schema()
        .field('word', DataTypes.STRING())
        .field('count', DataTypes.BIGINT())) \
    .create_temporary_table('mySink')
```

Note that we use `BatchTableEnvironment` above, as this is a very simple batch processing example, which can also be handled with Flink; however, we could also use `StreamTableEnvironment` instead for a proper streaming application. Now that we've defined sources and sinks, which was the more difficult part of the application, we can run a quick word count with an SQL-like query using `select` and `group-by` statements to aggregate words by counting them from the source `mySource` and storing them in the sink `mySink`:

Code

```
tab = t_env.from_path('mySource')
tab.group_by(tab.word) \
   .select(tab.word, lit(1).count) \
   .execute_insert('mySink') \
   .wait()
```

We put all the code used for this example into a file called `table_api.py`. To test this program, we can put the following content into the source `file /tmp/input`:

Code

```
flink
pyflink
flink
```

Then we simply run `python table_api.py`, which produces the expected word counts

Code

```
flink2
pyflink1
```

in the sink `file /tmp/output`.

A first example of the DataStream API can be computed much quicker, as we can create a streaming source directly from a Python collection to do so, and we don't have to specify an output schema. This time we're using a `StreamExecutionEnvironment` for running the Flink job and all we do is take the data we receive from the source and immediately store them to the sink, which is the same output file as before:

Code

```
from pyflink.common.serialization import SimpleStringEncoder
from pyflink.common.typeinfo import Types
from pyflink.datastream import StreamExecutionEnvironment
from pyflink.datastream.connectors import StreamingFileSink

env = StreamExecutionEnvironment.get_execution_environment()
env.set_parallelism(1)

ds = env.from_collection(
collection=[(1, 'flink'), (2, 'pyflink')],
type_info=Types.ROW([Types.INT(), Types.STRING()])
)
ds.add_sink(StreamingFileSink
.for_row_format('/tmp/output', SimpleStringEncoder())
.build())
env.execute("data_stream_example")
```

If you store this code in a file called `ds_api.py`, you can simply run this example with Python by executing

Code
```
python ds_api.py
```

You can check that this streaming job produces the following output in `/tmp/output`, a comma-separated value format with the entries provided in the code snippet above:

Code
```
1,flink
2,pyflink
```

Spark Streaming

Having discussed Apache Flink and its two main Python APIs, let's now turn back to the Spark ecosystem and see what the Spark Streaming module has to offer. The most notable thing about this streaming engine is that it integrates seamlessly with the rest of Spark's toolbox. For instance, if you want to leverage Spark SQL you can do this for streaming jobs the same way you did for batch jobs in Spark core. Spark Streaming takes in data from various sources and processes the continually incoming data with a set of transformations, before storing the processed data back to a (distributed) database like HDFS, or monitoring and reporting tools.

The basic data structure of a Spark Streaming application is that of a `DStream`. Instead of working with a `SparkSession`, as we did in Spark core, Spark Streaming works with what is called a `StreamingContext`. Structurally, Spark and Spark Streaming are very close to each other by design, but providing a complete example with real-time streaming data is slightly beyond the scope of this course book.

Apache Kafka

The last tool to mention in this section, although there are many other notable streaming data frameworks out there, is Apache Kafka. Kafka is advertised as a distributed streaming platform and was built to deal with real-time data streams. It was originally built by LinkedIn and open-sourced in 2011; it gets its name because it is optimized for writing (and the famous writer Franz Kafka wrote a lot), i.e., high throughput in data transfer. Notably, Kafka is a very popular choice in big data projects, even if these projects don't have a need for streaming analytics. This is because Kafka is often used in its role as a **message broker**, to handle events of any kind.

Message broker
In a message broker, one or several data producing applications, or producers, send messages to an entity called a broker, which can subsequently be retrieved by as many consuming applications, or consumers, as needed.

Kafka makes it easy to consolidate different data sources. An interesting example of this is when you have thousands of devices sending sensor data in an Internet of Things (IoT) application. These devices may not only send heterogenous signals, but also might send them in irregular intervals. Kafka can be used as the central hub and first stop to organize all these sources and collect the data as needed. Kafka stores key-value pairs and groups the data it receives into topics, which users can define to suit their needs. In our IoT exam-

ple, you might send both temperature and memory usage data for each connected device, and Kafka would help you to organize these different types of data by having a topic for each. Once you set up a Kafka cluster and create topics, you can use the Producer API to send messages to Kafka and the Consumer API to subscribe to data from one or more topics.

Interestingly, Kafka not only integrates well with other streaming platforms, like Spark Streaming, it is also often used in conjunction with other such tools. In our IoT example, you might find it useful to use Kafka for data collection as a message broker, and then have a Spark Streaming job consuming Kafka data to do some real-time analytics tasks like checking for overheating devices or memory leaks.

4.3 Distributed Query Engines

A query engine is a software solution that acts as an intermediate layer between a database or server on the one side and users or applications on the other side by interpreting and executing commands. Distributed SQL query engines are a special case of query engines. They allow a developer to query data transparently from multiple data sources such as HDFS, cloud storages, or relational databases with a single query.

While Spark, Flink, and other distributed computing engines are generally flexible enough to retrieve data of any kind, query engines are specifically designed to give you data access fast and in a convenient manner. These query engines are optimized for reading and aggregating data in real-time and generally achieve high performance doing so. There are many popular query engines out there, like Apache Impala, Apache Drill, or Apache Hive. In the subsequent section, the fundamental concepts of query engines will be introduced by focusing on Presto, a popular distributed query engine solution.

Presto

Presto is a distributed query engine designed to support many popular data backends. It was originally developed at Facebook in 2012 and open sourced in 2013. It enables its users and applications to communicate with a variety of different database systems using only standardized SQL statements. Many relational databases feature vendor-specific dialects of SQL, resulting in different feature sets. In order to make the Presto SQL queries compatible with all underlying database backends, it was therefore necessary to rule out non-standard SQL features.

The backend data sources supported by Presto out-of-the-box include widespread systems, such as MySQL, PostgreSQL, and MongoDB, and non-database sources, such as the message broker Apache Kafka. Common tasks that can be achieved with Presto are joins between different data backends, as well as ETL jobs in a single SQL query. Presto also supports transparent data migrations from one backend to another. This support of a variety of backends is accomplished by specialized plug-ins, called connectors. Presto offers a framework for developing custom connectors for data sources not yet officially supported.

There are multiple situations in which the introduction of Presto into a data infrastructure might be beneficial. For one, it provides a single SQL-based access point to many different data sources. Since SQL is a very well-known and widespread query language, making a non-SQL data source available to be queried with SQL will make that data source accessible to more developers because they don't have to learn an additional query language.

Presto can also be very helpful when it comes to moving data between two or more data sources. The combination of support for many different types of data sources and SQL as the primary interface allows developers to query data, transform them, and finally write them back to the original data source or another data source. For example, this means it is possible to transparently copy data from a key-value store into a relational database system without having to manually account for the differences in data representation. Aggregating data from multiple sources can lead to new insights, too (Fuller et al., 2020).

Moreover, it is common to take data from an operational RDBMS, or maybe an event-streaming system like Kafka, and move them into a **data lake** to ease the burden on the RDBMS in terms of querying by many analysts. ETL processes, now often also called data preparation, can be an important part of this process to improve the data and create a data model better suited for querying and analysis.

Data lake
A data lake is a single, central location, that holds all data of a company in an unprocessed format and provides ways to organize, prepare, and query that data.

Presto emphasizes the separation of compute and storage resources. A Presto setup consists of one coordinator node managing multiple worker nodes. The Presto coordinator is mandatory for every Presto installation. Its job is to parse statements, plan queries, and manage Presto worker nodes, while monitoring all the workers' activities to coordinate queries. It gets results from the workers and returns final results to the client.

In the context of Presto, the term "statement" refers to ANSI SQL statements in their textual (string) form. These standardized statements consist of clauses, expressions, and predicates. Presto then parses the incoming statement string into a specialized query object internally. Whenever a statement is submitted by a user, the coordinator parses, plans, and schedules a distributed query plan among all its worker nodes.

When Presto executes a query plan, it is broken up into stages which are again split into a series of tasks. This results in a hierarchy of stages and tasks that resembles a tree data structure. Like every tree, every query has a root stage which is responsible for aggregating the output from other stages. Stages are what the coordinator node uses to model a distributed query plan, but stages themselves don't run on Presto workers, just the tasks.

Presto comes with sophisticated query optimization methods. After first generating an initial, unoptimized query plan, a query optimization step is executed to transform the initial plan into a plan that is logically equivalent but can be executed much faster. One of the most important optimization transformations is referred to as **predicate pushdown** (Fuller et al., 2020).

As a practical example, consider a case where Presto manages two data sources: a MySQL database and a PostgreSQL database. Since we would like to transparently use these two data sources as if they were one, we decide to set up Presto. Luckily, it comes with two

Predicate pushdown
The predicate pushdown step aims to apply filtering conditions as early as possible during the execution of the query. This usually results in data reduction by eliminating result rows that would slow down other operations, such as joins, further down the query execution plan.

connectors for both data sources. In Presto, data sources are configured with a catalog. Catalogs are registered through catalog properties files. In our case, we need to define two catalog properties files as follows:

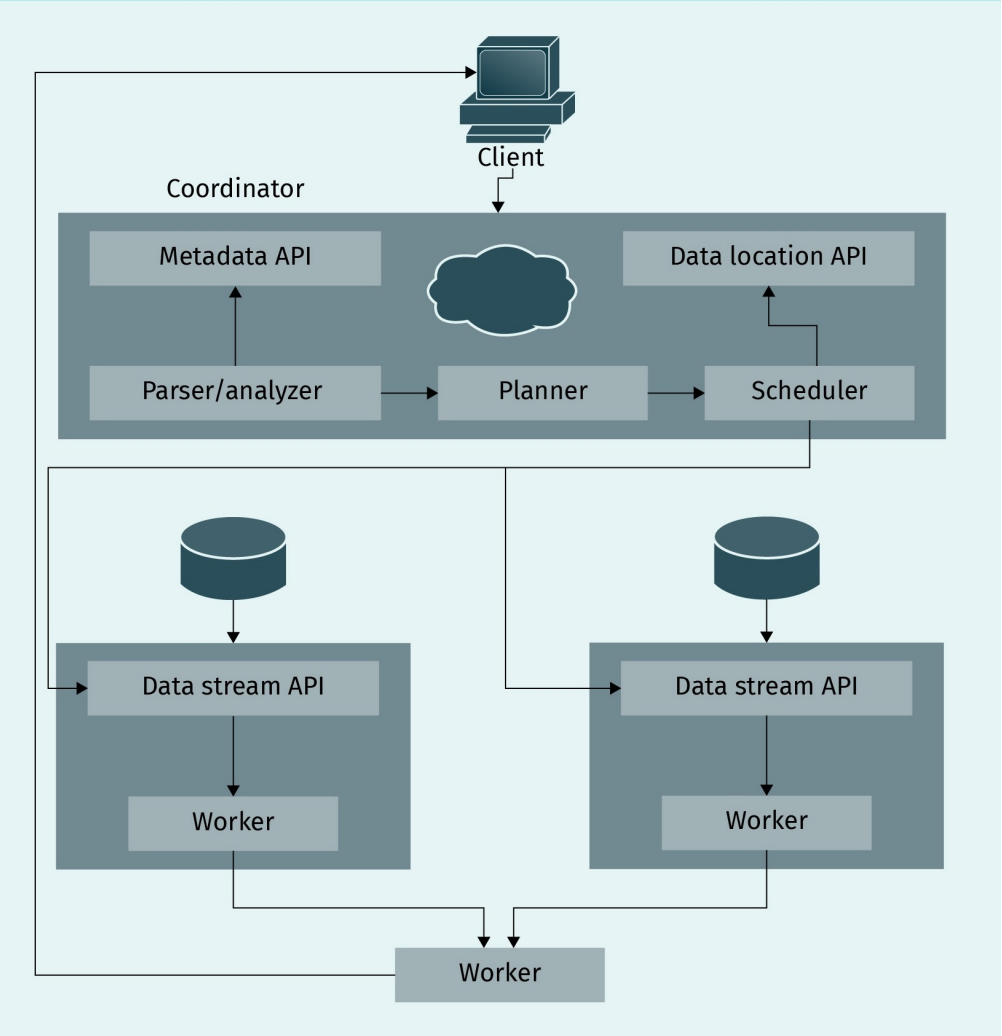
Code

```
connector.name=mysql
connection-url=jdbc:mysql://example.com:3306
connection-user=root
connection-password=secret

connector.name=postgresql
connection-url=jdbc:postgresql://example.com:5432/database
connection-user=root
connection-password=secret
```

In the case of relational databases, these two configuration files are sufficient for Presto to know everything it needs to know. The reason for this is that Presto's own data schema, which is based on tables, maps directly to a relational database schema. For other data sources, more translation work between the different data representations is needed. As already mentioned earlier, from the point of view of the developer, this translation happens transparently. It's the job of the connectors to ensure that the developer can work with all data sources in the same way. Thanks to a public API, it is possible to develop custom connectors if necessary (Fuller et al., 2020).

Figure 11: High-Level Diagram of Presto's Architecture



Source: Pumperla (2020).

4.4 Other Processing Frameworks

Apart from specialized frameworks for batch and stream processing, as well as query processing engines, there are many other tools for data scientists that more broadly fit into the paradigm of distributed computing. Quite a few tools would deserve mentioning here, but we'll focus on three examples from the Python ecosystem: Dask, Ray, and AirFlow.

Dask

Dask is a library for parallel computing written in Python. It is free and open-source, and it was first released in 2015. Overall, Dask is a very flexible platform that makes parallel computing accessible to researchers, data scientists, and programmers alike. Dask focuses on extending interfaces that data scientists are familiar with, hence making distributed computing much easier as compared to the tools discussed previously.

Dask was created to support research and government projects, such as climate simulations, which work with massive multi-dimensional arrays and can require thousands of computing nodes to be executed. However, it is very flexible and can support all sorts of workflows, from big data processing and training of machine-learning models, to scaling of generic Python code for an application. It can also be used interactively. Dask can even run on a laptop and take advantage of multi-core CPU architectures. Interestingly for this case, it can perform out-of-core calculations, i.e., to use the system disk as an extension of the memory.

This allows Dask to treat datasets that are bigger than the system memory, extending the capacity of Python programs. Another main selling point of Dask is that it is entirely written in Python. This is rare, as it's not uncommon for Python libraries to feature a mixture of languages, especially in the scientific or computing domain, and it's meant to facilitate the approach to parallel computing for people who might not be familiar with more complex programming languages, such as C++ or Java.

Directed acyclic graph (DAG)
A directed acyclic graph is a graph where a path starting from one node never returns to the same node. It is a common way to abstract computations in distributed systems.

Dask, like many other distributed computing applications, relies on a **directed acyclic graph** (DAG) to perform a calculation. A DAG is a description of the computation where each node is a task and tasks are connected to their respective dependencies and products, which could link to other tasks. It starts from the inputs and ends with the final outputs of the calculation.

Dask has two main components: a task scheduler and “collections”, which are data objects optimized for parallel computing. The task scheduler is the heart of Dask. It takes the tasks defined in the DAG and decides in which order to run them, optimizing the use of the resources of the system. Dask collections are extension of Python objects such as numpy arrays (Dask Array) or pandas DataFrames (Dask DataFrame), which are very popular in scientific computing and data science, as well as plain Python objects such as dictionaries, lists, tuples, and iterators. They help divide the Python objects into smaller chunks so that they can be separated and moved between different nodes of a computing cluster, or between disk and memory for an out-of-core calculation.

To give you a glimpse at Dask and its capabilities, let’s investigate how it naturally extends numpy arrays and pandas DataFrames in a simple example. After installing Dask in a Python environment, e.g., with `pip installdask`, you can create a random 10,000 by 10,000 Dask Array as follows:

Code

```
import dask.array as da
x = da.random.random((10000, 10000), chunks=(1000, 1000))
```

The “chunks” argument specifies that this large array is stored in a total of one hundred 1,000 by 1,000 numpy arrays under the hood, distributed across your compute cluster, which is just your computer in case you run this locally. The appealing thing about Dask is that it mimics the API of numpy as much as possible. For instance, the above array creation follows the numpy syntax for creating random matrices (apart from the “chunks” argument, which numpy has no need for). Moreover, much of the functionality numpy offers, like arithmetic, basic mathematics, reductions and summary statistics, transposition, slicing, and even some basic linear algebra, carries over to Dask as well. To give a concrete example, we can add the transpose of the above array x to itself and then matrix multiply it with itself to then compute the mean of its 100,000,000 entries as follows:

Code

```
y = x.dot(x + x.T)
y.mean().compute()
```

You need to instruct Dask to return the result as a numpy array by calling `compute()`. Other than that, the exact same syntax works for numpy arrays as well, which is very convenient, as it makes Dask Arrays very easy to learn. Depending on the compute resources of your computer, the above example might take much longer to process using numpy when compared to Dask (if it doesn’t, try to increase the number of rows and columns from 10,000 to 100,000 or more), and might even fail due to a memory exception.

As a second example of Dask, let’s see how closely it mimics pandas and their DataFrames concept. If you have a CSV file called “data.csv” on your machine, which has a categorical “type” column and a numeric “size” column, you can load it into a pandas DataFrame, group the values by “type”, and compute the mean of the “size” column as follows:

Code

```
import pandas as pd
df = pd.read_csv('data.csv')
df.groupby(df.type).size.mean()
```

As in the numpy example before, you can pretty much use Dask as a drop-in replacement for pandas in this example, by modifying the import statement and adding a “.compute()” at the very end.

Code

```
import dask.dataframe as dd
df = dd.read_csv('data.csv')
df.groupby(df.type).size.mean().compute()
```

Ray

Ray was created in 2017 by RISELab, the successor of the UC Berkeley institute that was behind the inception of Apache Spark. While Spark is great for parallelizing workloads for data processing, you can’t really scale out machine learning workloads easily (for instance,

it's not possible to leverage Google's TensorFlow library on Spark). By contrast, Ray sets out to be a general-purpose tool for distributed computing in Python that allows you to run both data processing and machine learning tasks at scale. It does so by providing a fairly simple and lean API for distributed computing on a low level that you can use to build your own applications, and by offering ready-to-use, high-level tools. One key design aspect of Ray is that it's built so that you can parallelize your Python workloads with minimal changes, which boosts productivity.

To get started with Ray, you first install it with `pip install ray`. On your local machine you can then initialize Ray as follows:

Code

```
import ray
ray.init()
```

If Ray was installed on a cluster, you could essentially do the same, but you'd have to pass more arguments to the above "init" call. But even if you start Ray on your local machine, if you have multiple CPUs, as is increasingly standard today, you can easily parallelize Python code with Ray across these CPUs. For example, let's say we want to compute the square of a number, but we want to apply this function to a huge list of numbers. This is an example that is perfectly parallelizable, and you can instruct Ray to define a function that is executed remotely on all the nodes of a cluster simply by using a little decorator:

Code

```
@ray.remote
def square(x):
   return x * x
```

Now you can run remote execution of the square function by calling "remote" on data that you provide locally, e.g., like this:

Code

```
remote_results = [square.remote(i) for i in range(4)]
print(ray.get(remote_results)) # [0, 1, 4, 9]
```

We compute the square for the first four natural numbers above by calling `square.remote`, which we store in a Python list called `remote_results`. As the name of this variable suggests, the result of this function call is still distributed across the cluster. If we want to get back the results, we need to use a `ray.get()` call first.

With the same `ray.remote` decorator above, you can also parallelize whole Python classes, not just functions. To give an example, let's say we want to implement a class that can count objects. This can be done by implementing two methods: one to increment the counter, and one to read the current state of the counter.

Code

```
@ray.remote
class Counter(object):
   def __init__(self):
       self.n = 0
       def increment(self):
           self.n += 1
       def read(self):
           return self.n
```

To initialize five remote counters on your compute cluster, increment each counter twice; once you get back the results, you can do the following, analogous to what we did before with functions:

Code

```
counters = [Counter.remote() for i in range(5)]
[c.increment.remote() for c in counters]
[c.increment.remote() for c in counters]
remote_results = [c.read.remote() for c in counters]
print(ray.get(remote_results)) # [2, 2, 2, 2, 2]
```

Apart from this low-level API called Ray Core, which we just described, Ray comes with additional higher-level libraries for different purposes:

- Ray Tune is a general-purpose tool to automatically tune parameters, instead of manually trying different combinations on your own. This is particularly interesting for machine learning models, which usually come with a set of tunable parameters called hyperparameters. Tune has many algorithms for this use case available, and the distributed nature of Ray helps to compute many parameter choices, instead of computing them sequentially.
- Ray RLlib is a library for a branch of machine learning called reinforcement learning, a paradigm in which a machine learning model learns to make sequential decisions based on receiving a reward or punishment signal.
- RaySGD is specifically designed for machine learning applications, and you can parallelize the training of machine learning models on a Ray cluster.
- Ray Serve makes it easy to reliably deploy trained machine learning models on a Ray cluster.

As you can see from this list of libraries, Ray has strong support for modern machine learning workloads, which is one of the primary reasons it has been gaining traction as a distributed computing tool in the data science community.

Apache Airflow

Apache Airflow is a workflow management application that can be applied in data science projects. It's an open-source project that was first released in 2014. A workflow in Airflow is defined as a sequence of tasks that may or may not be directly connected.

Airflow was initially created by AirBnB to perform ETL jobs and better track and integrate them with all sorts of data sources. This is still the most popular use case for companies using Airflow. It follows the principle of “configuration as code”, where scientists and engineers create programs in order to describe the tasks they need to perform. In practice, Airflow is very useful for automating repeating tasks, e.g., processing raw data to build usable datasets for data analysis or machine-learning models.

Apache Airflow is completely written in Python and uses Python programs to describe the workflows. This means that Airflow is at the same time accessible and very flexible, as the Python language allows you to build dynamic and complex architectures.

To use Airflow the user writes a configuration file that describes a series of tasks and their dependencies. In fact, this file defines a DAG, where tasks are the nodes of the graph, that can be then executed by Airflow. Tasks in the DAG can be very simple (like reading or writing a text file) or very complex (like merging tables from databases). Once the execution is started, Airflow will keep track of all the dependencies and will run tasks again if any dependency changes.

Another strong point of Airflow is scalability. It can run arbitrarily big workflow graphs (because it can run on distributed clusters of arbitrary size), and it's designed to perform even for very large workflows. Its UI makes it very user-friendly and easy to use. It is possible to visualize the DAG and status of tasks in various ways, while numerous metrics and logs are available for past executions. The extensive monitoring possibilities allow for an effective alerting feature: the user can receive notifications by email or via other channels in case some tasks fail. There is also the possibility to detect anomalies, such as when tasks take much longer than usual to complete.

In conclusion, Airflow is an approachable but very powerful workflow management system, which has established itself as one of the most popular solutions for scheduled ETL jobs and similar use cases.

SUMMARY

The advent of the internet and related technologies has made it necessary to develop new technologies that are able to deal with very large amounts of data. This unit aimed to introduce the basic building blocks of any IT infrastructure dealing with said large amounts of data.

First, we introduced one of the classic big data programming paradigms (MapReduce) and a framework that implements it (Hadoop). In the next step, we shed some light on Hadoop's file system and how it was designed from an architectural point of view. As a conclusion to the topic of batch processing, we presented Hadoop's spiritual successor, Spark, and why it is faster when processing large datasets.

We then differentiated between batch processing and stream-processing frameworks and discussed in which situations to choose which paradigm. Finally, we looked at how distributed query engines can help us leverage and manage infrastructures consisting of multiple heterogeneous data sources.

Lastly, we had a look at out-of-core computations of arrays and data frames with Dask, distributed computing with Ray, and complex workflow management with Airflow.

UNIT 5

BUILDING SCALABLE INFRASTRUCTURES

STUDY GOALS

On completion of this unit, you will have learned …

- why orchestration is important when using containers.
- the difference between a monolith and a microservice architecture.
- the advantages of using microservices.
- how technical components are arranged to build up a big data architecture.

5. BUILDING SCALABLE INFRASTRUCTURES

Introduction

Knowing how to store and process large amounts of data is at the core of big data applications and an absolute necessity to build big data systems. An aspect we have not discussed so far is how various databases, big data processing engines, such as streaming or batch processing tools, or data query frameworks are used together to make up a complete system that underpins the big data solution of a company. In other words, we haven't discussed any architectural questions, i.e., how individual components in a big data system need to be designed, built, and deployed in a production environment to suffice the use case a company might be interested in.

Of course, concrete architectures and infrastructures depend on the concrete use case a company faces, and many technology choices can only be decided on with confidence after having carefully considered the practical implications. But there are certain design patterns and ideas that come up in big data systems time and again. In this unit we're considering some of these core ideas.

5.1 Containers and Clusters

Virtualization is a software technique that aims at sharing the resources of a computing system with one or multiple users, effectively creating an emulation of the system or a subset of it. Users of the virtualized system have access to a set of resources defined by the system administrator. For instance, this allows for providing multiple users with independent computing environments all running on a single physical computer. Virtualization technology has advanced enormously in the past couple decades, pushed by the birth and growth of cloud-computing providers, which use the technology to provide on-demand computing resources tailored to user needs.

User space
A user space is the extent of resources (e.g., memory and disk space) that an operating system allows the user to access.

Containers are a form of virtualization at the level of the operating system, in which the container is an isolated instance of a **user space**, where usually a single service is running.

Containers provide a standard platform for running applications across systems. A containerized application will run the same way on a developer's laptop, a desktop computer with a different OS, or a virtual server on the cloud.

In contrast, a virtual machine (VM) is a guest operating system (e.g., Windows) running on the top of a host operating system (e.g., MacOS). VMs virtualize at the hardware level, not on the operating system level, like containers. Just like containers, VMs isolate applications from the host environment, but incur more disk and memory usage than containers, due to the lower level of virtualization (no shared operating system).

Docker

Docker is a set of software products that helps build, test, and deploy an application using a virtualization tool called containers. It's an open-source project and was first released in 2013 (Docker, 2013). Containers are isolated from the system and other containers and can only communicate through HTTP using user-defined ports.

Docker containers are designed to be lightweight, i.e., easy to launch and stop, and are stored and shipped in the form of relatively small "images". A **Docker image** is a way to package an application. The image contains all configuration and code necessary to run, as well as the required parts of the filesystem. Images contain all the necessary information to run a container.

Docker image
A Docker image contains the configuration, filesystem, and code necessary to run an application independently. Docker containers can only run from images.

A Dockerfile is used to build a Docker image. They are plain text configuration files to build a container, and you can think of them like recipes. Dockerfiles contain all the steps required to run an application. They will specify which code to use, which libraries to install, how to set up users and the network, and what command is needed to launch the application. The following Dockerfile shows the domain-specific language (DSL) that Docker uses to build a Docker container that runs a Python script:

Code
```
FROM python:3
ADD main.py
CMD [ "python", "./main.py" ]
```

The `FROM` indicates the so-called base image, an image to start from. In our case, we start from the official `python` image with the tag `3` (separated from the base image name with a ":"). The `ADD` command adds your Python script (here, "main.py" but could be any other name) to the image, and finally `CMD` tells Docker to execute a command when the image is loaded.

To build an image based on the Dockerfile above (notice the name "Dockerfile" is required), you use the following command in the directory where the Dockerfile exists to build an image with the tag "myimage" ("-t" stands for tag):

Code
```
$ docker build –t myimage .
```

The "." at the end of the command indicates that we want to build a container using files in the current working directory, which is why you also need to place the application `main.py` in the current directory. After building the image, you can run a container with the following command:

Code
```
$ docker run myimage
```

In the command above, `myimage` is the name of the image which we built before and which still exists in the local Docker repository.

Docker Hub is a location where the Docker images can be stored in order to be publicly accessed and used by developers to quickly produce fresh and composite applications. Thanks to the `push` and `pull` commands, we can write and read the images to and from the Docker Hub (Docker, 2013).

Docker has become very popular for application development, as its advantages vastly outnumber the disadvantages, especially for web applications running on the cloud. Docker containers are especially effective when paired with a container orchestration system, such as Kubernetes.

Cluster Computing

One of the defining characteristics of big data is that a single computer is not enough. It hence becomes necessary to use multiple computers that can share the computational load of data analytics. This is often called a computing cluster or distributed system for parallel computing. Distributed systems offer additional challenges over single machines, particularly in the sharing of data and tasks and the communication between machines within the cluster. The goal of distributed software is often to take care of these challenges so the user only has to worry about the task at hand.

Kubernetes

Orchestration is the automated configuration and management of computing systems and applications. Container orchestration automates the deployment, management, and scaling of containerized applications.

Kubernetes (often written as K8s for short) is a container orchestration system. It is an open-source project and was initially released in 2014 (Kubernetes, 2014). Kubernetes is very powerful and can manage systems distributed over thousands of computers.

Once we have built an application based on containers, the need for a tool such as Kubernetes becomes apparent. While having a single copy of the application on a single machine is manageable by a human, once the application needs to be scaled to hundreds or thousands of computers, the task is not humanly possible anymore. Given a desired structure and configuration of the application, Kubernetes makes sure that the desired state is always preserved.

Kubernetes is usually installed on a cluster of computers. Each computer is called a node. Nodes run Pods, which is the Kubernetes abstraction layer that runs one or more containers. Containers running in a pod are guaranteed to run on a single node. One or more pods can run on a single node. The master node does not run any pods and only serves as controller for the Kubernetes cluster.

Kubernetes allows **load balancing**: It can monitor the traffic to a service, and when it reaches a user-defined threshold, it will increase the size of the application by starting more containers automatically.

Load balancing
Load balancing is the attempt to distribute a workload over a set of resources in an efficient way, so that no resources are left idle or overwhelmed, but all are equally used.

Another feature is self-healing: in the event of a crash, Kubernetes makes sure that a new container is started to avoid any shortage of resources. Automated rollouts and rollbacks are a great feature for application deployment. Rollouts are deployments of new releases of an application: the user can describe the state of the new deployment, and Kubernetes can transition to the new state gradually, at a controlled rate determined by the user. Rollbacks are deployments of an application to a previous state or release. They are the same process as rollouts but in a different direction. Kubernetes allows for transitioning between these states in a seamless and fine-tuned way, minimizing the possibility of downtime for the application. Downtime is defined as the time when an application is not available or fails to work as expected, and is a key metric when maintaining applications, especially on the web.

In conclusion, Kubernetes is a very powerful tool that makes it possible to manage massive applications based on containers. It has become so popular that most applications designed to run on distributed systems are now compatible with it and take advantage of its ability to manage computing clusters. However, it is not without its difficulties. Kubernetes is a complex software with a steep learning curve that requires a considerable time investment before it becomes fruitful. It is also still in development, so keeping up with new features and adaptations requires more effort. Overall, while it can bring immense benefits to massive applications with millions of daily users, it remains overkill for applications of small size.

Mesos and DC/OS

Apache Mesos is a platform for managing computing clusters. It is an open-source project, first released in 2011 (Mesos, 2011).

Mesos aims at simplifying the usage of computing clusters by abstracting away the detail and complexity of CPU, memory, and I/O. The management of the machines composing the cluster is taken care of by Mesos, and applications running on it effectively see a single machine. Within this context, Mesos has also been called a distributed computing kernel. Like traditional OS kernels such as Linux or Windows, Mesos acts as a convenience layer between software and hardware.

Several distributed-computing analytics frameworks can take advantage of Mesos. Most notably Apache Hadoop and Apache Spark can run on top of Mesos, but theoretically any distributed application can take advantage of it.

DC/OS, which stands for “datacenter operating system”, is an open-source product developed by the company Mesosphere (DCOS, 2016). It was first released in 2016 and is built on top of Mesos. Continuing the analogy with single-machine computers, DC/OS is an operating system for distributed computing clusters. In this sense, DC/OS makes it easy to

install and use distributed applications. Ideally, it makes using a distributed system more like using a single machine, by taking advantage of Mesos and adding more user-friendly features.

DC/OS makes it easy to install software on distributed systems, with just a few clicks on a graphical interface. This has made it a perfect companion for cloud service providers, where managed services have been a staple for several years. The level of abstraction and ease of use provided by DC/OS does not come without a cost. Some of the biggest concerns about DC/OS have surrounded the limited support available, partly due to the smaller community (compared to other projects like Kubernetes). Another disadvantage of DC/OS is the scarce level of optimization, due to the high level of abstraction and consequent inability to fine-tune the cluster. Finally, autoscaling is not well supported, which makes it a far less ideal choice with respect to solutions such as Kubernetes.

5.2 Big Data Architectures

Throughout this course book we've been studying concepts and components that fall under the umbrella of big data. With the introduction of tools such as Kubernetes and DC/OS, using containers, we've seen how big data technologies can be deployed at scale. However, companies still face the issue of how to assemble a complete architecture that fits their needs. To this end we'll now consider three concrete architecture templates. Before doing so, let's recap and summarize which components are relevant for a big data solution. Note, however, that the overall technical infrastructure of any company will consist of many more parts not related to big data.

- Data sources: Big data systems have to deal with many, potentially heterogenous data sources, which exhibit a wide range of different data.
- Extraction, collection, and aggregation: All data sources have to be processed as needed, which might require streaming engines or batch processing systems, but could be as simple as an Extraction-Transform-Load (ETL) step to transform data in a classical, relational database. This step varies greatly in complexity, depending on your data sources and specified needs of downstream applications.
- Storage: Obviously related to the first two points, your data have to be suitably stored in one or several databases, depending on your needs. Often data need to be duplicated across several storage solutions to serve different needs, e.g., fast querying of data versus ensuring high availability of production data.
- Analysis: Once your system is ready to take on, process, and store various data sources, further processing it in analytical tasks, e.g., by creating machine learning models, is a crucial part for any big data system.
- Reporting and visualization: We haven't discussed this topic in this course book, but of course it's a vital part of any big data solution to make sense of the results you have attained in previous steps. This is achieved by having (automated) report generation for decision-makers, which very often also includes suitable data visualization. This topic, often framed in the context of business intelligence, is beyond the scope of this course book, but the necessity of reporting and visualization still needs to be taken into account when deciding on an architecture for your big data solution.

Microservices

Applications are often composed of several parts. A generic application could be described as composed by a frontend, a backend, and a database. In traditional application design (until around 2010) these different parts were all integrated in a single package and could be often intertwined in a way that made it difficult to see them separately. This kind of software design is often referred to as monolith architecture, as it's effectively a single block of inseparable components. Popular online businesses, such as Amazon and Netflix, realized that monolith software architectures were not able to keep up with their demands in terms of users' growth and consequent scalability needs.

This is where the concept of microservices comes in. The microservice design paradigm is aimed at keeping each part of an application separate, effectively running as small, independent applications. Each microservice which is part of the main application will communicate with other microservices via HTTP, each using their own application processing interface **(API)**.

Application programming interface (API)
An application programming interface (API) is a generic interface to a computing system or an application. On the web, the REST API paradigm using the HTTP protocol has become ubiquitous for interfacing with web services.

Microservices have several advantages with respect to monoliths. First, they are language-independent: the use of protocols like HTTP for communication means that, internally, each microservice could be written in a different programming language. This allows the programmers to choose the best language for the job, instead of writing everything in a single language, which is usually the constraint of a monolith. Another advantage is that when a microservice fails, it does not make the whole application crash. The damage can remain limited to that microservice, and it's easier to fix the problem or quickly replace the service. Scalability is another very strong point of microservices: small applications can be more easily started and stopped on-demand than a single large application and therefore are very suitable for the needs of large online businesses. Microservices are advantageous also from a security standpoint: in a monolith, whoever has access to the application can access all components, while microservices only get the minimum permission to access the required other parts of the application. Therefore, accessing a microservice does not give access to all the other services in the application, which limits the damage in case of a cybersecurity attack. Ultimately, microservices are the perfect use case for containers, which make deployment and scaling doable and effective.

Microservices do not come without disadvantages. The complexity of the communication network between services can become overwhelming and hard to manage. Similarly, the management of microservices, in terms of maintenance, deployment, and scaling, is quite complex and has some extra overhead with respect to monoliths. This is where container orchestration systems such as Kubernetes come into play.

The microservices approach is becoming more and more popular, and it is very well suited to work with and leverage the strengths of orchestration tools like Kubernetes. However, microservices describe a general set of principles that don't necessarily address the concrete pain points you face when building a big data solution. For instance, when you have a business that requires you to both process incoming data in real-time (e.g., for alerting operations when something goes wrong) while also making sure the same data are aggregated and processed later (e.g., for training a machine learning model), then microservices don't give you concrete answers for how to deal with such a scenario. All that could be

said is that you'd likely have to have two separate, containerized applications: one for real-time processing and one for your analytics task. But how exactly to architect such a solution would not be clear. The next two architectures are more specialized to big data solutions.

Lambda Architecture

Lambda is an architecture designed to handle large amounts of data that need to be processed both by a batch-processing and a stream-processing system. Lambda is based on three layers.

Here's a more detailed description of each layer. The basic assumption is that new data feeds into the system continuously and needs to be processed in different ways.

- The batch layer receives all incoming data (simultaneously with the speed layer) and is used for long-running, more complex data processing steps carried out on a schedule, usually not more often than once a day. In this layer you find the "master data", i.e., this is where you should go for all system-critical data queries. This layer usually has more traditional building blocks, like a data warehouse using relational databases, but that ultimately depends on the use case.
- The speed layer, also sometimes called the stream layer, is concerned with processing streams of data that haven't passed through the processing steps of the batch layer yet. This is done so that you have a complete view of your data, as the batch layer might only process your data once a night or so. As you're working with more real-time data and only have a snapshot of more recent data, this layer has less reliable information.
- The outputs of both the batch and speed layers are consolidated in the serving layer, which is optimized for data querying. In this layer you have both views of the real-time data from the speed layer and batch views for outputs from the batch layer.

Lambda architecture takes its name from the way it splits data into two separate systems, schematically making the architecture look like the Greek letter *lambda* ("Λ").

The benefit of Lambda architecture is that you have both stream- and batch-processing systems at your disposal, which can be quite useful for complex use cases. Your application will also be scalable, as each individual layer can be flexibly scaled up or down as needed. Lambda architecture also ensures low-latency queries of real-time data.

One of the core challenges of Lambda architecture is its high complexity and the fact it can be difficult to realize in practice. For instance, developers usually have to maintain two code bases for batch and speed layers, which can be a difficult to keep in sync.

Kappa Architecture

Kappa architecture was created in direct response to Lambda architecture and can be seen as both constructive criticism and simplification of the latter. In essence, Kappa architecture states that, if done correctly, you can do without the batch layer and simply process all information in a single real-time layer, while still retaining the serving layer.

The key idea, which is somewhat counterintuitive, is that you can do the same sort of high throughput data processing tasks that you would have done with a batch-processing system with a stream-processing system, too. To do that, it's crucial to have a system in place that ensures re-computation of your processing if something changes (e.g., code has been updated or some part of the data were faulty and needed to be fixed at the source). The real-time component chosen to keep track of the original data is often Apache Kafka, which you can configure to retain your data should they need re-processing.

One of the key advantages of Kappa architecture is that it is conceptually simpler than Lambda architecture. This architecture is, by its nature, particularly well-suited for real-time applications. One potential drawback arises from the fact that cloud services often don't allow for data retention over a long enough period to ensure the full re-computation capabilities of Kappa architecture. This is why this architecture might not be available to you when you build your infrastructure in the cloud.

SUMMARY

In this unit you've learned some of the basics of building scalable infrastructures for big data systems and how to build an architecture capable of dealing with big data.

First we introduced virtualization and discussed container technology, in particular Docker. We then had a look at how to use such technology to orchestrate clusters of computers with Kubernetes or Mesos DC/OS.

Then we discussed some common architectures that should be considered when building big data systems, i.e., microservices, Lambda architecture, and Kappa architecture.

BACKMATTER

LIST OF REFERENCES

Bernstein, P. A., & Das, S. (2013). Rethinking eventual consistency. *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (pp. 923—928). Association for Computing Machinery.

Codd, E. F. (1970). A relational model of data for large relational databases. *Communications of the ACM, 13*(2), 377—387. Association for Computing Machinery.

Connolly, T. M., & Begg, C. E. (2005). *Database systems: A practical approach to design, implementation, and management.* Pearson Education.

Date, C. J. (2007). *Logic and databases: The roots of relational theory*. Trafford.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., & Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review, 41*(6), 205—220.

Dinsmore, T. W. (2016). In-memory analytics. In *Disruptive analytics*. Apress. https://doi.org/10.1007/978-1-4842-1311-7_5

Dunning, T., & Friedman, E. (2014). *Time series databases: New ways to store and access data*. O'Reilly.

Fox, A., & Brewer, E. A. (1999). Harvest, yield, and scalable tolerant systems. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems* (pp. 174—178). IEEE.

Fuller, M., Moser, M., & Traverso, M. (2020). *Presto: The definitive guide*. O'Reilly.

Haerder, T., & Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR), 15*(4), 287—317.

Hueske, F., & Kalavri, V. (2019). *Stream processing with Apache Flink*. O'Reilly.

Pavlo, A. & Aslett, M. (2016). What's really new with NewSQL? *SIGMOD Record, 45*(2), 45—55. https://doi.org/10.1145/3003665.3003674

Perkins, L., Redmond, E., & Wilson, J. (2018). *Seven databases in seven weeks: A guide to modern databases and the NoSQL movement.* Pragmatic Bookshelf.

Rodriguez, M. A. (2015). The Gremlin graph traversal machine and language. In *Proceedings of the 15th Symposium on Database Programming Languages* (pp. 1—10). ACM.

Sadalage, P. J., & Fowler, M. (2013). *NoSQL distilled: A brief guide to the emerging world of polyglot persistence*. Pearson Education.

Samadi, Y., Zbakh, M., & Tadonki, C. (2016). Comparative study between Hadoop and Spark based on Hibench benchmarks. *2016 2nd International Conference on Cloud Computing Technologies and Applications (CloudTech)* (pp. 267—275). IEEE. https://doi.org/10.1109/CloudTech.2016.7847709

van Dongen, G., & van den Poel, D. (2020). Evaluation of stream processing frameworks. *IEEE Transactions on Parallel and Distributed Systems, 31*(8), 1845—1858. https://doi.org/10.1109/TPDS.2020.2978480

Wright, D. (2014, September 17). *The advantages of a shared nothing architecture for truly non-disruptive upgrades*. NetApp. https://blog.netapp.com/blogs/the-advantages-of-a-shared-nothing-architecture-for-truly-non-disruptive-upgrades/

Yusof, M. K., & Man, M. (2018). Efficiency of flat file database approach in data storage and data extraction for big data. *Indonesian Journal of Electrical Engineering and Computer Science*, *9*(2), 460—473.

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.