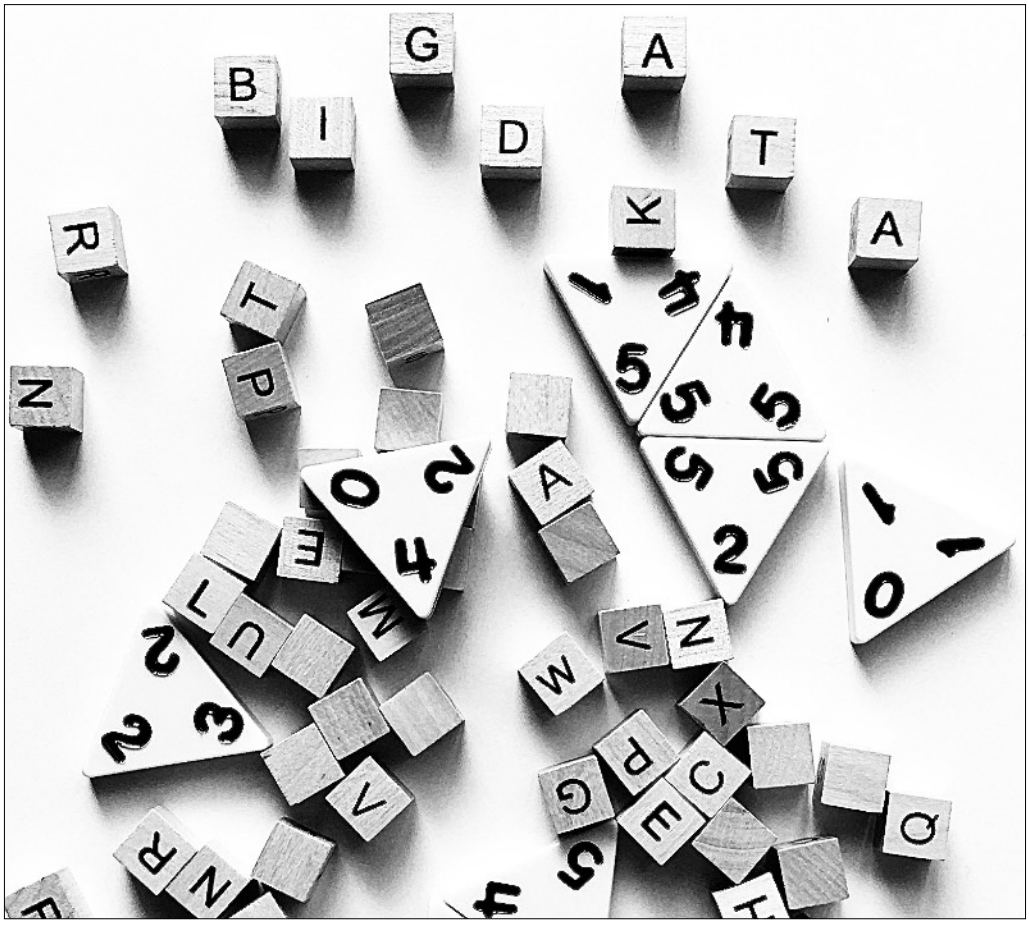
Piotr Fulmański

Engineering of Big Data Processing

Early Access Version
Edition 1.0, June 2022



Simple Introduction Series

Engineering of Big Data Processing

SIMPLE INTRODUCTION SERIES

www: https://fulmanski.pl
email: book@fulmanski.pl
GitHub: https://github.com/fulmanp/Engineering-of-Big-Data-Processing

Edition: 1
First published: 1.0, January XX, 202X (planned)
This edition: 1.0, June 2022 (early access)
Build number: 202206162324

ISBN-13: XXX-XX-XXXXXX-X-X

While the author has used good faith efforts to ensure that the information and instructions contained in this work are accurate, the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. The author makes no warranty, express or implied, with respect to the material contained herein.

If any code samples, software or any other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.

Release Notes

(**A** – add, **N** – new, **U** – update)

Edition 1.0
release: May and June 2022

- Add cover.
- A lot of changes, chapter renaming and new content. Too many to mention all of them here. Now chapters are: ch1: Introduction, ch2: Cleansing, transforming, and integrating data, ch3: Big Data concepts and terminology, ch4: Big Data paradigms, ch5: Storage concepts for Big Data. In preparation: ch6: Processing concepts for Big Data

Edition 1.0
release: April 2022

- Chapter: Big Data paradigms (**N**)
- Chapter: Processing concepts for Big Data (**N**)
- Chapter: Multiprocessing (**N**)
- Chapter: Message queues (**N**)

Edition 1.0
release: March 2022

- Chapter: Storage concepts for Big Data (**N**)

Edition 1.0
release: February 2022

- Chapter:Introduction (**N**)
- Chapter: Getting data (**N**)
- Chapter: Cleansing, transforming and integrating data (**N**)

Edition 1.0
release: April 2021

- Preface (**N**)
- Chapter: Big Data concepts and terminology (**N**)

Table of contents

Preface xi
Introduction 19
Engineering 21
Cleansing, transforming, and integrating data 25
ETL – Extract, Transform and Load 27
Data or garbage 31
Logs 55
Big Data concepts and terminology 59
What makes data to be considered as big 61
How you can describe values you have 71
Data 79
Most wanted properties of data 87
Different data usefulness 95
Big data characteristics 101
Summary 111
Big Data paradigms 113
Paradigm 115
Factors you should consider 125
Summary 139
Storage concepts for Big Data 143
Different data formats 145
Databases 149
Ways you work with data 155
Fact based data model 161

Data warehouse, lake, mart....167
Low level storage ideas....179
Processing concepts for Big Data....183
OLTP and OLAP....185
Paralelizm....189
Cluster, grid, cloud, fog and edge....193
Ways to process data....199
Map Reduce....209
SCV principle....223
Message queues....227
xxx....229
Adoption issues and considerations....231
xxx....233
Do we really need Big Data....235
xxx....237
xxx....239
xxx....241
Getting data....243
Cleansing, transforming, and integrating data....271
Multiprocessing....275
Bibliography....279
Bibliography....281

Preface

Why

In academic year 2017/2018 I introduced to didactic cycle on the faculty I have been working a new lecture: Big Data Engineering (formally: Engineering of Big Data Processing). You may ask: *Why not simply "Big Data"?* Is term engineering really necessary? I thought that Big Data are much different than "classic" data sets. The way you should think about it is different then data you used to work with. If you can apply everyday techniques you know well, there would be no need to talk about different kind of data; you will still consider just data. Prefix "big" suggests that you deal with something which is not any more a "typical" data. If not typical, why you should use typical techniques to process it? And this is the point. Big data, as a type of data, require different procedures to work with them. All practical aspects are different. You may say that *engineering* of big data processing is different. That is why the tile is *Engineering of big data processing*. I don't want to make another one book about big data where you can find (again) the same information that someone has already provided somewhere. **Every body talk about big data but nobody talk about *engineering* of big data.** Nobody talk that the way you perceive this type of data must be different. **You can find tones of big data blogs and examples where people solves problems which may be solved using old-school data processing approach. This is not big data. In this book I'm going to convince you that you have to change the way you think about data which are big. If you feel you don't**

have to do this, your data are not big. At most there is a lot of data, but it is not big data.

Who this book is for

This book is addressed to **all the people who want to understand how Big Data differs from Data and why they should be treated different way**. It may be good both for someone with no computer scientist background and for those who have some IT experience but want to put in correct order the whole dispersed knowledge one may have. I don't want to dive deep into details of specific technologies or solutions. Instead of that, I want to **explain why things are as they are**. Whenever it is possible I present a general way of thinking.

Early access

This book is a work in progress, presented in early access version. Early access allows to publish and share some ideas before the final version appears. This way, participating in early access, you may contribute how the final version will look like. English is not my native language and I know that I make a lot of mistakes but I hope that text is more than readable and at least a little bit understandable. I believe that everything can be better and there is always a space for improvements. I can say that the Toyota Way is the way I live and work focusing on continuous improvement, and respect for people. That is why I would be very grateful if you somehow contribute improving this book. Any comments, corrections and suggestions are more than welcome. I write this book not for myself but to share what I know with others, so help me make it's contents better.

If this book is buggy or not complete yet why it's not free? Simply because I need money to finish it and to make it better. Everything costs. The most precious is time. I, as all other people, have to work to

live and to support my family. And this consumes most of my days. Then I have a choice: play with kids or write a book. I choose playing with kids. I don't want to reduce a time spent with my family because I don't want to be a virtual parent. So I have just a little time for book. If I could reduce my job engagement (from full-time to half-time) I could spend more time on book.

I believe that there is no book like this on the market and I want to make it better and better. I can do this. I don't have to ask publisher if they agree to prepare another version. If something deserves for improvement I simply do this and publish right after that. Paying for a book you allow me to spending more time on book without sacrificing my family life. Having money I can pay for professional translation, text correction or simply buy better images.

What will you learn in this book?

I don't want to write this book and forget. My idea is to keep it as accurate and up to date as it is only possible so you can expect updates in a future even if I reach stable "final" version. As I wrote above, always there is something to improve. As for now book covers the following topics (chapter titles highlighted in red indicate finished or, if stated explicitly, in progress content).

Preface This is what you are reading right now. Here I explain what you can expect in this book. I also try to convince you to actively participate in shaping it's contents.

Chapter 1 Introduction In this chapter I try to define what Big Data is and why you shouldn't think about it as a "normal" data.

Chapter 2 Cleansing, transforming, and integrating data TODO

Chapter 3 Big Data concepts and terminology TODO

Chapter 4 Big Data paradigms TODO

Chapter 5 Storage concepts for Big Data TODO

Chapter 6 Processing concepts for Big Data TODO

What you will NOT learn in this book?

This book is not intended to be a user guide for a given Big Data technology or software. Even if I use some software I do this to show key features of a whole class of similar solutions, their pros and cons. Today everything changes so fast that it may happen that software used in examples is no more supported. Writing a book is a time consuming task and, especially in IT area, some fact described at the beginning of the book ma be outdated when author complete final chapters. It does not change the general nature of the considerations contained in this book as I made an effort to present base, universal and essential features which is quite common among different solutions of the same type. That is why **you will not find a chapters devoted to installation or configuration** (except some simple cases when necessary).

The number of topics covered in this book is quit big, and it shouldn't surprise you that some of them are only announced. There is no way to fully describe all of them. I don't think it's possible to know everything. There's a tremendous amount of information we get every day because the technology is evolving so fast. **Let me know if you think that some parts should be described in more details or in a completely different way.**

Give this book a try, and please let me know what you think. Any feedback is very much encouraged and welcomed! If you think that my time is worth this effort, you can support what I'm doing now and help me finalize this project. Please use email (book@fulmanski.pl) or GitHub (https://github.com/fulmanp/Engineering-of-Big-Data-

Processing/issues) to give your positive or negative, but in all cases constructive, feedback.

Thank you for your engagement.

Piotr Fulmański

Conventions used in this book

For your convenience I will use the following typographical convention:

Italic
Indicates new terms.

Italic
Indicates old terms but for some reason I want to distinguish them from normal text flow, definitions, citations.

`Constant width`
Indicates source code, filenames, file extensions, variables, parameters, etc.

`Constant width`
Indicates commands or any other text that you should type literally (as it is given).

`Constant width`
Indicates parts of scripts or commands which you need to pay special attention to.

Bold
Indicates statements which you need to pay special attention to. Sometimes it is used in combination with previous styles, for example:
`Constant width with bolded part`
This way I will mark for example crucial parameter in some important command.

```
This is how source
code is displayed
```

```
This is how a terminal
text is displayed with
bolded command prompt.
```

Something worth to remember or just one-sentence summary of some part of a section or chapter.

RULE

Most important dos and don’ts while working with big data.

NOTE

Note block

I use this block to give you some additional explanation or information, possibly loosely related to a main text.

CHAPTER 1

Introduction

What you will **learn**:

- Why big data requires its own engineering

SECTION 1

Engineering

Engineering

The *software engineering* term is well established in computer science and IT industry. In short it is about the systematic application of engineering approaches to the development of software. There are a lot of materials devoted to this topic, so I'm not going to talk about it. Instead I will recall definition which fits to my understanding of this subject (Software Engineering at Google[SOFENG:1]):

'software engineering' encompasses not just the act of writing code, but all of the tools and processes an organization uses to build and maintain that code over time. [...] Software engineering can be thought of as 'programming integrated over time.

Over all I prefer definition attributed to Ian Sommerville[SOFENG:1] because of it's compactness:

an engineering discipline [...] is concerned with all aspects of software production.

[...] all aspects of software production -- mostly all practical aspects. The term embodying the advocacy of a specific approach to computer programming when we should think about making software as an engineering enterprise rather than an art or a craft. In some sense, software engineering is about best software production practices. It help

you to focus on what, how and when you should do to create and then maintain your software.

As you can read in Preface, my opinion is that *big data* is much different than *data*. If this were not the case, then why would a new term such as big data be introduced? If you can apply everyday techniques you know well, there would be no need to talk about different kind of data; you will still consider just data. Prefix "big" suggests that you deal with something which is not typical data. If not typical, why you should use typical techniques to process it? And this is the point. Big data, as a type of data, require different procedures to work with them. All practical aspects are different. You can say, that *engineering of big data* processing is different.

It's not about best practise

Avoid to use *best practise* term as it kills innovations and leads to stagnation.

TODO

What people say about big data

Before I move on to the main part of the material, it is quite didactic to stop for a while and see what people say about big data. How they define it, how they perceive it. This may help me or you to focus on the most confusing concepts, highlighting what is essential part of working with big data and demystify the most common misunderstandings.

CHAPTER 2

Cleansing, transforming, and integrating data

What you will **learn**:

- What makes data to be considered as **big**
- How you can **describe** data
- What is **data polymorphism**
- Different **data usefulness --** what data may brings to you and how

ETL – Extract, Transform and Load

ETL term is used for the general procedure of *copying data* from one or more sources into a destination system which represents the data differently from the source. It is divided into three steps:

- *Data extraction* involves retrieving data from homogeneous or heterogeneous sources.
- *Data transformation* processes data by three sub-steps:
  - data cleansing,
  - data transforming,
  - data integration.
- *Data loading* describes the insertion of data into the final target system.

A properly designed ETL process extracts data from the source systems, enforces data quality and consistency standards, conforms data so that separate sources can be used together, and finally provide data in a ready to use format so that data may be used by developers, applications or directly by end users making decisions based on them.

Data extraction and data load, to a limited extent, is the subject of this chapter. Data transformation is discussed in the next chapter.

ELT – Extract, Load and Transform

TODO

ETL or ELT

TODO

Golden rule of every data scientist

ETL process may be very simple, like typical copy. Unfortunately, this is not a typical case. Usually you will change your data somehow in a way which may be irreversible. For example, if you cast doubles to integers or decrease image resolution there is no method to restore fractional part based on integers or increase image resolution and get back to original values. The golden rule of every data scientist is to always keep a copy of your original data.

> Always keep a copy of your original data.

Remember: storage space is cheaper than retrieval data again and sometimes repeating the getting data process is even impossible.

Data or garbage

Preventing *garbage in* to not get *garbage out*

Great, you are happy. You have your data. But this is just the beginning of a hard way. At the end of it, you will find a final presentation and automation (with AI – artificial intelligence, ML – machine learning) step proceeded by data exploration and data modeling steps. Before you reach it, you have to survive a long journey which, as every journey, is demanding, full of pitfalls and unpredictable traps. As every traveler, to minimize the risk of failure, you have to be equipped with tools you can rely on. In the world of data science it means that you have to use data that are clear, correct, consistent and in the right form.

> In the world of data science you have to use data that are clear, correct, consistent and in the right form.

The data received from the data retrieval phase is likely to be "a diamond in the rough". Your task is to sanitize and prepare it for use in subsequent phase. Doing so correctly is extremely important important for future processes. Why? There is a very simple answer: GIGO. What? GIGO – *Garbage in, garbage out* is a classic saying in computing about how *problematic* input data or instructions will produce *problematic* outputs. Let's be more literal: flawed, or *nonsense* (garbage) input data produces *nonsense* output. This distinction is important: with bad data you will not have problems as much as you will get nonsense, idiotic and

useless results. You can solve problems, but you can't fix something that is wrong from the ground up. This is especially relevant in ML. RIRO – *rubbish in, rubbish out* is an alternate wording for this concept. According to the CrowdFlower survey [Data:1], data preparation and cleaning take more than 50% of the time of data scientists and analytics professionals. This does not take into account the time needed to first collect and aggregate the required data for the problem at hand. According to the Toyota Way principle number 5: *Build a culture of stopping to fix problems, to get quality right the first time*. Other words, fix whatever you are working on as early as you detect any problems even if you have to stop in progress. It's a good habit to correct data errors as early on in the process as possible. The best is to completely avoid data errors. If this is not possible, try to minimize errors making data cleansing, transforming, and integrating. All of this steps fit into general transform phase in ETL process.

Cleansing data

Data cleansing focuses on removing errors and inconsistency in your data.

A good practice is to mediate data errors as early as possible in the data collection chain and to fix as little as possible inside your program while fixing the origin of the problem.

Now that we've given the overview, it's time to explain these errors in more detail.

Data entry errors

Data collection and data entry are error-prone processes. Especially when performed by human. Errors can arise from:

- loss of concentration (instead od 1.75 you may enter 17.5),

- typos like omission of a letter, swapping letters in places, entering an incorrect letter (instead of *frequency* you may enter *frequeny*, *freqeuncy* or *freguency*),
- human sloppiness (instead of *MiB* you may enter *mB*).

But data collected by machines or computers isn't free from errors either and arise from:

- machine or hardware failure,
- transmission errors,
- bug in one of the stage of data processing pipeline.

As an example, consider NMEA 0183 protocol (briefly referred to as NMEA) which is widely used in marine navigation electronics and GPS devices. The data is transmitted as sentences written in ASCII code. A single sequence contains up to 82 characters. The character starting the data in the protocol is `$`, followed by the sentence identifier and the data fields separated by commas, and finally the symbols `<CR><LF>` (carriage return, line feed). Below I present two sentences of this protocol — correct and incorrect:

```
$GPGLL,5149.66629,N,01925.96178,E,202100.00,A,A*62
$GPGLL,5149.66647,N,01925.9$GPRMC,202102.00,A,5149.66679,N,0192
5.96219,E,0.117,,230721,,,A*7A
```

As you can see, in the middle of the second `GPGLL` sentence another one (`GPRMC`) starts (marked with red color). It's highly probable that when `GPGLL` was saving, the power was down and sequence was unexpectedly cut. After power was restored, recording continued just after the last character (`9`).

Another real life example concerns medical data:

record number	gender	height	weight	Blood pressure	
				systolic	diastolic
1	1	151	67	120	80
2	1	157	93	130	80
3	1	55	81	130	90
4	1	57	61	130	90
5	1	170	64	10	70
6	2	168	59	-150	80
7	1	160	105	200	11000

Look closely at row number 3 and 4. Do you thing person of weight 81kg (178.5lb) or 61kg (134.5lb) may be 55cm (21.5in) or 57cm (22.5in) height? Or is this possible to have systolic blood pressure of value -150mm Hg or 10mm Hg (row 5 and 6). Diastolic blood pressure of value 11000mm Hg (row 7) should arouse our distrust as well.

If you think you will never be affected by this kind of error be aware that in 2013 a problem with scanners / copiers of the popular Xerox WorkCentre line was discovered that randomly altered numbers in pages that were scanned [BDP:1]. Patches of the pixel data were randomly replaced in a very subtle and dangerous way: the scanned images looked correct at first glance, even though some numbers actually were incorrect. As you can read in [BDP:1]:

We got aware of the problem by scanning some construction plan last Wednesday and printing it again. Construction problems contain a boxed square meter number per room. Now in some rooms, we found beautifully lay-outed, but utterly wrong square meter numbers. You really have to read the numbers to find out; this is why it is so hard to find out. In the present case, we found out because one room in the construction plan was — as the copy told us — about 22 square meters large, whereas the next room, a lot larger, was assigned a label with 14 square meters.

For example, a correct-looking scan at the first glance (check [BDP:1] to see an image of real scan):

```
  original                      copy

110.000 54.60               110.000 54.80
125.000 60.00               125.000 60.00
140.000 65.40               140.000 85.40
155.000 70.80               155.000 70.80
170.000 76.20               170.000 76.20
```

contains an error. It was found because usually, in such cost tables, the numbers are sorted ascending.

The error does occur because image segments, that are considered as identical by the pattern matching engine (not OCR) of the Xerox scan copiers, are only saved once and getting reused across the page. If the pattern matching engine works not accurately, image segments get replaced by other segments that are not identical at all, e.g. a 6 gets replaced by an 8. This is not a simple pixel error — on scan one can clearly see the characteristic dent the 8 has on the left side in contrast to a 6.

What is worse, the errors were caused by an eight year old bug and were not related to OCR so this could have affected thousands of documents — nobody knows how many.

Redundant white characters

White characters, especially whitespaces may turn out to be a nightmare for you — haven't you lost a few hours in your work because of a "bug" that was caused by whitespaces somewhere in a string?

Consider the following sentence: *Let the force be with you.* It might be difficult for you to say what is wrong in sentence: *Let the force be with you.* or *Let the force be with you.* It will be much easier if you compare

them with each other. The following sentences, although it may seem so, are not the same strings:

Let the force be with you.
Let the force be with you.
Let the force be with you.
Let the force be with you.
Let the force be with you.
Let the force be with you.

It gets less confusing if you use monospaced font and replace space with **o** and tab with **< TAB >**:

```
>Let the force be with you.<
>Let the force be with you.o<
>Let the force be owith you.<
>oLet the force be with you.<
><  TAB  >Let the force be with you.<
>oooooooooooLet the force be with you.<
```

Remember, to remove all unwanted whitecharacters as early as possible. It should be a kind of unconditional reflex as in my case is saving my job every few new sentences I type in text editor — I don't think about this, simply I press **Command + S** and I don't trust autosave option.

Letter mismatches

Quite common type of error. You can distinguish few types:

- omission of a letter: *Let the foce be with you;*
- extra letter: *Let the forcee be with you*
- incorrect letter: *Let the forse be with you*

- rearrangement the order of two adjacent characters:
  Let the force be wtih you;

- incorrect letter case: *Let the forCe be with you.*

Quite easy to catch if you only have a complete dictionary. May be difficult or even impossible to identify errors in proper noun like names of cities, clubs, bars, etc. as well as people's second names.

Impossible values and coherence check

Here you check the value against physically or theoretically impossible values such as systolic blood pressure of value -150mm Hg example given earlier or size of all files saved on hard drive which exceeds physical capacity.

Outliers

An outlier is a data that seems to be distant from other of that class. You should identify them, save in log file, and pass to data scientist who will work with them. You may also ask the data provider if values much different than others are really correct. Never ever change them or "repair" on your own. Remember that outlier data may follow a different logic than data obtained from "normal" cases and can stil be valid. Valid or even very precious as they can provide informations about unknown, different than typical, cases.

TODO Add plot

Dealing with missing values

If you have missing values in your data it does not automatically mean that something is wrong but for sure you should look at them carefully.

- Are missing values a problem for you at all?

- Why are they missing?

- In which records?
- Are missing values correlated with any other values?

Very common, due to its simplicity, is setting missing value to `null`. However take into account that `null`s are like dreadful poison and propagates in system like NaN (not a number) values in computations. When you decide to fix missing values, you have few methods to choose from (starting from the simplest to the most advanced):

Method	Pros	Cons
Omit the values	Easy to perform	You may lose the data. Sometimes entails the need to delete the entire record.
Use an arbitrary value such as 0 or the mean	Easy to perform You don't lose information from the other variables in the same record	Can lead to false estimations from a model
Imput a value from an estimated or theoretical distribution	Does not disturb the model if you only assume a correct and applicable estimation or data distribution (which may be not so simple and obvious)	Difficult to perform You make data assumptions which may not correspond to reality

Method	Pros	Cons
Modeling the value	Does not disturb the model if you only have a correct model which may not be so obvious	Difficult to perform – you should have or create a model which may be not simple You make data assumptions which may not correspond to reality Can lead to too much confidence in the model

Deviations from a code book

A code book describes the type of data expected at each column as well as their "ranges":

Type of data	Range
Number	All integers
	All positive integers
	Integers in range [-50,+50]
String	Unlimited length
	No longer than 30 characters
	Exactly 10 characters
	Exactly two words
Boolean	0 or 1
	T or F
	Y or N

Type of data	Range
Three state logic	Yes, No, Unknown
	1, 0, -1
	Yes, No, Sometimes
Discrete set	red, green, blue, white, black, yellow

Validating your data against code book in most cases is fast and easy to perform.

PUT DATA ON A CORRECT ORDER

Sorting your data is a time consuming task. Think if you really need it. Sometimes it doesn't matter. For example, counting sum of all orders is not dependent on data order. Conversely, analysing time series data, like exchange rates, must preserve their order. Sometimes, even for non time series data you may want to keep them sorted because of they way you will use them. For example,

TODO SQL example

Transforming data

In cleansing stage you don't change your data. If something is correct you keep it as it is; if something is incorrect you fix it. Transforming stage is different — now you will change your data, in most cases, with future use in mind. Type and scope of transformation is dependent on the software, algorithms or processes you plan to use.

CHANGING DATA FORMAT

On computers every data is stored and processed as sequence of zeros and ones. To make it possible, these sequences are created in a specific way called a *data representation format*. Many different formats exists, and if you want, you can create your own. html, pdf, png, mp3, exe,

docx, pptx — these are just a few of the elementary examples that you have surely heard. There are also formats dedicated to less complex data, like numbers (for example: IEEE 754; formally: IEEE Standard for Floating-Point Arithmetic) or characters (for example: UTF8, ISO 8859-2). In most cases you change format to unify data and to make its processing possible.

For example, numbers like 1.1 and 2.2 do not have exact representations in binary floating point. In consequence 1.1 + 2.2 is display as 3.3000000000000003 which is not what you expect to see. In binary floating point, the result of 0.1 + 0.1 + 0.1 - 0.3 is 5.5511151231257827e-017. While near to zero, the differences prevent reliable equality testing and what is worse such errors accumulate. If handled incorrectly may be a source of many hard to find problems, incorrect analysis and results. That is why you should never use binary floating point for monetary data and replace it with decimal.

Another example concerns document transformation from XML to JSON or changing weight from pounds to kilograms.

Changing data relationships

Sometimes relationships among data make difficult to analyse them. For example, if you look at Amazon's long-term stock line chart you may think that at the beginning, in 1999, company evolved very slow but started to grow very fast from 2015. This is because the increase in price from $56.74 to $86.09 in January and March 1999 is shown as if it was a meaningless move, and the stock appears to be almost standing still. But numbers say something different: increase of 51%. Contrary, in 2018 prices $1447.34 (March) and $2012.71 (August) gives increase of 39%. The difference is that the price of the Amazon stock hovered around $50 in January 1999, and in March 2018 hovered around $1400 — the $50 change in 1999 and 2018 has different contribution to the stock price. As you can see, processing numerical data over a very wide range of values expressed in linear scale may hide data relations. That is

why it may be justified to transform your data to express them in nonlinear scale, very typically a kind of logarithmic scale.

TODO add image (AmazonStock)

Reducing the number of variables

It's not surprising that in big data world you have a lot of data. Unfortunately, the more data you have, the more difficult is to say if you really need them. Having too many variables makes your model difficult to handle. In consequence it is always a good practice to verify your values if they are not redundant and bring you something useful. Redundancy in this context doesn't just mean exactly the same values but also data you may estimate with acceptable error using other, already owned, data.

PCA, Principal Component Analysis, is one of the technique you may use to reduce the number of variables. Consider the following exemplary set of data

TODO add image (PCA)

Turning variables into dummies

Trying to limit the number of data is by all means commendable but sometimes you have to do exactly the opposite — increase the number of variables. Whether it is actually increasing is hard to say. For example, imagine you have data you want to use as an input for artificial neural network. Base on the way artificial neural networks work, you should provide only pure numerical values. So how to code e.g. a day of the week. First attempt may be:

```
Monday:    1
Tuesday:   2
...
Saturday:  7
```

Although such a coding is theoretically correct (different numbers to denote different things) in practice values are too close, and may be skipped as irrelevant by neural network. It is much more better to use the following encoding:

```
Monday:    1,0,0,0,0,0,0
Tuesday:   0,1,0,0,0,0,0
...
Saturday:  0,0,0,0,0,0,1
```

This way you artificially increase the number of data. In fact, you increase the size of input vector but not the size of data because seven-components vector `[0,0,0,1,0,0,0]` still codes one day of the week, a Wednesday. In this case, you use dummy variables taking only one or zero to indicate the presence or absence of a categorical effect that may explain the observation. One is used as an indicator that something has happened on a given day and not any other.

PREPARE TIME DATA

Time data as no other data type is strongly related to time. You may take it for granted, but it has serious consequences. Every single time data has exact "location" on time axis. It means, that value is valid only at one point. Exactly at that point. It's not valid fraction of a time earlier or later. This significantly complicates the matter of processing such data. You should not assume that your time data has been collected at regular intervals. Of course, it would be best, but it is worth preparing for a different situation because in many cases you will work with unevenly spaced time series data.

Consider the following *unevenly spaced time series data* (time given in microseconds counted from sytem start):

```
number time  delta_time  value
1      20     0          10
2      29     9          15
3      44    15          13
4      58    14           8
```

```
5       74     16            9
6       87     13           15
7      105     18           16
8      141     36           20
9      161     20           18
10     169      8           15
11     185     16            9
```

As you can see, time difference between the two samples (`delta_time`) is not constant. Sometimes, as for data 3, 4 and 5 it is almost constant and close to average delta time which is 15. Other time it takes value twice as big as average (data number 8). Most tools require (assume) constant `delta_time`. You have no choice and you have to "discretises" your data, choosing constant `delta_time`. In intervals you obtain this way, you may define value as average of every data point belonging to a given interval, minimum value or any other method you want. For `delta_time = 10` you may obtain:

```
                                                 discretised
number time  delta_time  value  interval         time  value
1       20      0          10    [ 20,  30)        25
2       29      9          15    [ 20,  30)        25   (10+15)/2
3       44     15          13    [ 40,  50)        45   13
4       58     14           8    [ 50,  60)        55    8
5       74     16           9    [ 70,  80)        75    9
6       87     13          15    [ 80,  90)        85   15
7      105     18          16    [100, 110)       105   16
8      141     36          20    [140, 150)       145   20
9      161     20          18    [160, 170)       164
10     169      8          15    [160, 170)       165   (15+9)/2
11     185     16           9    [180, 190)       185
```

TODO add data plot

Of course, making discretisation you introduce an error to your data. Even if constant difference between the two samples is not strictly required, you will have problems in next stage, the integration, as I will describe it in a moment later.

Among all time data I distinguish a special case of data which I name: *time sensitive data*. As all time data it is valid exactly at a given point on time axis, but what is more important, over time, the data is less and less suitable for use. Consider salary. 30 years ago you earned x, now you earn y and probably $x < y$. But does this mean that you actually earn more? Can you buy proportionally more? Does it make sens to use both x and y as they are now and prepare learning data for artificial neural network to decide if grant or not a loan for the purchase of a house? Trying to make a decision based on historical data may lead to nonsenses because x now is worth much less than 30 years ago. For currencies, prices, salaries etc. you should make a transformation to expres them in a new units making them independent of the influence of time passage.

This idea is not new. Have you ever heard about Big Mac Index? Created in 1986, is a price index published by The Economist as an informal way of measuring the currency purchasing power [XXX:1-2]. The main assumption is that a sample basket of goods and services should cost "the same" over time. It means that x amount of money 30 years ago and y today are "equal" if you could then, and you still can, afford the same things or services.

Calculate derived and aggregated values

This type of transformation is intended to simplify and speed up processing. It is not so rare that anticipating future needs or the way data will be used, you may want to precompute some derived data or aggregate what you have. Imagine you have a list of GPS paths — each path is a sequence of locations. You can use such data to make for example a plot but if you want to say something more you have to compute it every time you want to display path. Some data are display (or generally used) so often that it doesn't make any sense to compute them gain, again and again. In case of GPS you may want to *aggregate* data and determine total travel time for each path or *derive* speed at every collected point.

Integrating data

Data varies in size, type, and structure, ranging from relational databases, key-value stores, document stores, Excel files to CSV files or even unstructured text documents. After cleaning the data errors and applying transformations, you *combine* information from different data sources.

During integration process you may face a problems of transformation nature but sometimes impossible to catch during transformation process as they become apparent only at the stage of data merging. Notice that I use term *problem* instead of *error*. In fact, at this stage all data taken separately is 100% correct — it is only when you start to connect them that problems arise.

Different units and formats

Units

When integrating two data sets, you have to pay attention to their respective units of measurement. An example of this would be when you gather data describing dimensions and weight of different goods in inches and pounds or in centimetres and kilograms. Another example worth to mention is fuel consumption. In Europe it is given in litters per 100km while in USA it is the number of miles per galon. A simple conversion will do the trick in this case.

Inconsistency

An example of this class of problems is putting `true` in one table and `T` or `1` in another when they represent the same thing: positive answer to the question. Another example is that you use millimetres (mm) or inches (in) in one table and centimetres (cm) or foots (ft) in another.

Time format nightmare

As a comment on this issue I will only give a few examples of date and time formats [XXX:3]:

Compliant with the ISO 8601:

```
yyyyMMddZ                       19990322+0100
yyyyMMdd                        19990322
yyyy-MM-dd G                    1999-03-22 AD
yyyy-MM-ddXXX                   1999-03-22+01:00
yyyy-MM-dd'T'HH:mm:ss.SSS'['VV']'
                              1999-03-22T05:06:07.000[Europe/Paris]
yyyy-MM-dd'T'HH:mm:ss.SSS     1999-03-22T05:06:07.000
yyyy-MM-dd'T'HH:mm:ss         1999-03-22T05:06:07
yyyy-MM-dd'T'HH:mm:ss.SSS'Z'  1999-03-22T05:06:07.000Z
yyyy-MM-dd'T'HH:mm:ss.SSSXXX  1999-03-22T05:06:07.000+01:00
yyyy-MM-dd'T'HH:mm:ssXXX      1999-03-22T05:06:07+01:00
yyyy-DDDXXX                     1999-081+01:00
YYYY'W'wc                       1999W132
YYYY-'W'w-c                     1999-W13-2
yyyy-MM-dd'T'HH:mm:ss.SSSXXX'['VV']'
                      1999-03-22T05:06:07.000+01:00[Europe/Paris]
yyyy-MM-dd'T'HH:mm:ssXXX'['VV']'
                      1999-03-22T05:06:07+01:00[Europe/Paris]
```

Compliant with the English, United Kingdom:

```
dd MMMM yyyy                      22 March 1999
EEEE, d MMMM yyyy Monday,         22 March 1999
dd-MMM-yyyy                       22-Mar-1999
dd MMMM yyyy HH:mm:ss z           22 March 1999 05:06:07 CET
EEEE, d MMMM yyyy HH:mm:ss 'o''clock' z
                       Monday, 22 March 1999 05:06:07 o'clock CET
dd-MMM-yyyy HH:mm:ss              22-Mar-1999 05:06:07
dd-MMM-yy hh.mm.ss.nnnnnnnnn a  22-Mar-99 05.06.07.000000888 AM
```

Compliant with the English, United States:

```
M/d/yy                            3/22/99
MM/dd/yy                          03/11/22
MM-dd-yy                          03-22-99
M-d-yy                            3-22-99
MMM d, yyyy                       Mar 22, 1999
MMMM d, yyyy                      March 22, 1999
```

```
EEEE, MMMM d, yyyy              Monday, March 22, 1999
MMM d yyyy                      Mar 22 1999
MMMM d yyyy                     March 22 1999
MM-dd-yyyy                      03-22-1999
M-d-yyyy                        3-22-1999
yyyy-MM-ddXXX                   1999-03-22+01:00
dd/MM/yyyy                      22/03/1999
d/M/yyyy                        22/3/1999
MM/dd/yyyy                      03/22/1999
M/d/yyyy                        3/22/1999
yyyy/M/d                        1999/3/22
M/d/yy h:mm a                   3/22/99 5:06 AM
MM/dd/yy h:mm a                 03/22/99 5:06 AM
MM-dd-yy h:mm a                 03-22-99 5:06 AM
M-d-yy h:mm a                   3-22-99 5:06 AM
MMM d, yyyy h:mm:ss a           Mar 22, 1999 5:06:07 AM
EEEE, MMMM d, yyyy h:mm:ss a z
                          Monday, March 22, 1999 5:06:07 AM CET
EEE MMM dd HH:mm:ss z yyyy      Mon Mar 22 05:06:07 CET 1999
EEE, d MMM yyyy HH:mm:ss Z      Mon, 22 Mar 1999 05:06:07 +0100
d MMM yyyy HH:mm:ss Z           22 Mar 1999 05:06:07 +0100
M/d/yy                          3/22/99

MM-dd-yyyy h:mm:ss a            03-22-1999 5:06:07 AM
M-d-yyyy h:mm:ss a              3-22-1999 5:06:07 AM
yyyy-MM-dd h:mm:ss a            1999-03-22 5:06:07 AM
yyyy-M-d h:mm:ss a              1999-3-22 5:06:07 AM
yyyy-MM-dd HH:mm:ss.S           1999-03-22 05:06:07.0
dd/MM/yyyy h:mm:ss a            22/03/1999 5:06:07 AM
d/M/yyyy h:mm:ss a              22/3/1999 5:06:07 AM
MM/dd/yyyy h:mm:ss a            03/22/1999 5:06:07 AM
M/d/yyyy h:mm:ss a              3/22/1999 5:06:07 AM
MM/dd/yy h:mm:ss a              03/22/99 5:06:07 AM
MM/dd/yy H:mm:ss                03/22/99 5:06:07
M/d/yy H:mm:ss                  3/22/99 5:06:07
dd/MM/yyyy h:mm a               22/03/1999 5:06 AM
d/M/yyyy h:mm a                 22/3/1999 5:06 AM
MM/dd/yyyy h:mm a               03/22/1999 5:06 AM
M/d/yyyy h:mm a                 3/22/1999 5:06 AM
MM-dd-yy h:mm:ss a              03-22-99 5:06:07 AM
M-d-yy h:mm:ss a                3-22-99 5:06:07 AM
MM-dd-yyyy h:mm a               03-22-1999 5:06 AM
M-d-yyyy h:mm a                 3-22-1999 5:06 AM
yyyy-MM-dd h:mm a               1999-03-22 5:06 AM
yyyy-M-d h:mm a                 1999-3-22 5:06 AM
MMM.dd.yyyy                     Mar.22.1999
d/MMM/yyyy H:mm:ss Z            22/Mar/1999 5:06:07 +0100
dd/MMM/yy h:mm a                22/Mar/99 5:06 AM
```

Compliant with the Lithuanian:

```
yy.M.d                          99.3.22
yy.M.d HH.mm                    99.3.22 05.06
yyyy-MM-dd HH.mm.ss             1999-03-22 05.06.07
```

GPS Format

After what you have seen in connection with the date and time formats, the ways of saving GPS data may seem to you simplistic and limited to a just few forms. Traditionally, there are a bunch of format based on latitude and longitude:

```
Latitude:               Longitude
41.40338                2.17403
41°24'12.2"N            2°10'26.5"E
41 24.2028              2 10.4418
N43°38'19.39"           W116°14'28.86"
43 38 19.39             -116 14 28.86
43.63871944444445       -116.2413513485235
N 1.286785              E 103.854503
```

Although specifying location as latitude and longitude is a common practice, it is neither very convenient nor easy to memorise and interpret. For this reason few other, simpler to use, formats exist. For example the Military Grid Reference System (MGRS) allows you to save location as a one string; the longer the string is the more accurately it describes position:

```
4Q                  GZD (grid zone designator )only
4QFJ                precision level 100 km
4QFJ 1 6            precision level 10 km
4QFJ 12 67          precision level 1 km
4QFJ 123 678        precision level 100 m
4QFJ 1234 6789      precision level 10 m
4QFJ 12345 67890    precision level 1 m
```

There are other systems like the Open Location Code (OLC, location codes created by the OLC system are referred to as *plus codes*),

Universal Transverse Mercator (UTM) or W3W (What 3 Words) just to mention the most common. For example, the place where I live can be coded as [XXX:3,4]:

```
Decimal degree              N 51.828368 E 19.433184
Degrees Minutes             N 51° 49.702080 E 19° 25.991040
Degrees Minutes Seconds     N 51° 49' 42.1248 E 19° 25' 59.4624
MGRS                        34UCC 92029.524 43109.822
UTM                         34U 392029.524 5743109.822
Plus Code                   9F3XRCHM+87
W3W (What 3 Words)          badly.majority.clash
```

TIMELINE YOUR DATA

Imagine you have multiple time series data. Maybe some of them have constant `delta_time` (difference in time between two subsequent time data), but it is almost impossible that two or more time series have their timestamp synchronised. Consider the following data:

```
time    series1  series2
 0      100
 3               37
 4      110
 6               41
 8       90
 9               38
12      100      39
15               42
16      120
18               40
20      115
21               41
```

Even if you think you have a lot of data you may find that there are only few moments when all series meet. In all other cases one of values is correct, while all other can only be interpolated. Another problem is variable `delta_time` — in the above example you obtain:

```
delta_time: 3, 1, 2, 2, 1, 3, 3, 1, 2, 2, 1
```

Different levels of discretization (aggregation)

Imagine you combine data coming from different restaurants (e.g. incomes, number of clients, the need for food ingredients) in some of them aggregated per week, in other separately per work week and per weekend and in another cases per each day. Sometimes you can unify them to have the same level of aggregation — for example, if you have a sum of incomes per each day you can add them to get sum of incomes per week. Be aware that this is not always possible — for example if you have an average of something. Also it would be impossible to "split" sum of incomes per week into sum of incomes per each day.

Different ways of combining data

Very often data integration entails physical data copy, move or reorganization.

Appending data

Limiting the considerations to the tabular form of data representation, appending data is equivalent to adding new record to one table with data taken from another table.

Joining data

Joining tables allows you to combine dispersed data. You may do this either logically, with foreign keys (FK) from one table referring to primary keys (PK) in another table as you do typically when you save your data in relational databases:

```
Before join                        After join
CSV files                          Relational database

CSV1          CSV2                 Tab1          Tab2
                                   PK            PK  FK
a 12 13 14    a 11a 12a 13a        a 12 13 14     1  a 11a 12a 13a
b 22 23 24    b 11b 12b 13b        b 22 23 24     2  b 11b 12b 13b
c 32 33 34    a 21a 22a 23a        c 32 33 34     3  a 21a 22a 23a
              c 11c 12c 13c                       4  c 11c 12c 13c
              a 31a 32a 33a                       5  a 31a 32a 33a
              b 21b 22b 23b                       6  b 21b 22b 23b
```

Another option is to combine physically data. This may lead to redundancy but allows to make faster queries and is typical for storing data in NoSQL systems:

```
Before join                        After join

CSV1            CSV2               Table
                                   PK
a 12 13 14      a 11a 12a 13a       1 a 12 13 14 11a 12a 13a
b 22 23 24      b 11b 12b 13b       2 a 12 13 14 31a 32a 33a
c 32 33 34      a 21a 22a 23a       3 a 12 13 14 21a 22a 23a
                c 11c 12c 13c       4 c 32 33 34 11c 12c 13c
                a 31a 32a 33a       5 b 22 23 24 11b 12b 13b
                b 21b 22b 23b       6 b 22 23 24 21b 22b 23b
```

Use views

A view behaves as if you're working on a table, but this table is nothing but a virtual layer that combines the tables for you. With view you can simulate data appends and joins without touching your precious data. Tempting? Not always. Not in datawarehouses. Not when speed is crucial.

Logs

TODO Short intro to the topic

At the top of log file you should save the checksum of the file for which this report is created. This will prevent you from accidentally using your log data to a similar but different file (which is not that hard to do).

Keep the scanning log in the form:

```
{"checksum": {"md5": "123", "sha1": "456"}}
{"line": 71, "errors": [{"column": 81, "type": "ERROR_1"}]}
{"line": 72, "errors": [{"column": 82, "type": "ERROR_2"}]}
{"line": 73, "errors": [{"column": 83, "type": "ERROR_3"}]}
{"line": 74, "errors": [{"column": 84, "type": "ERROR_4"}]}
```

Why so? Why not as valid JSON, e.g .:

```
{
  "meta": {"checksum": {"md5": "123", "sha1": "456"}},
  "log": [
    {"line": 71, "errors": [{"column": 81, "type": "ERROR_1"}]},
    {"line": 72, "errors": [{"column": 82, "type": "ERROR_2"}]},
    {"line": 73, "errors": [{"column": 83, "type": "ERROR_3"}]},
    {"line": 74, "errors": [{"column": 84, "type": "ERROR_4"}]}
  ]
}
```

This allows you to read data line by line without having to read the entire log file and put it into memory. Try to load the simplest, not complete JSON:

```
{
  "meta": {"checksum": {"md5": "123", "sha1": "456"}},
  "log": [
    {"line": 7, "errors": [{"column": 8, "type": "ERROR_1"}]}
```

Keep in mind that 99% of the time, no one will manually search and view the log file — it cannot be done for large data sets. Thus, such a file does not necessarily have to be comfortable for a human to read, as it will usually be further processed by the computer anyway. Therefore, you can safely use a more compact form, which will save a lot of disk space. The following form:

```
{"c":{"md5":"123","sha1":"456"}}
{"l":71,"e":[{"c":81,"t":"ERROR_1"}]}
{"l":72,"e":[{"c":82,"t":"ERROR_2"}]}
{"l":73,"e":[{"c":83,"t":"ERROR_3"}]}
{"l":74,"e":[{"c":84,"t":"ERROR_4"}]}
```

allows you to reduce the size relative to the initial file from 283B to 184B. It can be even shorter, reducing the file size to 168B (i.e. a size reduction of 40%):

```
{"c":{"md5":"123","sha1":"456"}}
{"e":[71,{"c":81,"t":"ERROR_1"}]}
{"e":[72,{"c":82,"t":"ERROR_2"}]}
{"e":[73,{"c":83,"t":"ERROR_3"}]}
{"e":[74,{"c":84,"t":"ERROR_4"}]}
```

Another example — simple JSON with places worth to see:

```
{
  "data": [
    {
      "country": "Poland",
      "location": "Łódź",
      "coordinates": {
        "lat": "51°46’36”N",
        "long": "19°27’17”E"
      }
    },
```

```
    {
      "country": "Peru",
      "location": "Arequipa",
      "coordinates": {
        "lat": "16°25’03”S",
        "long": "71°32’12”W"
      }
    },
    {
      "country": "Austria",
      "location": "Kals am Großglockner",
      "coordinates": {
        "lat": "47°00’N",
        "long": "12°38’E"
      }
    }
  ]
}
```

526B

To make it clear where line ends I use tag **<NL>**:

```
{"data":[<NL>
{"country":"Poland","location":"Łódź","coordinates":{"lat":"51°4
6’36”N","long":"19°27’17”E"}},<NL>
{"country":"Peru","location":"Arequipa","coordinates":{"lat":"1
6°5’03”S","long":"71°32’12”W"}},<NL>
{"country":"Austria","location":"Kals am Großglockner","coordin
ates":{"lat":"47°00’N","long":"12°38’E"}}<NL>
]}
```

339B

```
{"location":["Poland","Łódź",{"gps":["51°46’36”N","19°27’17”E"]}
]}<NL>
{"location":["Peru","Arequipa",{"gps":["16°25’03”S","71°32’12”W"
]}]}<NL>
{"location":["Austria","Kals am Großglockner",{"gps":["47°00’N",
"12°38’E"]}]}<NL>
```

204B, (i.e. a size reduction of 62%)

CHAPTER 3

Big Data concepts and terminology

What you will **learn**:

- What makes data to be considered as **big**
- How you can **describe** data
- What is **data polymorphism**
- Different **data usefulness --** what data may brings to you and how

What makes data to be considered as big

Definition

In many different sources a term big data is defined as a [BD: 1]:

[...] data sets that are too large or complex to be dealt with by traditional data-processing application software.[...]

In big data world data volume is important but it's not the only factor you should take into account. Saying the truth, voluminous data is not a problem if volume is isolated from other possible elements which potentially may influence processing. The problem becomes a challenge when more factors coexist at the same time. Reading carefully the rest of the definition mentioned above you will find the following sentence:

[...] Big data usually includes data sets with sizes beyond the ability of commonly used software tools to capture, curate, manage, and process data within a ***tolerable elapsed time****.*

This simple sentence changes drastically the way you perceive what big data is. Time factor plays crucial role. Now you can infer that size is not so important as more important is to complete data processing on time. There may be situation that your data is not voluminous but because of the way you process it, you will miss your deadline. In such a case your

data, even not so big in absolute terms, in practice is to big (to complicated) since you can't deliver the result on time

The second definition is my favorite. It is short, descriptive and including much wider ideas, I have found in [BD: 2]:

Big Data is a field dedicated to the analysis, processing, and storage of ***large collections*** *of data that* ***frequently*** *originate from* ***disparate sources***.

So big data is not only about data itself but, which seems to be more important, about everything we need to use large collections of data. Specifically, big data addresses distinct requirements, such as (but not limited to):

- the combining of multiple unrelated datasets,
- processing of large amounts of unstructured data,
- and harvesting of hidden information in a time-sensitive manner.

For the last few years you can observe something I name Big Data Madness. Everyone announce to use big data (together with cloud computing, machine learning and blockchain) either they think they need them or they must say so to be in the mainstream of contemporary state of the art technologies, according to the rule: *whether I need this or not, if my business competitors have this, then I must have one too*. Because of this, big data may appear as a new discipline, but it has been developing for years. The management and analysis of large datasets has been a long-standing problem; recall one of the best known: the US census[BD: 3].

In 1880, as new arrivals flooded into the United States and the population exploded, the US census turned into an administrative nightmare. The work of measuring and recording the fast-growing country's population was maddeningly slow and expensive. Clerks

would need eight years to finish compiling the census. So, barely finished one thing and you have to start the other. As the data processing results were obtained with so long delay, they were practically worthless due to the very high dynamics of social changes.

Next time, in 1890, the Census Bureau decided to use The Hollerith's Punched Card Tabulator machine to work on census. It did the job in just two years, and saved the government US$5 million. By the way we got an extra new value. Not only could the machines count faster, but they could also understand information in new ways. Rearranging the wires on a tabulating machine allowed users to learn things they never knew they could learn, and at speeds no one thought possible. And this is something really precious which makes a real difference. A traditional analytic approaches, based on statistics, approximate measures of something via sampling it. Big data adds to this a possibility to processing of **entire** datasets, making such sampling unnecessary and analysis more accurate.

A traditional analytic approaches, based on statistics, approximate measures of something via sampling it. Big data adds to this a possibility to processing of entire datasets, making such sampling unnecessary and analysis more accurate.

Surprisingly, the first application of big data approach dates back even earlier, to the ancient time [BD:4]. In 431 BCE, Sparta declared war on Athens.

TODO

Thucydides, in his account of the war, describes how besieged Plataean forces loyal to Athens planned to escape by scaling the wall surrounding Plataea built by Spartan-led Peloponnesian forces. To do this they needed to know how high the wall was so that they could make ladders of suitable length. Much of the Peloponnesian wall had been covered with rough pebbledash, but a section was found where the bricks were

still clearly visible and a large number of soldiers were each given the task of counting the layers of these exposed bricks. Working at a distance safe from enemy attack inevitably introduced mistakes, but as Thucydides explains, given that many counts were taken, the result that appeared most often would be correct. This most frequently occurring count, which we would now refer to as the mode, was then used to calculate the height of the wall, the Plataeans knowing the size of the local bricks used, and ladders of the length required to scale the wall were constructed. This enabled a force of several hundred men

to escape, and the episode may well be considered the most impressive example of historic data collection and analysis. But the collection, storage, and analysis of data pre-dates even Thucydides by many centuries, as we will see.

Talking about definition, I have to stress that the one quoted above is the best for me but not the only one. The whole spectrum of ideas focused under the big data name blends mathematics, statistics, computer science and subject matter expertise. This mixture has led to some confusion as to what comprises the field of big data and the response one receives will be dependent upon the perspective of whoever is answering the question. What is more, the boundaries of what constitutes a big data problem are also changing due to the ever-shifting and advancing landscape of software and hardware technology. This is due to the fact that the definition of big data takes into account the impact of the data's characteristics on the design of the solution environment itself. Long time ago, in 90s, I had a PC with 4MiB RAM and I could play the most demanding games, while one gigabyte of data was a big data problem requiring special purpose computing resources. Now, gigabytes of data are commonplace and can be easily transmitted, processed and stored even on a mobile phone.

The applications and potential benefits of big data are broad including, but not limited to

- optimization,
- identification,
- predictions,
- fault and fraud detection,
- improved decision-making,
- scientific discoveries.

However, please don't fall into big data madness and remember to considered numerous issues when adopting big data analytics approaches. These issues need to be understood and well weighed against anticipated benefits.

Final though

What makes data to be considered as big? Is this its size? Not exactly. Size is the factor you can easily measure. And in most cases, as size increases also data processing gets more and more complicated. For me big data is any data set based on which I can't provide useful results in required time. It may be a matter of size. But as well it may be a matter of processing complexity or any other factors.

Let's say that you process data coming from LHC (Large Hadron Collider). In this case, it is really a big data in terms of size. It is enough to say that the LHC experiments produce about 90 petabytes of data per year [BDS:4].

NOTE

LHC data size

To have a relation to something you know from real life you can find how big storage you need to save it. When I was young very often as a capacity unit a number of CDs was used. How much CDs do you need in this case? Let's do simple math:

- 1 pebibyte is 2^{50} bytes (1125899906842624 bytes);
- 1 mebibyte is 2^{20} byes (1048576 bytes);
- 1 CD is 700 MiB;
- 90PiB is approximately 1533916.89 * 700 MiB, which is approximately 138052521 CDs;
- CD thickness is 1.2mm;
- CD weight varies from 14 to 33 grams (average 23.5g);
- the tower formed of cd's would be 165.66 kilometres high ($(138052521 \cdot 1.2)/1000000 \approx 165.66$) and a mass of 3244.23 tons ($(138052521 \cdot 23.5)/1000000 \approx 3244.23$).

If this is too much for you, you can change unit and express size in DVDs instead of CDs:

- 1 gibibyte is 2^{30} byes (1073741824 bytes);
- 1 DVD is 4.7GB which is 4.38GiB; approximately 4702989189 bytes;
- 90PiB is approximately 21546083 DVDs;
- DVD thickness is 1.2mm;

- DVD weight varies from 14 to 33 grams (average 23.5g);
- the tower formed of cd's would be 25.86 kilometers high ($(21546083 \cdot 1.2)/1000000 \approx 25.86$) and a mass of 506.33 tons ($(21546083 \cdot 23.5)/1000000 \approx 506.33$).

For really big sets of data even the number of DVDs become so large that it again becomes abstract. This way more voluminous units are used – for example well known library, like the American Library of Congress.

Unfortunately now you can only estimate its size and it varies from 6TiB (*[...] stores some 32 terabytes (32,768 GB) of data. That's five times more than the world's largest library – the US Library of Congress*) to 250TiB (*[...] keeps 500 terabytes of storage [...] that's about twice the amount needed to hold the entire Library of Congress*) [BDS: 1, 2]. If you are interested in where these large discrepancies come from, read [BDS: 2]. Assume that it takes 200TiB which is 0.2PiB. Than LHC every year produces $90/0.2 = 450$ times more data than is stored in the American Library of Congress.

Obviously, such a large amount of data requires a sufficiently long time to process them. But it doesn't matter if you get your results in 3 months or 3 weeks. No one will die because of this, the results will be just as useful in 3 months as in 2. Only scientists will be more and more impatient...

For comparison, stock market data is orders of magnitude smaller than the LHC data but results are needed right now. You have to collect them, filter, save, process and provide useful output within seconds and it must be so as long as market is open. In this case also size of data is a problem — less data you have, highly probable is that you complete the

whole process on time. But considering only the absolute size of the data, it is easy to see that these problems are incomparable.

How you can describe values you have

In this section I present some terminology and concepts related to data. I will show how you can characterize data you have.

Property types

I intentionally use in this subsection a term *property* because it let me define a little bit later what data is. Property describes somehow given *object* or *thing*. It may be its size (width, height etc.), color, taste, the group to which you include it (high people, speed cars, nice wether etc.), subjective opinions — anything you can say about it. There are two main types of properties: *qualitative*, *quantitative*.

QUALITATIVE AND QUANTITATIVE PROPERTIES

Qualitative properties are properties which can be observed but cannot be computed nor measured with a numerical result. Qualitative data is about the emotions or perceptions of people, what they feel. You use this type of properties to **describe a given topic in more abstract way** including even *impressions*, *opinions*, *views* and *motivations*. This brings depth of understanding to a topic but also makes it harder to analyze. You consider this type of properties as being unstructured. When the data type is qualitative the analysis is non-statistical. Qualitative properties asks (or answers) the question: ***Why?***

Advantages of qualitative properties
It helps in-depth understanding: you can ask about things which are not visible at first sight or even at all:

- *What do you think about…?*
- *Does it smell nice?*
- *Do you recommend it?*
- *Is it comfortable to wear it?*
- *What you think about its height?*

It helps you to understand what people think, how they perceive something. Since the questions asked to collect qualitative data are open-ended questions, respondents are free to express what they feel.

Disadvantages of qualitative properties
Collecting qualitative data is much more time consuming. What you collect strongly depends on you and your respondents involvement. You have to ask right questions, people have to provide honest answers and you have to note them correctly. Because nature of answers is very subjective they are not easy to generalize.

Quantitative properties are focuses on numbers and mathematical calculations and can be calculated and computed. We consider this type of properties as being structured and statistical. Quantitative properties asks (or answers) ***How much?*** or ***How many?***

Advantages of quantitative properties
Due to the numerical nature of quantitative data, the personal bias is reduced to a great extent. As the results obtained are objective in nature, they are extremely accurate. Quantitative data can be statistically

analyzed — if it is done correctly it minimizes personal biases and attitude to something.

Disadvantages of quantitative properties

You get very precise but limited answers. If for example you ask about height in centimeters you will get nothing else than object's height in centimeters. This is much different than qualitative question: *What you think about its height?* What you collect strongly depends on you.

Factor	Qualitative	Quantitative
Meaning	The data in which the classification of objects is based on attributes and properties	The data which can be measured and expressed numerically
Type of data	Unstructured	Structured
Analysis	Non-Statistica	Statistical
Used to	Get initial understanding.	Recommends final course of action.
Methodology	Exploratory	Conclusive
Question	Why?	How many or How much?

Discrete and continuous properties

Your properties can be understood as the quantitative information about a given topic. If the property characteristic is qualitative it should be transformed into quantitative one, by providing numerical data of that characteristic (you have to map qualitative information into numbers), for the purpose of statistical analysis.

The quantitative characteristic can be considered as being discrete, or continuous.

Discrete is a type of data that can assume only fixed number of distinct values and sometimes lacks an inherent order. Also known as a *categorical*, because it has separate, invisible categories.

Continuous is a type of statistical information that can assume all the possible values within the given range. If a property can take an infinite and uncountable set of values, then the property is referred as a continuous.

Both definitions work in real, analog, word. In digital word everything is discrete; every analog data is changed into discrete data. So, how can you tell the difference between these two properties? To answer this question you need a concept of *isolated point*.

In mathematics, a point x is called an *isolated point* of a subset S of space X if x is an element of S and there exists a neighborhood of x which does not contain any other points of S. If the space X is a Euclidean space, then x is an isolated point of S if for any (small) ε there exists an open ball of radius ε around x which contains no other points of S. For example, if X is a real line and subset is defined as:

$$S = \{0\} \cup \{1, 1/2, 1/3, \ldots\}$$

then each of the points $1/k$ is an isolated point, but 0 is not an isolated point because there are other points in S as close to 0 as desired. A set that is made up *only of isolated points* is called a discrete set.

In computer science the space X is a set of all the numbers you can represent on a computer. In consequence radius ε can't be infinitely small; it must be for example at least greater than machine epsilon.

NOTE

Machine epsilon

Machine epsilon is the maximum relative error while rounding a floating point number [CS:1, 2]. Simply speaking:

- it is the *largest* floating point number ε such that when added to 1 gives 1: $1 + \varepsilon = 1$;
- it is the *smallest* floating point number ε which when added to 1 gives something different than 1: $1 + \varepsilon > 1$

If for all values of some property you have it may happen that you may receive other values which are "close" (lie inside of a circle, or more general: ball, with radius ε), then this property is continuous. Otherwise it is discrete.

Even if it sound good, it is not so easy to decide in real life cases. For example, the space of car colors of some brand is discrete. You may assign numbers to every color: 0 — black, 1 — red, 2 — green, etc. But you can try to be a wiseacre and assign such a numbers: $0 — 1 + \varepsilon$, $1 — 1 + 2\varepsilon$, $3 — 1 + 3\varepsilon$, etc Than according to above definition this should be classified as continuous space. If you consider RGB color space is this space discrete or continuous?

I think, that every time you decide: discrete or continuous, you must use a common sens. If you blindly apply definitions and formulas you will run into problems. The reference for you should be an analog, real word. If in reality something is continuous, the digitally it makes sens to think about it as continuous even if it is represented with natural numbers like RGB colors.

Factor	Discrete	Continuous
Meaning	Refers to the variable that in real word assumes a finite number of isolated values	Refers to the variable that in real word assumes infinite and uncountable number of different values
Represented by	Isolated points	Any point has another points arbitrarily close to it
Values are obtained by	Counting	Measuring
Classification	Non-overlapping	Overlapping

Accuracy or precise?

Sometimes you can heard about data precision or that some data is accurate. Very often both terms are treated as synonyms but in fact they have different meaning. So what is the difference between precise and accurate?

Accuracy refers to how close a measurement is to the true or accepted value.

Precision refers to how close measurements of the same item are to each other.

Precision is independent of accuracy. That means it is possible to be very precise but not very accurate, and it is also possible to be accurate without being precise.

To illustrate the fundamental difference between both terms, the analogy to a shooting target is instructive.

- Considering the case of a rifle with calibrated sighting scope in the hands of a professional marksman with a steady hand we will get *accuracy* and *precision*.

- Considering the result for a professional marksman using a rifle whose sighting scope is not calibrated we will get *no accuracy* and *precision*.

- Considering the result for an amateur (with a shaky hand) using a calibrated rifle we will get *accuracy* and *no precision*.

- Considering the result for an amateur shooting an un-calibrated rifle we will get *no accuracy* and *no precision*.

Of course, the best quality scientific observations are both accurate and precise.

Data

In this section I want to show different faces of data. I want to show you that data depending on some other "factors" can be treated or interpreted differently. By analogy to computer programming languages or biology I call it *data polymorphism*.

NOTE

Polymorphism

In biology: the occurrence of two or more clearly different *forms* or *morphs*, in the population of a species.

Word *morph* used as a verb means change smoothly from one form to to another by small gradual steps; for example image to another image by small gradual steps using computer animation techniques.

In computer science (programming languages): the use of a single symbol to represent multiple different types.

Data and datasets

Datum is a single value of qualitative or quantitative variable (property). A set of such values is *data*. *Dataset* is a collection or group of **related** data. Each group or dataset member shares the same set of attributes or properties as others in the same dataset.

NOTE

Data

The Latin word *data* is the plural of *datum*, (en. *(thing) given*) neuter past participle of *dare* (en. *to give*). In consequence, *datum* should be used in the singular and *data* for plural, though, in non-specialist, everyday writing, *data* is most commonly used in the singular, as a mass noun (like "information", "sand" or "rain"). Saying the truth, I observe this tendency to be more and more common. The first English use of the word *data* is from the 1640s. Using the word *data* to mean *transmittable and storable computer information* was first done in 1946. The expression *data processing* was first used in 1954 [BD:1].

So *datum* is any **single** value like: `12`, `red`, `high`, `fast`, `19°27'17"E`, `2021-04-04 09:36`, etc. *Single* means: of the form **you treat** as *atomic*, even if it is compound like timestamp or GPS coordinates.

Data is a set of different values like:
`{20210328000007, 281c0489060000bc, 7.687500}`
or
`{Poland, Łódź, 51°46'36"N, 19°27'17"E}`.

You can create a *dataset* of temperatur measurements (in CSV form):

```
timestamp, sensor_id, temperature_celsius
20210328000007, 281c0489060000bc, 7.687500
20210328073157, 281c0489060000bc, 7.062500
```

```
20210328141137, 281c0489060000bc, 13.312500
20210328184357, 281c0489060000bc, 11.312500
```

or places worth to see (as JSON):

```
{
  "data": [
    {
      "country": "Poland",
      "location": "Łódź",
      "coordinates": {
        "lat": "51°46’36”N",
        "long": "19°27’17”E"
      }
    },
    {
      "country": "Peru",
      "location": "Arequipa",
      "coordinates": {
        "lat": "16°25’03”S",
        "long": "71°32’12”W"
      }
    },
    {
      "country": "Austria",
      "location": "Kals am Großglockner",
      "coordinates": {
        "lat": "47°00’N",
        "long": "12°38’E"
      }
    }
  ]
}
```

Difference between data and information

In common understanding and use *data* and *information* are synonyms and very often are used interchangeably. Sometimes I also do it when I have no choice (e.g. to avoid repetition in the text), but I do my best to avoid it.

Data is a *dumb* set of values. Nothing more. When the data is processed and transformed in such a way that it becomes **useful** to the users, it is known as *information*.

Raw facts you collect about an events, ideas, entities or anything else, is called data. In most basic case, data are simple text and numbers. You turn raw facts into information when we process and interpret it. So when data starts to "speak", when something valueless is turned into priceless, we have an information. The same "thing" may be considered as data or information depending on the context it is used.

Factor	Data	Information
Meaning	Data means raw facts gathered about someone or something, which is bare and random	Facts, concerns a particular event or subject, which are refined by processing
Physical form	It is just text and numbers	It is somehow refined
Storage form	Unorganized	Organized
Usefulnes	Who knows? May or may not be	Always

Data polymorphism -- DIKW pyramid

The DIKW (data, information, knowledge, wisdom) pyramid shows that *data*, produced by **events**, can be **enriched with context** to create *information* -- this is what you know. If you go further, information can be supplied with **meaning** to create *knowledge* and knowledge can be **integrated** to form *wisdom*, wich is at the top.

Consider the following example. Number 20 is only a meaningless value. You can't say anything about it, you don't know what you can do with it. And it is exactly what **data** is: simply speaking, only values. If I tell you that it is a value of temperature expressed in degrees centigrade you will know for what it can be used or what you can think about it — you know its usage context. Now the same value is turned into **information**, as you know more than pure number — this information (as now you know how to interpret it) might be useful, provided that you know more. "More" means that you need meaning — you need to know what this temperature is about. It may be a temperature in my fridge, garage or jet engine. So now, the same value, but supplied with context and meaning, is turned into **knowledge**. Knowledge about object, process or anything else it is related to. If it is a temperature in my fridge it means that fridge is broken. If it is a temperature in my garage, it is ok. On the other hand, if this temperature reaches value 60, 200 or more, this may mean that there is a fire in the garage. *Broken fridge* or *fire in garage* is not "encoded" in this value as it is simply a number. So only thanks to **wisdom**, thanks to various knowledge integration, you may infer this. Only thanks to your experience, you know that 200 is really very hot and is not usual for garages. If you registered it, this means that something extremely extraordinary must have happened, for example fire. This knowledge is not coming from pure number but is an effect of combining broader knowledge you have about temperatures, fires, etc. Saying the truth, wisdom is something much more. I would say, that wisdom is a wise, proper and adequate use of knowledge. There is a nice saying (by Miles Kington):

Knowledge is knowing a tomato is a fruit.
Wisdom is not putting it in a fruit salad.

The same value may have different "faces", may represent different type, shape or aspect (chose what is more adequate for you) of data. In other words, you can give your data a value depending on the way you use it. This is how you change the value of your data from *hindsight* through *insight* to *foresight*.

- **Hindsight**: understanding of a situation or event only after it has happened or developed.
- **Insight**: the capacity to gain an accurate and deep intuitive understanding of a person or thing.
- **Foresight**: the ability to predict or the action of predicting what will happen or be needed in the future.

Example:

Hindsight: Understand why the plane crashed.

Insight: Understand what is going on with the plane at the moment and whether it will result in crash soon.

Foresight: Predict that the plane may crash in the future if the engines will be not maintained.

Data analysis and analytics

Data analysis is the process of examining data to find facts, relationships, patterns, insights and/or trends. The overall goal of data analysis is to:

- discover useful information,

- make conclusions,
- support better decision-making.

Data analytics is a term that encompasses data analysis. Data analytics is a discipline that includes the management of the complete data lifecycle, which encompasses collecting, cleansing, inspecting, organizing, storing, transforming, analyzing and modeling data. The term includes the development of analysis methods, scientific techniques and automated tools. Data analytics enable *data-driven decision making* with scientific backing so that decisions can be based on factual data/information and not simply on past experience or intuition alone.

Most wanted properties of data

You can collect, store and process data according to different rules and requirements. Whatever and however you work with data, it is true that:

- you don't want to lose your date;
- you want it to be true now and in the future;
- you want to be able to get back with all changes you made on it.

More technically, you want it to be *raw*, *immutable* and *eternally true*.

Raw form

Everything I do in my professional life I try to be faithful to my own rule: **never ever destroy original data**. If I prepare an image for one of my tutorials I always keep original screenshot even if know I need only a small part of it. When I need to prepare a set of artificial data to be used during my classes I'm saving a source code of the program used to generate it. In most cases operations I do on my data are not reversible. For example when you paint a line on a bitmap and save it you can't revert this action after reloading the image. That is why I always keep original data.

Storing data in the rawest form possible is hugely valuable because maximizes your ability to obtain new insights, whereas any processing like aggregating, overwriting, or deleting limits what your data can tell you.

When you think about rawest, please keep in mind that:

Note 1:

Unstructured data is rawer than normalized data. Recal from physic or mathematic classes vector normalization.

Let v and w are vectors of the form:

$$v = [1,0],\ w = [2,0]$$

Now you can normalize length to not exceed 1. First you calculate length of each vector:

$$|v| = 1,\ |w| = 2$$

Next you can find maximum length:

$$l = \max(|v|, |w|)$$

Having l you can finally normalize your vectors:

$$v_n = \frac{v}{l} = [0.5,0],\ w_n = \frac{w}{l} = [1,0]$$

After vector normalization their relative magnitudes and directions are preserved but this way you loose some information (original absolute magnitudes) and you can't restore it. Having only information that

$$v_n = [0.5,0],\ w_n = [1,0]$$

it is impossible to guess their initial values because there are infinitely many candidate vectors like:

$v_n = [2.5,0],\ w_n = [5,0]$

or

$v_n = [0.05,0],\ w_n = [0.1,0]$

Having only v_n and w_n there is no method to get back v and w.

Note 2:

More informations doesn't necessarily mean rawer data. Sometimes additional information serve only as the container for the contents and shouldn't be part of your raw data.

TODO: example needed

Immutable form

Data immutability means that you don't update or delete data – you only add more. By using an immutable schema for Big Data systems, you make the system more *fault tolerant*. Making your system, and data particularly, resistant to any faults is an essential property. Especially faults generated by humans can be destructive. People make mistakes, and you must limit the impact of such mistakes and have mechanisms for recovering from them.

> People make mistakes, and we must limit the impact of such mistakes and have mechanisms for recovering from them.

With a mutable data model, a mistake can cause data to be lost, because values are actually overridden in the database. With an immutable data model, no data can be lost. If bad data is written, earlier data (unaffected by any mistakes) still exist. Fixing the data system is just a matter of deleting the bad data records (or rather marking as deleted without physical removal) and recomputing what should be fixed. Such an immutable dataset is sometimes called a *master dataset*.

One of the trade-offs of the immutable approach is that it uses more storage than a mutable schema. Rather than storing a current snapshot of the world, as done by the mutable schema, you create a separate record every time a data/information evolves. You track each data field so the entire history of data changes is stored rather than just the current view of the world. In such a case, when multiple instances of every data field exists you have to tie them to a moment in time when the information is known to be true or somehow enumerate them to be able to determine the order of the records.

EXAMPLE 1.1 – THE DATA IS IRRETRIEVABLY CHANGED DURING THE UPDATE

Initial state of the table:

table

id	object_id	data_1	data_2
1	1	1	2
2	2	3	4

State of the table after series of updates:

table

id	object_id	data_1	data_2
4	1	2	2

6	3	3	2
7	2	5	7

As you can se it is impossible to determine how many times object 1 was changed and in consequence it is impossible to recovery bad data if there are any.

EXAMPLE 1.2 – TRACK EVERY DATA CHANGE

You store the whole history of data changes. You save new record not by update but by addition along with timestamp (or counter) and `is_active` marker which informs which record of given object is the newest one.

Initial state of the table:

table

id	object_id	data_1	data_2
1	1	1	2
2	2	3	4

State of the table after series of updates:

table

id	object_id	data_1	data_2	counter	is_acive
1	1	1	2	1	false
2	2	3	4	1	false
3	1	1	3	2	false
4	1	2	2	3	true
5	2	3	3	2	false
6	3	3	2	1	true

7	2	5	7	3	true

If you think that this is a waste of storage space remember you are in Big Data word which is called so for a reason. You have big data not just for fun but to give you new perspectives with various data processing options and paths. Do you need a simple and strongly human-fault tolerant master dataset? No problem, take advantage of the ability to store large amounts of data using Big Data technologies to get the benefits of immutability.

Eternally true form

TODO

Thanks to immutable form, timestamp or counters accompanying your data, each piece of data coming from this set is true in perpetuity.

Different data usefulness

There are four general categories of analytics that are distinguished by the results they produce:

- descriptive,
- diagnostic,
- predictive,
- prescriptive.

The order of categories matter: value and complexity increase from descriptive to prescriptive analytics. Descriptive analytics can be concerned as pure data, while prescriptive as optimization. The different analytics types leverage different techniques and analysis algorithms. This implies that there may be varying data, storage and processing requirements to facilitate the delivery of multiple types of analytic results.

DESCRIPTIVE ANALYTICS

Descriptive analytics are carried out to answer questions about events that have already occurred: ***What has happened?***

This type of analytics **use data aggregation and data mining** to provide insight into the past.

Descriptive analysis or statistics does exactly what the name implies: they *describe*, or *summarize* raw data and make it something that is interpretable by humans. It is analytic that describes the past. The past refers to any point of time that an event has occurred, whether it is one minute ago, or one year ago. Descriptive analytics are useful because they **allow you to learn from past** behaviors, and understand how they might influence future outcomes. For example, descriptive analytics examines historical electricity usage data to help plan power needs and allow electric companies to set optimal prices.

Most of analytics results are descriptive in nature. Descriptive analytics provide the least worth and require a relatively basic skillset. Descriptive analytics are often carried out via ad-hoc reporting or dashboards.

DIAGNOSTIC ANALYTICS

Diagnostic analytics also concern past but answers question: ***Why did it happen?***

Diagnostic analytics aim to **determine the cause** of a fact that occurred in the past using questions that focus on the reasons behind the events. The goal of this type of analytics is to **determine what information is related to something in order to be able to answer questions that seek to determine why it has occurred**.

Diagnostic analytics provide more value than descriptive analytics but also require a more advanced skillset. It usually requires collecting data from multiple sources and storing it in a structure that allows easily perform various analysis. Diagnostic analytics results are often viewed via visualization tools that enable users to identify *trends* and *patterns*.

PREDICTIVE ANALYTICS

Predictive analytics, as descriptive and diagnostic, also concern past but looks into the future and answers question: ***What will happen?***

It use statistical models and various "forecasts" techniques to analyze historical and current facts in case to **understand the possible future and predict it with the highest possible probability**.

With this type of analytics you generate **future predictions based upon past events**. To do this, some models of the past are created. It is very important to understand that these models are very tightly connected with the conditions under which the past events occurred. If these underlying conditions change, your prediction will fail and then the models that make predictions need to be updated. The problem is that you cannot change your models ahead of time — you can do this only after something happens. Even if you know that conditions are changing you have to wait and reconcile with thoughts that your prediction will be incorrect.

This kind of analytics involves the use of large datasets and various data analysis techniques. It provides greater value than both descriptive and diagnostic analytics for the price of a more advanced skillset. The tools used vary and it is very common to use few of them. Various tools and languages, broad theoretical knowledge and unconventional approaches and open mind may be needed to do something in the field of predictive analytics.

PRESCRIPTIVE ANALYTICS

There will be nothing strange if we say that also this type of analytics concern past to think about future but in this case answers question: ***What should I do?***

Prescriptive analytics attempt to prescribing a number of different possible action "sequences" that should be taken to reach a goal. What is

important, **these analytics attempt to evaluate the effect of future decisions and actions before they are actually made**. In this sens, prescriptive analytics is much more than predictive analytics, because **it not only predicts what will happen, but also explains why it will happen**. So the focus is not only on which prescribed option is best to follow, but why.

It use optimization and simulation algorithms to tell us **what I should do to get assumed result**. This is the most demanding analytics require much more than predictive analytics. To be successful in this area you have to be very often an expert in the field of interes. Having knowledge is not enough; you have to "feel" it. For example to answer the question *When is the best time to trade a particular stock?* you have to deeply understand stock market and all its nuances; the best option is we are a trader. This type of job can not be outsourced to "general purposes" company or staff.

SECTION 6

Big data characteristics

Trying characterize big data, you should abstract from any technical specifications, measures and absolute values. What was, in technical terms, big data years ago now is a common and every day task. Over the years, incredible progress has been made in the field of hardware [COM:1]:

By 1961, a few universities around the world had bought IBM 7090 mainframes [designed for large-scale scientific and technological applications]. The 7090 was the first line of all-transistor computers, and it cost US $20 million in today's money [a typical system sold for $2.9 million which is equivalent to $20 million in 2020], or about 6,000 times as much as a top-of-the-line laptop today. Its early buyers typically deployed the computers as a shared resource for an entire campus. Very few users were fortunate enough to get as much as an hour of computer time per week.

The 7090 had a clock cycle of 2.18 microseconds, so the operating frequency was just under 500 kilohertz. But in those days, instructions were not pipelined, so most took more than one cycle to execute. Some integer arithmetic took up to 14 cycles, and a floating-point operation could hog up to 15. So the 7090 is generally estimated to have executed about 100,000 instructions per second. Most modern computer cores can operate at a sustained rate of 3 billion instructions per second, with much faster peak speeds. That is 30,000 times as fast, so a modern chip with four or eight cores is easily 100,000 times as fast.

But, really, this comparison is unfair to today's computers. Your laptop probably has 16 gigabytes of main memory. The 7090 maxed out at 144 kilobytes. To run the same program would require an awful lot of shuffling of data into and out of the 7090—and it would have to be done using magnetic tapes. The best tape drives in those days had maximum data-transfer rates of 60 KB per second. Although 12 tape units could be attached to a single 7090 computer, that rate needed to be shared among them. But such sharing would require that a group of human operators swap tapes on the drives; to read (or write) 16 GB of data this way would take three days. So data transfer, too, was slower by a factor of about 100,000 compared with today's rate.

As you can see, your laptop is way more powerful than you might realize. A week of computing time on a modern laptop would take longer than the age of the universe on the 7090.

A week of computing time on a modern laptop would take longer than the age of the universe on the IBM 7090 mainframe designed in 1959 for large-scale scientific and technological applications.

In consequence you should rather think about concepts and ideas emerging when you work with data to which you cannot apply the current approach and tools due to their high ineffectiveness.

For a dataset to be considered *big data*, it must possess more than one characteristics commonly referred to as the Five Vs:

- volume,
- velocity,
- variety,
- veracity,

- value.

These five big data characteristics are used to help differentiate data categorized as *big* from other forms of data. Three of them were initially identified by Doug Laney in early 2001 when he published an article describing the impact of the *volume*, *velocity* and *variety* of e-commerce data on enterprise data warehouses. To this list, *veracity* and *value* has been added later to emphasize that big data is not only about data but also bout data usefulness.

VOLUME

Almost any aspect of our live is or can be a source of data and they can be generated by human, machines as well as environment. The most common are (H is used to denote human, M — machines, E — environment):

- (H) social media, such as Facebook and Twitter,
- (H) scientific and research experiments, such as physical simulation,
- (H) online transactions, such as point-of-sale and banking,
- (M) sensors, such as GPS sensors, RFIDs, smart meters and telematics,
- (M) any woking device we monitor for our safety, such as planes or cars,
- (E) weather conditions, cosmic radiation.

Today, we collect all data we can get regardless whether we really need them or not. Amount of data is really enormous.

The size seems to be the most obvious factor — hence the name *big* data. We see all the problems through the prism of the size of the data, and undoubtedly this is often the case. But big data is not only about

data itself and size is not so important as more important is to consider also other factors like complete data processing on time. There may be situation that your data is not voluminous but because of the way you process it, you will miss your deadline. It is impossible to give exact size of data above which classify them as big. It doesn't make sense. You can imagine situation when petabytes od data, for example from Large Hadron Collider, is waiting for processing. Yes, it's a lot of data but no one will die if the results are in a month instead of today. In this context, it's not big data but rather time consuming task. Time consuming tasks are common but nobody call them big. You download Linux DVD iso image — it takes time. You run massive system update for your os — it takes time.

Volume is important as far as it occurs in combination with other factors. Because of this I think that *big data* term is a little bit misleading. Personally I use the term *problematic data* which does not emphasize only one of many possible factors.

Velocity

Velocity describes influence of new incoming data on processing pipeline. Even if you have a lot of data but you don't expect any new, things are time consuming but quite simple. Conversely, if chunks of data are tiny but arrive at very high frequency you may have serious problems keeping up with their processing because it may result in accumulation of data that is becoming unprocessable due to the constant flow of new data. There is no time to postpone some processing.

This way volume combined with velocity may get you in troubles turning data into problematic data.

Variety

Data variety refers to the multiple formats and types of data that need to be supported by big data solutions. It also concerns data format

variability — the data coming even from the same source may arrive in different formats. Data variety brings challenges in terms of data transformation, integration and storage. All of this requires additional time reducing this way volume of data possible to processing and limiting the frequency with which the data that can be received arrives.

Veracity

Veracity refers to the quality or fidelity of data. If you want to do something valuable with data it need to be assessed for quality. Because any data can be a part of the signal or noise, it lead to data processing activities to resolve invalid data and remove noise. Noise is data that cannot be converted into information and thus has no value, whereas signals have value and lead to meaningful information. Data with a high signal-to-noise ratio has more veracity than data with a lower ratio. The signal-to-noise ratio of data is dependent upon the source of the data and its type. For example data that is acquired via online customer registrations, usually contains less noise than data acquired via blog postings.

Note that this requirement is not of a technical nature, but rather we can talk about a functional requirement, about *usability*. To name a set of data a big data set is not enough anymore for it to be voluminous with data incoming in various formats and high frequency. If it's everything, then your data is not big data because it doesn't offer you a special features. If you cannot rely on your data then you have garbage, no data. So what if there is a lot of data, since they are not suitable for anything anyway.

Value

Another one nontechnical requirement. Value is defined as the *usefulness of data*. In most cases it is considered in relation to problem — in many cases to the data processing time. A 20 minute delayed stock quote has no value for making a trade compared while you can wait several months to get results of a complex physical simulation.

Simplifying the issue, you can say that, *value and time are inversely related*. The longer it takes for data to be turned into meaningful information, the less value it has for a business. Outdated information, even if of very good quality and high probability, is totally useless.

The goal of big data usage is to conduct analysis of the data in such a manner that high-quality results are delivered in a timely manner, providing optimal *value* to the company.

More "V"s?

In this chapter I have described five Vs. It's worth to note, that initially (2001) there was only three Vs: *volume*, *velocity*, *variety* extended later by *veracity* and next by *value*. Along with how the area and method of using big data changes, the characteristics of what we consider as big data (what big data should be) also change. According to some sources [BD:5] today big data features may be summed up in 10 different Vs like: volume, velocity, variety, veracity, value, validity, volatility, variability, viscosity and vulnerability. Some authors go even further and try to distinguish more Vs: 17, 51 or even 56 [BD:6, 7, 8]. Does it make sense? Not really. Remember that the factors characterizing big data have been conceived in such a way that they define this type of data only if they occur all together. While it is possible for 7 or 10 factors, 56 seems to be a marketing procedure. Below I am going to describe few of them which I found to be the most sensible or interesting. Do not get attached to specific names but rather concepts behind them.

Volatility

Volatility means for how long data is useful to the user. This concept applies to both data obtained a moment ago or just now, requiring immediate processing, and data from the past, which over time lose their value and eventually become completely useless.

Viscosity

Viscosity characterise a time difference the event occurred and the event being described.

Vulnerability

Every organizations have a big responsibility to protect every data, especially personal data, and be more transparent about its usage. This can be addressed by adding *vulnerability* as another essential consideration, with regards to every piece of data which is collected. Challenges need to be faced while addressing people's concerns about their personal data – particularly when it comes to medical or financial information.

Processing large amount of data you have a chance to discover things you would never know working with small data sets. This is no wonder, as this is the essence of using big data, isn't it? Most of the time, you expect it, but sometimes you can get in trouble or cause trouble for someone else.

TODO search for examples

VISUALIZATION

It's good if results you obtain are visualisable in a meaningful way. Any data, especially big which are quite difficult to "summarize" all at once, when visualized helps data scientists or analysts understand it better.

So visualisation is another one functional requirement increasing usability of your data.

SECTION 7

Summary

Big data characteristic:

- **Volume** is important as far as it occurs in combination with other factors.
- **Velocity** describes influence of new incoming data on processing pipeline.
- **Variety** results in increased processing times reducing this way volume of data possible to processing and limiting the frequency with which the data that can be received arrives.
- **Veracity** results in reliable data; otherwise you have garbage, no data.
- **Value** is about turning raw data into meaningful information no later than it is required.

RULE

Factors characterizing big data should be considered all together.

CHAPTER 4

Big Data paradigms

What you will **learn**:

- Why big data requires its own engineering

Paradigm

Best practices? No, thank you

Word *paradigm* comes from Greek *paradeigma*, "*pattern, example, sample*" from the verb *paradeiknumi*, "*exhibit, represent, expose*" and that from *para*, "*beside, beyond*" and *deiknumi*, "*to show, to point out*".

The Oxford English Dictionary defines the *paradigm* term (pronounced /ˈpærədaɪm/) as "*a typical example or pattern of something; a pattern or model*"[1]. The term was introduce to contemporary vocabulary by the historian of science Thomas Kuhn when he adopted the word to refer to the *set of concepts and practices that define a scientific discipline* at any particular period of time. In his book, *The Structure of Scientific Revolutions* published in 1962, he defines a scientific paradigm as: "*universally recognized scientific achievements that, for a time,* ***provide model problems and solutions for a community of practitioners***".

In contemporary, common, science and philosophy meaning, *a paradigm is a distinct set of concepts or thought patterns, including theories, research methods, postulates, and standards for what constitutes legitimate contributions to a field.*

[1] Oxford, paradigm, retrieved 2022.04.02, `https://en.oxforddictionaries.com/definition/paradigm`

A paradigm from the *dogma* (see further) is distinguished by several essential features:

- It is not given once and for all. It is adopted on the basis of the consensus of most researchers, which does not imply that the scientists vote on accepting or rejecting the paradigm. What counts is the paradigm's correspondence to existing knowledge and the fulfilment of many conditions (in terms of, for example, existing proofs).
- It may periodically undergo fundamental changes leading to profound changes in science what is called the scientific revolution.
- There is no something such *absolute correctness*.

A good paradigm has several essential features:

- It is consistent logically and conceptually.
- It is as simple as possible and contain only those concepts and theories that are really necessary for a given science.
- It give the possibility of creating detailed theories in accordance with known facts.

NOTE

Dogma

The term *dogma* is transliterated in the 17th century from Latin *dogma* meaning "*philosophical tenet*", derived from the Greek *dogma*

meaning literally "*that which one thinks is true*" and Greek *dokeo* "*to seem good*".

It is used in pejorative and non-pejorative senses.

- In the non-pejorative sense, dogma is an official system of principles or tenets of a church, or the positions of a philosopher or of a philosophical school.

- In the pejorative sense, dogma refers to enforced decisions, such as those of aggressive political interests or authorities. More generally it is applied to some strong belief that the ones adhering to it are not willing to rationally discuss. It is often used to refer to matters related to religion, but is not limited to theistic attitudes alone, and is often used with respect to political or philosophical ideas.

In this chapter, I intentionally talk about *paradigms* instead of *best practices*. Why not *best practices*? – you may ask. Best practices? Because according to Toyota, and I agree with that, there is no such thing as *best practice*. When you say *best practices*, you behave like you have just found a Holy Grail and, being satisfied, you will never search again for any other improvements. You stuck in what you have. Although the key principles of Toyota's Production System can be conveniently summarised, it is vital to remember *it represents a* ***way of thinking****, not just a set of tools and techniques*. For those of you looking for a quick fix you may as well stop reading now. There isn't one. It's taken Toyota 60 years to create what they have and they still haven't finished.

NOTE

The Toyota Production System [XXX:5]

1. Adopt A Long Term Philosophy: It's Not Just About Making A Profit

- Until most recently, Toyota was different from most companies. Decisions are not driven entirely by satisfying short term profit and loss forecasts. They actually think beyond the desire to satisfy the accountants. Toyota plan and act with the long term in mind, not the quarterly results.

- Toyota's mission and guiding principles is not a poster on the wall, they're practiced each and every day.

2. Investment In People: "We Build People Before We Build Cars"

- Every company says its products are only as good as its people. Toyota actually means it. They invest considerable effort in recruiting and keeping the very best. The recruitment of an assembly line worker can take up to 14 months before they are offered a full time contract. When asked: *How can you afford this kind of recruitment process*, Toyota's response is swift *How can YOU afford not to?*

- Toyota reward team performance, not that of the individual. Promotion can take time. Slow promotion and rewards for team work is the norm.

- Toyota sees ongoing education as vital. They have even built their own University to ensure a steady flow of high quality engineers! **They regard education as an investment, not a cost.**

- **Managers are seen as mentors and coaches, teachers not dictators.**

- **Toyota generally promote from within. They grow leaders rather than purchasing them.**

3. Focus On The Production Line: Keep The Main Thing The Main Thing

- Everyone focuses on servicing the production line. Every other activity is seen as *non value added.*

- An *upside down* management structure. Team leaders and managers support those at the sharp end, not the other way around. **Assembly line operators are at the top of the pyramid, not the bottom.**

- **There is a genuine belief that only assembly operators and engineers add value. Everyone else must justify their reason for being!**

- Everyone must serve their apprenticeship and develop deep understanding of the process. Even those in HR have "time served" on the assembly line.

- **Managers and team leaders spend up to 80% of their time on the production line solving problems and adding value -– not in meetings or "emailing".**

- **Leaders manage from where the work is done, not from their office.**

4. A Genuine Learning Organisation: "Solving Problems Is Key To Our Success"

- Toyota intentionally runs very, very lean. There are no buffer stocks to fall back on. Running lean means that problems can have a dramatic impact on production output. **Problems have to be fixed, and fixed quickly.**

- The pervading attitude is that **problems and process deviations are really good news, providing you learn from them. It's accepted that "real time" solving problems at source saves time and money later on downstream.**

- Toyota famously adopts the *andon approach (which is a manufacturing term referring to a system to notify management, maintenance, and other workers of a quality or process problem; the alert can be activated by button, by the production equipment itself or even manually by a worker using a pullcord)* to problem solving. Deviations are acted upon within minutes and always solved at the site of the incident, never from behind a desk. Deviations are brought to the surface quickly and solved within hours.

- Suggestion schemes are used to generate new ideas and ways of working with over 90% of suggestions implemented. **Payments are weighted towards the small incremental improvements not the big ones.**

5. Only Focus On The Manufacturing Process And Nothing Else

- They rarely use any new or unproven technology.

- **Technology is only used when it can add value and keep things simple.**

- Toyota's attitude is that **people do the work, computers only move the information**. If technology distracts or confuses the user, it is simply not used.

6. Standardisation Is The Name Of The Game

- **Standardisation is about finding out the scientifically best way of doing a task, proving that it works and then "freezing" it.** Although people are expected to "follow the rules" SOPs (SOP — Standard Operating Procedure) are not allowed to stifle innovation and further improvement. Users are encouraged to share best practice, "hints and tips" and improve SOPs further. SOPs are constantly reviewed and improved, not every 2-3 years.

- The level of procedural compliance is very high for one simple reason — user involvement. SOPs are developed from the bottom up not from the top down. Management have very little input. Users are seen as the document owner. They write, design, refine and implement all new SOPs. Not surprisingly compliance is not a problem because they usually work.

7. The War On Waste: It Never Stops

- Anything that adds no value is removed from the system.

8. Performance Measures: Less Is More!

- **Less is more. Toyota measures only what is important and avoids the "death by measure".**

- They only select measures that drive the right behaviour. For example, assembly line workers are rewarded for raising deviations.

> After all, **you learn more from your mistakes than from your successes**. Contrast this approach with that taken by many [...] companies, namely to encourage people to reduce deviation numbers. This measure drives completely the wrong behaviour!

Notice that Toyota's principles are applicable to many different and distant business branches. They have proven themselves in the large and difficult business which is the production of cars. Due to the significant competition, it requires considerable creativity and innovation both in solving current problems and introducing new solutions. And yet this is exactly what working with large datasets also requires.

See the summary below. Think which of the listed elements does not match big data. Throw it away and use the rest.

1. Always look far ahead in what you do. Short-term gains are a source of great losses in the future.

2. Invest in education. Managers are mentors and coaches, teachers not dictators. Promote team work, reward team performance, not that of the individual.

3. Managers, team leaders, managers of managers, etc must justify their reason for being. Everyone must serve their apprenticeship and develop deep understanding of all data processing steps. They have to understand technology both in theory and in practice. Write code, instal and configure software — without it you will not understand what you are doing.

4. When you work with big data stream, you have to fix problems quickly to prevent their accumulation and to protect from their negative impact on business processes.

5. Do not blindly follow the current trends in technology. Technology is only used when it can add value and keep things simple.

6. Never say: *I will not do it, it is against best practice.*

7. Listen other people, listen users and involve them. Allow them to write, design and refine every step of big data processing.

8. Don't be ashamed to make mistakes. Mistakes are there to be learned from.

In the rest of this chapter, you'll see some factors to consider when working with large data sets.

Factors you should consider

In this section you'll read about some factors I suggest to consider when working with big data sets. Whether you agree or not you will have to face with them, they are real things and you can not get away from them. They influence the ways in which you can/should think about big data systems. You may adopt them if you think they are applicable to your case. You may like them or reject. But you can't deny their existence, so it's better to know them.

CPU limits

All we ever heard about the Moore's Law — an empirical law, resulting from the observation that the economically optimal number of transistors in an integrated circuit increases in subsequent years in accordance with the exponential trend (it doubles in almost equal lengths of time). The authorship of this law is attributed to Gordon Moore, one of the founders of Intel, who in 1965 described a doubling every year in the number of components per integrated circuit, and projected this rate of growth would continue for at least another decade. In 1975, looking forward to the next decade, he revised the forecast to doubling every two years. The period is often quoted as 18 months because of Intel executive David House, who predicted that chip performance would double every 18 months (being a combination of the effect of more transistors and the transistors being faster).

One of the main reasons why this exponential growth is possible is the use of smaller and smaller elements in the fabrication process. Today, 65, 45, 32 and recently 14 nm dominate, when 500 nm technology was used in the early 1990s. Taking into account classical physics, these dimensions can not be reduced indefinitely – the limit is the size of atoms, and the next limitation is the speed of light in a vacuum that sets the upper limit for the speed of information transmission. In short: computers can't be much faster than they are today.

On the other hand you run at the same time more and more applications. Even if you don't know, operating systems executes many services to make your work smoother, more natural and enjoyable. All this happens without interrupting your work. How it is possible? We learned to effectively use concurrency. Thanks to fast context switching system assigns its resources for a small time interval to every application — executing a bit of each application — giving the impression of their simultaneous work. Later a multi-core processors were introduce increasing the ability to run concurrently many applications. This occurred to be a move in a right direction. Nowadays the trend of the entire computer industry is towards the creation of multi-core systems and parallel processing (used so far only in efficient servers and supercomputers).

To visualise this, I made a comparison. I selected an old dual-core Intel Xeon serwer processor. To be more precise, I decided to average benchmark results of processors from this family. According to Wikipedia [COM: 2]: *The 3000 series, codenamed Conroe (product code 80557) dual-core Xeon (branded) CPU, released at the end of September 2006,* I took from [COM: 3] two marks:

Intel Xeon 2006 (2C/2T, DTP 65W)	CPU Mark	Thread Mark
3040 (1.86GHz)	789	474
3050 (2.13GHz)	841	762
3060 (2.33GHz)	822	735
3070 (2.66GHz)	1025	1024
3075 (2.66GHz)	1096	1096
3085 (3.00GHz)	1070	1124
Average	940	879

You can compare these values with averaged results for the best CPU available today (February 2021):

Top scored desktop AMD:

Model	CPU Mark	Thread Mark
AMD Ryzen Threadripper PRO 3995WX 2.7GHz (64C/128T, 280W DTP)	88731	2665

Top scored server AMD:

Model	CPU Mark	Thread Mark
AMD EPYC 7702 2.0GHz (64C/128T, 200W DTP)	71929	2099

Averaging both results you obtain 80330 for CPU Mark (which is 85.5 times better than Xeon 2006) and 2382 for Thread Mark (which is 2.7 times better than Xeon 2006).

Top scored desktop Intel:

Model	CPU Mark	Thread Mark
Intel Core-i9-10980XE @ 3.00 (18C/36T, 165W DTP)	34290	2653

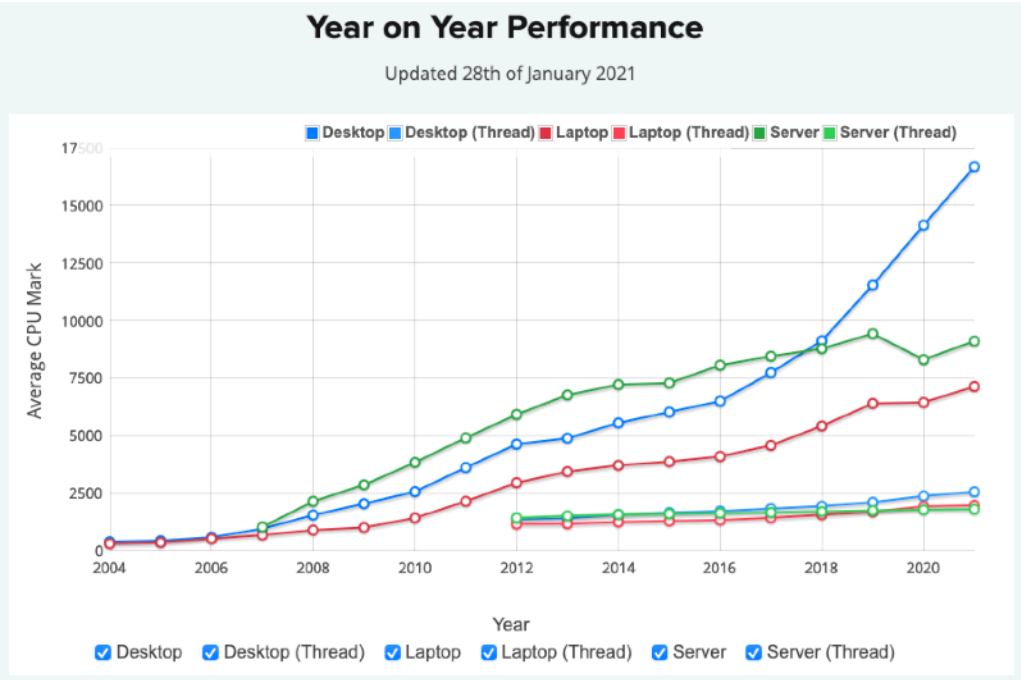
Top scored server Intel:

Model	CPU Mark	Thread Mark
Intel Xeon W-3275M @ 2.50GHz (28C/56T, 205W DTP)	39478	2695
Intel Xeon Gold 6248R @ 3.00GHz (24C/48T, 205W DTP)	38521	2270

Averaging all three Intel results you obtain 37429 for CPU Mark (which is 39.8 times better than Xeon 2006) and 2539 for Thread Mark (which is 2.9 times better than Xeon 2006).

It's worth to note two things:

- Desktop solutions (at least in case of AMD) outperforms server solutions [COM: 4].
- An average CPU mark per thread is not changing much through last few years (from 2012 till now, 2021). So why modern CPU are so fast? How they get so high Average CPU Mark result? The answer is: they use multiple cores, so single core result is multiplied by the number of cores. Today the highest number of cores among server CPUs is 28 for Intel and 64 for AMD which is doubled to 56 and 128 respectively thanks to Hyper-Threading Technology.



Multi-core approach is limited by few factors:

- The more program execute the more data is needed and this can cause high delays in access to data.
- Big problem of current technologies is high power consumption and heat generated. The latter, in particular, is very problematic as we are slowly losing the ability to effectively take away the heat emitted by the system.
- To effectively utilize such system, an application must be designed and written with paralel execution in mind. In this case concurrency is limited according to Amdahl's law

NOTE

Is it worth to upgrade your CPU? [COM: 5]

Now that Intel's latest 9th-gen Core mobile chips are on their way, it's time to figure out whether it'll be worth it to pay top dollar for the new chip; buy or keep a laptop with an 8th-gen CPU (plenty of current models remain available), or upgrade from your 6th-gen or 7th-gen model.

The issues differ from generation to generation. ***In many cases, we've said you can wait a few years before upgrading.*** *The 8th-generation jump was an exception: It represented one of the biggest laptop CPU improvements in a long time, worthy of upgrading even from a 7th-gen chip.*

With the 9th gen, however, we're mostly back to the incremental clock speed upgrades we've seen from Intel for years. *That's not to say it's all underwhelming — because you can get that fancy new 802.11ax/Wi-Fi 6, too. But* ***for those interested in speed boosts, the sparse improvements in core counts mean you can largely ignore this series except for the Core i9 lineup.***

As you can see, the only true reason that one can be interested in upgrading system is rather the number of cores than clock speed upgrade.

AMDAHL'S LAW

In computer architecture, Amdahl's law is a formula which gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved. It is named after computer scientist Gene Amdahl, and was presented at the AFIPS Spring Joint Computer Conference in 1967.

Amdahl's law is often used in parallel computing to predict the theoretical speedup when using multiple processors. For example,

- If a program needs 100 hours using a single processor core, and a particular part of the program which takes 50 hour to execute cannot be parallelized, while the remaining 50 hours ($p = 50/100 = 0.5$) of execution time can be parallelized, then regardless of how many processors are devoted to a parallelized execution of this program, the minimum execution time cannot be less than that critical 50 hours. Hence, the theoretical speedup is limited to at most 2 times ($1/(1 - p) = 1/(1 - 0.5) = 2$).

- If a program needs 100 hours using a single processor core, and a particular part of the program which takes 5 hour to execute cannot be parallelized, while the remaining 95 hours ($p = 95/100 = 0.95$) of execution time can be parallelized, then regardless of how many processors are devoted to a parallelized execution of this program, the minimum execution time cannot be less than that critical 5 hours. Hence, the theoretical speedup is limited to at most 20 times ($1/(1 - 0.95) = 20$).

According to Amdahl's law the theoretical speedup of the execution of the whole task increases with the improvement of the resources of the system and that regardless of the magnitude of the improvement, the theoretical speedup is always limited by the part of the task that cannot benefit from the improvement. **Any theoretical speedup is limited by the serial part of the program. For this reason, parallel**

computing with many processors is useful only for highly parallelizable programs.

RULE

Think parallel.

TODO update all this section from this point

Software availability

Nowadays word are driven by free, open sourced and easy to make first step software (get it, install it and run it) and we have a lot of options to choose from. In consequence, we can easily mix different technologies and test them before we will use them in real production system. This give us a great flexibility in terms of tools we use as far as a high adaptability to follow changing market requirements.

be able to support and interoperate with a wide range of applications/ systems

be easy extensible

RULE

Be ready to mix technologies, be easily extensible.

Need to be easy adaptable to follow changing market requirements

In today's world...

be easy adaptable to follow changing market requirements

RULE

Be flexible, be agile. Be easy adaptable to follow changing market requirements.

Fault tolerancy

be fault tolerant

RULE

Be fault tolerant.

Use what you have

We can say that there are two options: either we can use specialized, carefully tuned one-node system or we can use easy available commodity nodes (components). Whether it pays off, it depends on what is important to us. Let's point out some aspects.

- Even specialized, carefully tuned one-node system can not be used forever and extended infinitely. There will come a time when we will have to think not about modifying but about replacing the system.
- What if you spend a lot of money and time to tune your one-node system if one day, according to agile approach and some business requirement, you will have to change your concept and make 180 degree turn in system design.
- What is cheaper to buy: one one-node monolithic system or ten commodity nodes?
- What is cheaper in everyday running: one one-node monolithic system or ten commodity nodes?
- What is easier to maintain: one one-node monolithic system or ten commodity nodes?
- What is easier to use: one one-node monolithic system or ten commodity nodes? DB problem - distributed queries.
- What is easier to program: one one-node monolithic system or ten commodity nodes?
- What is easier to extend: one one-node monolithic system or ten commodity nodes?

RULE

High-tech may not pay off.

High-tech may not pay off

Compare CPU case

RULE

High-tech may not pay off.

Hardware rental

Everybody heard about cloud technology. We can say that clouds are everywhere: on the sky which is obvious, and in our life regardless we realize about that or not. Many services are cloud-based and this seems to be an irreversible trend. Despite cloud is trendy, it offers something else: great elasticity. Today it may not pay to own private hardware in private location. It may be much easier and cheaper to rent hardware on demand which is known as *Infrastructure as a Service* (IaaS). With this you can increase or decrease the size of your infrastructure nearly instantaneously with just one-button click, depending on the job you are going to do. So today horizontal scaling, i.e. adding or removing nodes (computers) to/from a system, is not a problem and can be completed on demand within just a few seconds.

On the other hand... cost etc.

RULE

Rent or own? Think carefully.

SECTION 3

Summary

Desired properties of a Big Data system

In the proceedings sections some trends and problems were described. All of them should be taken into account if we want to "define" any patterns or models. Have in mind, that no pattern or model is ethernal and given once for all and should be verified and re-adopted on the basis of the state of the art knowledge, theory and practice.

A Big Data system should: TODO check, update....

- be scalable;
- be distributed (the databases and computation systems you use for Big Data should be aware of their distributed nature);
- be able to support and interoperate with a wide range of applications/ systems;
- be ready to mix technologies, be easily extensible;
- be flexible, be agile; be easy adaptable to follow changing market requirements;
- be fault tolerant;
- remember that high-tech may not pay off;

- think about rent or own.

CHAPTER 5

Storage concepts for Big Data

What you will **learn**:

- Why big data requires its own engineering

Different data formats

Data can come from a variety of sources and be represented in various formats or types. The primary formats of data are:

- *structured data*,
- *unstructured data*,
- *semi-structured data*,
- *metadata*.

TODO add more text to every subsection. Add examples

Structured data

Structured data **conforms to a data model or schema** and is often stored in tabular form. It is used to describe relationships between different entities and in consequence most often stored in a relational database.

Examples: invoices and receipts, sensor data, online forms, spreadsheets.

Unstructured data

Data that **does not conform to a data model or data schema** at all. Unstructured data cannot be stored in predictably ordered columns and rows. One type of unstructured data is typically stored in a BLOB (binary large object), a catch-all data type available in most relational database management systems. Unstructured data may also refer to irregularly or randomly repeated column patterns that vary from row to row or files of natural language that do not have detailed metadata.

It is estimated that unstructured data makes up to 80% of the data within any given enterprise [BDC:2]. Unstructured data has a faster growth rate than structured data.

Examples: social media content, emails, text document.

Semi-structured data

Semi-structured data **has a defined level of structure and consistency, but is not relational in nature**. Instead, semi-structured data is hierarchical or graph-based. This kind of data is commonly stored in files that contain text. For instance, XML and JSON files are common forms of semi-structured data. Due to the textual nature of this data and its conformance to some level of structure, it is more easily processed than unstructured data.

Metadata

Metadata is **data which are not data themself but provides information about a dataset's characteristics and structure**. The tracking of metadata is crucial to Big Data processing, storage and analysis because it provides information about the data birth as well as all processing steps. In many cases metadata helps to process data. For example you may keep some metadata about the image resolution and number of colors used. Of course you may get this data from graphic file, but for the price of longer processing time.

Databases

Database

TODO: intro why database, why we need tem etc... because organize wa of data storage

Essentially, a database is an organized collection of data. Databases are classified by the way they store this data. Early databases were flat and limited to simple rows and columns. Today, the popular databases are:

• relational databases (SQL or RDBMS databases), which store their data in tables;

- no-relational databases (NoSQL databases), which store their data in various formats defined by practical requirements.

SQL DATABASES

I assume you have basic knowledge and understanding of databases related concepts and ideas as well as basic skills to work with them. That is why bellow I present only a short summary of SQL databases.

- Relational databases are good for handling **transactional workloads** involving **small amounts of data with random read/write properties**. They are ACID-compliant, and, to honor this compliance, they are generally **restricted to a single node**. For this reason, they do not provide out-of-the-box redundancy and fault

tolerance.

To handle large volumes of data arriving at a fast pace, databases need to scale. Relational databases employ **vertical scaling**, not horizontal scaling, which is a more costly and disruptive scaling strategy. This makes them less than ideal for long-term storage of data that accumulates over time.

- Relational databases generally **require data to adhere to a schema**. As a result, storage of semi-structured and unstructured data whose schemas are non-relational is not directly supported.

  It is true you can store XML or JSON with the help of built in queries and functions but it seems to be against relational theory. Saying the truth, you can store anything you want simply as a BLOB object which does not mean that the relational databases are suitable for this type of data.

  Furthermore, with a relational database **schema conformance is validated** at the time of data insert or update by checking the data against the constraints of the schema. This **introduces overhead that creates latency**.

- After mentioned latency makes relational databases a less than ideal choice for storing high velocity data that needs a highly available database storage device with fast data write capability.

As a result, a traditional relational systems are generally not useful as the primary storage device in a big data solution environment.

NoSQL databases

Not-only SQL (NoSQL) refers to technologies used to develop next generation databases — non-relational databases.

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

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

- **High availability**
  Often are built on cluster-based technologies that provide fault tolerance out of the box.

- **Oriented on aggregations**
  Unlike relational databases that are most effective with fully normalized data, NoSQL storage devices store data in denormalized, highly aggregated form, which is a merged, often repeatedly nested, data for an object. This eliminates the need for joins and extensive mapping between application objects and the data stored in the database.

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

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

- **Eventual consistency**
  Data reads across multiple nodes but may not be consistent

immediately after a write. However, all nodes will eventually be in a consistent state.

- **BASE, not ACID**
  BASE compliance requires a database to maintain high availability in the event of network/node failure, while not requiring the database to be in a consistent state whenever an update occurs. The database can be in a soft/inconsistent state until it eventually attains consistency. As a result, in consideration of the CAP theorem, NoSQL storage devices are generally AP (availability and partition tolerance) or CP (consistency and partition tolerance). Third case, the CA (consistency and availability) is not considered in NoSQL as partition tolerance is a fundamental requirements imposed by distributive nature such storage systems.

- **Lack of standard SQL**
  Data access is generally supported via API based queries, including RESTful APIs, whereas some implementations may also provide SQL-like query capability.

- **Lower operational costs**
  Many NoSQL databases are built on open source platforms with no licensing costs. They can often be deployed on commodity hardware.

As a result, they are not appropriate for use when implementing large scale transactional systems.

NewSQL databases

To me NewSQL databases appear to be a buzzword and marketing term for describing systems we want to have but impossible to get.

NewSQL systems, as they are described, combine the ACID properties of RDBMS with the availability, scalability and fault tolerance offered by NoSQL storage devices. Additionally they support SQL compliant

syntax for data definition and data manipulation operations, and they often use a logical relational data model for data storage.

As a comment to this idea, I remind you that NoSQL appears because it has not been possible to meet these specified requirements at the same time.

SECTION 3

Ways you work with data

Depending on various points of view, needs and possibilities there are different ways you may work with your data. Now I want to discuss them taking into account storage factors.

Pure data

The most basic and primitive is working with *pure data*. In context of storage, pure data means that you have no intermediate layer separating you from data. Simply speaking, you have raw data, saved in the very raw form in the simples format you can imagine – as a flat text or binary file. Anything you want to get requires a lot of commitment on your part. You have to know where data you are looking for are located inside the file, how many bytes or characters you have to read and which decoding you should apply. If you know this, next you have to write on your own custom code. If you are lucky, you may use some libraries like *pandas* or *NumPy*. Otherwise, you have to write everything from scratch. This approach gives you, probably, the highest control on your data, what and how you process. At the same time, it is also the most time consuming approach. It requires extensive knowledge, various skills and patience, and various mistakes happen very often. You have to be very cautious and responsible because in many cases you work with somehow collected pure data that cannot be derived from anything else than original data source which may not exist anymore. Destroying data can lead to irreversible disaster, because there may be no way to recovery what you have collected.

Queries

Queries are questions you ask of your data. Queries changes your perspective. Now you can focus on digging or discovering information "hidden" in (pure) data instead of struggling with their physical layout and organization. With queries you say what you want to get with significantly reduced need for knowledge about underlying data organization.

Saying *what* instead of *how* while searching you are free of all low level details. This makes queries and all other accompanying processes much more easier and understandable even for nontechnical people compared to pure data. This is really important, because in most cases data, in order to extract informations and knowledge from them, are processed by people who are not IT specialists by education. Apart from that, for example SQL queries are just much more readable than code written in C or Python. This helps to keep your code and intention clear and easy to maintain.

Views

View is an abstraction you put over your various sources of data. It shows what you have in the form which is much more convenient to further processing. In most cases view rather presents what is derived from data or information you already have than create or acquire new items. To make it simple, they are designed to help query data and answer questions about information "hidden" in pure data.

Views are very common in, but not limited to, databases where it combines, often with the help of joins, data from two or more tables. *Combine* means not "glue" together to make huge records but rather to select what you really need or present it in the form which simplifies

next processing steps. This makes them convenient to "hide" (or abstract) complicated queries. View can but often doesn't store real data data and is computed in real time on demand. Because you can query view like you can query simple table, some refer to a views as *virtual tables*.

Views provide several advantages over tables:

- **Simplicity** Views can join multiple tables into a single virtual table. This way views provide a simplified and flattened "image" of the database perfectly suitable for reporting or ad-hoc queries.

- **Preprocess data** With views you can present data aggregated from multiple tables as it is a natural part of the data. You can also present data in convenient way, for example change its format or type.

- **Apply business rules** You can use view to define various business rules to present data in one unified way. For example, you may create a view to present the most profitable product from your shop, where the term *profitable* depends on the adopted business strategy.

- **Security** View allows you to control *what*, *when* and *how* is presented or visible for queries you make on view. You can grant no permissions on the pure data table and the same time create view that limits column or row access and grant permissions to users to see the view (which in result allows them to see some but not all data). This way view can restrict direct access to a table, yet allow users to access only what is intended for them. Depending on permissions you may have an access to all records, limited numer or none of them. For example with view you can simply filter only last inserted or updated record related to given object, keeping the entire history of all changes hidden.

- **Space** View take up very little space, as it defines only how to present data but not store real data (unless you use materialized view which is

a database object that contains the results of a query). Based on some tables you may create many different views. All of them will present "new" data (old data in a new way) based on the same set of pure data which is stored only once in its source table.

Fact based data model

There are many ways you could choose to represent master dataset. You have a choice between traditional relational tables, structured XML, semistructured JSON documents or any other possibilities you only know. It's up to your needs and knowledge. All of them tells you nothing about how to store your data; they are about physical method. In this part I want to describe *logical* data model known as *fact-based model*.

To understand the idea behind this method, consider the following initial state of the data "packed" into relational table:

table

id	data_1	data_2	data_3
1	a1	b1	c1
2	a2	b2	c2
3	a3	b3	c3

Now imagine you change `c2` to `c22`. The worst solution is when you simply update it:

table

id	data_1	data_2	data_3
1	a1	b1	c1
2	a2	b2	**c22**

3	a3	b3	c3

This way you irrevocably lost the history of your data. The consequence is the inability to recover the previous values in case the new value turns out to be incorrect.

Much better is when you use timestamp, as it was explained in *Example 1.2 – track every data change* of *Chapter 3 Big Data concepts and terminology*, *Section 4: Most wanted properties of data*, *Subsection: Immutable form*:

table

id	data_1	data_2	data_3	time
1	a1	b1	c1	t1
2	a2	b2	c2	t2
3	a3	b3	c3	t3
4	a2	b2	**c22**	t4

Now you keep the whole history. The drawback of this approach is high level of redundancy: record with `id=2` and `id=4` has the same data in column `data_1` and `data_2`.

Another problem you may face with is nonexisting data. If nonexisting why this is a problem? Because even data which does not exist occupies some storage space. The wider records you have, the more empty cells you may find:

table

id	data_1	data_2	data_3	time
1	a1	b1	c1	t1
2	a2	b2	c2	t2

3	a3	b3	c3	t3
4	a2	b2	c22	t4
5	a4		c4	t5

Fact-based model allows to solve after-mentioned problems. In this approach you store only what has been changed and what is really needed; you record only relevant facts. In this approach every data is stored in separate table:

table for data_1

id	oid	d_1	t
1	1	a1	t1
2	2	a2	t2
3	3	a3	t3

table for data_2

id	oid	d_2	t
1	1	b1	t1
2	2	b2	t2
3	3	b3	t3

table for data_3

id	oid	d_3	t
1	1	c1	t1
2	2	c2	t2
3	3	c3	t3

Physical data scattering does not have to be a problem because it can be masked, hidden from the user, with the help of views which will present data from many tables in one. User may have no knowledge about their underlying structure.

When you change `c2` to `c22`, you have to perform only a "small" insert to one table:

table for data_1

id	oid	d_1	t
1	1	a1	t1
2	2	a2	t2
3	3	a3	t3

table for data_2

id	oid	d_2	t
1	1	b1	t1
2	2	b2	t2
3	3	b3	t3

table for data_3

id	oid	d_3	t
1	1	c1	t1
2	2	c2	t2
3	3	c3	t3
4	2	**c22**	t4

Fact-based model also gracefully handles nonexisting data:

table for data_1

id	oid	d_1	t
1	1	a1	t1
2	2	a2	t2
3	3	a3	t3
4	4	a4	t5

table for data_2

id	oid	d_2	t
1	1	b1	t1
2	2	b2	t2
3	3	b3	t3

table for data_3

id	oid	d_3	t
1	1	c1	t1
2	2	c2	t2
3	3	c3	t3
4	2	c22	t4
5	4	c4	t5

As you can see, if data doesn't exist there is no need to store any information about this at all.

SUMMARY – BENEFITS OF THE FACT-BASED MODEL

A fact-based model is an ever-growing list of immutable, timestamped atomic facts. For sure this isn't a pattern that relational databases were built to support as it requires a tones of joins, but who said you had to use a relational databases at all. This model provides a simple but powerful representation of your data by naturally keeping a full history of each entity over time. Its append-only and high level of distributivity nature supports data partitioning, makes it easy to implement in a distributed systems.

SECTION 5

Data warehouse, lake, mart...

Data warehouse

A data warehouse is *central repository of integrated data from one or more disparate sources*. It store current and historical data in one single place that are used for reporting and data analysis and is considered a core component of *business intelligence*.

NOTE

Business intelligence

Business intelligence is a set of methodologies, processes, architectures, and technologies that transform raw data into meaningful and useful information used to enable more effective strategic, tactical, and operational insights and decision-making. It combine:

- data gathering,
- data storage,
- knowledge management,

- analysis

to evaluate complex information for presentation to planners and decision makers, with the objective of improving the timeliness and the quality of the input to the decision process.

A typical data warehouse often includes the following:

- Database (very often relational) for data storage and management.
- A solutions to extract, load and transform (ELT) data to prepare it for analysis.
- Statistical analysis, reporting and data mining functions.
- Possibly advanced analytical applications that generate useful information through the use of data analytics and algorithms based on artificial intelligence (AI).
- Tools for the visualization and presentation of data to business users.

As you can see, warehouse architecture is made up of multiple levels called *tiers*. The bottom tier of the architecture is the database server, where data is loaded and stored. The middle tier consists of the analytics engine that is used to access and analyze the data. The top tier is the front-end client that presents results through reporting, analysis, and data mining tools.

A place of data warehouse in a basic architecture of a business system:

TODO make an image

```
                    ETL
Data          Stagin  Warehouse     Data        Users
sources       area                  marts
Operational           Meta data     Sales       Analysis (1,2)
```

```
system
Flat file             Raw data      Purchasing  Reporting (1,2)
                      Summary data  Inventory   Mining (2,3)
```

THE FOUR PILLARS OF THE DATA WAREHOUSE

William Inmon, one of the precursors of the data warehouse concept, points out that data warehouses are characterized by four fundamental conditions:

- Oriented to a single subject or a particular functional area. For example, it is oriented to company sales.
- Unify and create consistency among data from disparate sources.
- Persistent and immutable. Once data enters a data warehouse, it stays there and does not change.
- Structured in time intervals. To provide information from a historical perspective, data warehouses record information over different intervals, such as weekly, monthly, quarterly, etc.

A well-designed data warehouse is high-performance and responsive to queries. It also provides flexibility so users of the data warehouse can query from different points of view. Users can alternate between a high-level overview and deep queries at the greatest level of detail as they wish.

BENEFITS

You shouldn't think about data warehouse as a simple copy of data from the source transaction systems. In fact this is much more than copy, because architectural complexity provides the opportunity to:

- Integrate data from multiple sources into a single database and data model enabling a central view across the enterprise.

- Improve data quality, consistency and accuracy.
- Organize and disambiguate repetitive data.
- Independently on data source maintain data history.
- Behaves like a data buffer and separate analytics processing from transactional databases, which improves performance of both systems by mitigate the problem of database isolation level lock contention in transaction processing systems caused by attempts to run large, long-running analysis queries in transaction processing databases.
- Transform the data so that it makes sense to the business users.

Data warehouse concepts

Facts and dimensions

Facts are the measurable events related to the functional area covered by a data warehouse. They describe quantitative transactional data like measurements, metrics, or the values ready for analysis and consequently they are often of numerical nature. Facts are stored in fact tables, and have a foreign key relationship with a number of *dimension tables*. You can find a fact table at the center of a *snowflake schema* or *star schema*.

Dimensions, on the other hand, are collections of reference information about the facts in a data warehouse. Dimensions categorize and describe the facts recorded in a data warehouse to provide meaningful, categorized, and descriptive answers to business questions. A dimensional table stores information that provides dimensions of a fact and is joined by a foreign key to a fact table. You can find dimension tables at the edges of a snowflake or star schema. They contain detailed data that is descriptive, complete, and quite wordy.

To illustrate these concepts, consider the following set of fact and dimensions:

```
FACTS

Sales
==========
transaction_id
sales_dollars
sales_units
...
product_key (FK)
store_key (FK)
date_key (FK)


DIMENSIONS

Product                 Store               Date
===================     ==============      =============
product_key (PK)        store_key (PK)      date_key (PK)
product_description     store_name          date
product_type            store_type          year
category_name           store_region        day_of_week
...                     ...                 ...
```

For this set you can:

- *Calculate* sum of sales in dollars using only `sales_dollars` column from fact table.
- *Filter* your data by year using `year` column from `Date` dimension table.
- *Group by* category using `category_name` column from `Product` dimension.

The following table summarizes the major differences between facts and dimensions:

Characteristics	Fact	Dimension
What the data	Measurements, metrics, and facts	Descriptive attributes
Design	Located at the center of a star or snowflake schema	Located at the edges of the star or snowflake schema
Complexity	Atomic	Wordy, descriptive, complete
Data organization	Does not contain a hierarchy	Contains hierarchies

THE MOST COMMON DATA WAREHOUSE SCHEMAS

Data Warehouse environment usually transforms the relational data model into some special architectures. There are many schema models designed for data warehousing but the most commonly used are:

- star schema,
- snowflake schema,
- fact constellation schema.

A **star** schema is called so because the fact table sits at the center of the logical diagram, and the small dimensional tables branch off to form the points of the star. Each star schema database only has a single fact table.

A **snowflake** schema database is similar to a star schema in that it has a single fact table and many dimension tables. However, for a snowflake schema, each dimension table might have foreign keys that relate to other dimension tables.

The main difference between star schema and snowflake schema is that the dimension table of the snowflake schema is maintained in the

normalized form to reduce redundancy. The advantage here is that such tables (normalized) are easy to maintain and save storage space. However, it also means that more joins will be needed to execute the query. This will adversely impact system performance.

Fact constellation, also known as **galaxy**, is a schema for representing multidimensional model. It is a collection of multiple fact tables having some common dimension tables. It can be viewed as a collection of several star schemas and hence the name.

THE MOST COMMON DATA WAREHOUSE DESIGN

When it comes to designing a data warehouse, the two most commonly discussed methods are the approaches introduced by Bill Inmon and Ralph Kimball.

Inmon's method uses the relational (ER) model which is the 3NF whereas Kimbal's approach uses a multidimensional model which is a star schema and snowflakes.

Inmon argues that relational model helps attain enterprise-wide data consistency and simplifies to spawn off the data marts. In this approach, analytics systems can only access data from the enterprise data warehouse through data marts. The data is stored in the normalized form and the warehouse is not created directly. Instead, the data is fed into different data marts which the data is filtered down to the subset of specific needs. As an example, the sales department will have data that only used by the sales team. Because Bill Inmon recommend starting with building a centralized enterprise-wide data warehouse by several databases to the analytical needs of departments, which are later known as data marts. Hence for this approach the name *top-down* is used.

Kimbal argues that dimensional model helps actual users to understand, analyze, aggregate, and explore inconsistencies of data more easily. In this approach, keeping in mind the most important business aspects or departments, first you have to create data marts.

These provide a thin view into the organisational data and, as and when required, these can be combined into a larger data warehouse. This approach is known under the *bottom-up* name.

Kimball's bottom-up approach

DATA WAREHOUSE VS DATABASE

Characteristics	Warehouse	Database
Use for	Analytics, reporting, big data	Transaction processing
Data source	Data collected and normalized from many sources	Data captured as-is from a single source
Data capture	Bulk write operations typically on a predetermined batch schedule	Optimized for continuous write operations as new data is available to maximize transaction throughput
Data normalization	Denormalized schemas, such as the star schema or snowflake schema	Highly normalized, static schemas
Data storage	Optimized for simplicity of access and high-speed query performance	Optimized for high throughout write operations to a single row-oriented physical block

Data mart

A data mart is a data warehouse that serves the needs of a specific team or business unit, like finance, marketing, or sales. It is smaller, more focused, and may contain summaries of data that best serve its community of users. A data mart might be a portion of a data warehouse, too.

DATA MART VS DATA WAREHOUSE

Characteristics	Mart	Warehouse
Use by	A single community or department	Organization-wide
Use for	Decentralized, specific subject area	Centralized, multiple subject areas integrated together
Data source	A single or a few sources, or a portion of data already collected in a data warehouse	Many sources
Size	Small	Large

Data lake

Data lakes and data warehouses both act as repositories, but they are designed for very different purposes. Data warehouses work best for specific projects while data lakes are a centralized repository for all data, including structured, semi-structured, and unstructured. The data lake tends to ingest data very quickly and prepare it later as people need it. A data warehouse requires that the data be organized in a tabular format, which is where the schema comes into play. The tabular format is needed so that SQL can be used to query the data. But not all applications require data to be in tabular format. Some applications, like big data analytics, full text search, and machine learning, can access data even if it is 'semi-structured' or completely unstructured.

DATA LAKE VS DATA WAREHOUSE

Characteristics	Lake	Warehouse
Data	All data, including structured, semi-structured, and unstructured	Structured data, mostly relational data
Data quality	Any data that may or may not be curated (i.e. raw data)	Highly curated data that serves as the central version of the truth
Data source	A single or a few sources, or a portion of data already collected in a data warehouse	Many sources
Use by	Optimized for managing all incoming big data	Work best for specific projects with set resources

Data swamp: When your data lake gets messy and is unmanageable, it becomes a data swamp.

Data mesh

TODO

The data mesh is widely considered the next big architectural shift in data. Much in the same way that software engineering teams transitioned from monolithic applications to microservice architectures, the data mesh is, in many ways, the data platform version of microservices. The name *data mesh* comes from *mesh networking*, as this new model is also based on a decentralized architecture and proposes to move beyond a monolithic data lake.

NOTE

Mesh network

A mesh network (or simply meshnet) is a local network topology in which the infrastructure nodes (i.e. bridges, switches, and other infrastructure devices) connect directly, dynamically and non-hierarchically to as many other nodes as possible and cooperate with one another to efficiently route data to and from clients.

This lack of dependency on one node allows for every node to participate in the relay of information. Mesh networks dynamically self-organize and self-configure, which can reduce installation overhead. The ability to self-configure enables dynamic distribution of workloads, particularly in the event a few nodes should fail. This in turn contributes to fault-tolerance and reduced maintenance costs.

Mesh topology may be contrasted with conventional star/tree local network topologies in which the bridges/switches are directly linked to only a small subset of other bridges/switches, and the links between these infrastructure neighbours are hierarchical.

Data mesh is not a technology stack or physical architecture. Data mesh is a *process and architectural paradigm*, originated in 2019, that delegates responsibility for specific data sets to *domains*, or *areas* of the business that have the requisite subject matter expertise to know what the data is supposed to represent and how it is to be used. Instead of assuming that data will reside in a data lake, each area of business is responsible for choosing how to host and serve the datasets that they own. Unlike traditional monolithic data infrastructures that handle the consumption, storage, transformation, and output of data in one central data lake, a data mesh supports distributed, domain-specific data consumers and views “data-as-a-product,” with each domain handling their own data pipelines.

SECTION 6

Low level storage ideas

On-disk storage

On-disk data storage is a fundamental means of data storage utilizes low cost hard disk drives for long term storage. Generally it can be implemented via:

- a files on a distributed file system,
- a database.

DISTRIBUTED FILE SYSTEMS

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

A distributed file system is **not dedicated to handle datasets comprising a large number of small files** as this requires excessive disk-seek activity, slowing down the overall data access. It works best with fewer but **larger files accessed in a sequential manner**. As a result, distributed file system storage is suitable when large datasets of raw data are to be stored or when archiving of datasets

is required. Because of this feature, this means of storage may require combining multiple smaller files into a single file to enable optimum storage and processing either on operating system or file system level, or on application level. This allows to have increased performance when data must be accessed in streaming mode with no random reads and writes.

A distributed file system storage, by design, provides out of box redundancy and high availability by copying data to multiple locations via replication. This allows almost unlimited horizontal scaling.

DATABASES

xxx

TODO explain, say something about inmemory databases

one more abstraction layer

In-memory storage

An in-memory storage generally utilizes the main memory of a computer as its storage medium to provide fast data access reducing the latency of disk I/O operations.

This type of storage is appropriate when:

- Data arrives at a fast pace and requires realtime analytics or stream processing.
- The same dataset is required by multiple data processing jobs.

An in-memory solutions are not appropriate when:

- Data processing consists of batch processing.
- Very large amounts of data need to be persisted for a long time.
- Datasets are extremely large and do not fit into the available memory.
- High level of support for durable data storage is required.
- An enterprise has a limited budget, as setting up an in-memory storage may require upgrading processing nodes, which could either be done by node replacement or by adding more RAM.

CHAPTER 6

Processing concepts for Big Data

What you will **learn**:

- Why big data requires its own engineering

OLTP and OLAP

OLTP

The term *transaction* can have two different meanings:

- In the realm of computers or database, transaction denotes an *atomic change of state*.
- In the realm of business it typically denotes an exchange of data or informations.

It is not rare that transactions of the first type are used to record transactions of the second.

OLTP, *online transaction processing*, has been used to refer to processing in which the system responds immediately to user requests. An automated teller machine (ATM) for a bank is an example of a commercial transaction processing application. Online transaction processing applications have high throughput and are insert- or update-intensive. OLTP systems emphasize very fast query processing and maintaining data integrity in multi-access environments. Very often these applications are used concurrently by hundreds of users. The key goals of OLTP applications are availability, speed, concurrency and recoverability. Effectiveness of such systems is measured by the number of transactions per second. Usually OLTP databases contain detailed and current data.

OLAP

OLAP, *online analytical processing*, tools enable users to analyze data organized multidimensionally in an interactively way from multiple perspectives. Compared to OLTP, OLAP is generally characterized by much more complex queries involving aggregations, in a larger volume, for the purpose of business intelligence or reporting rather than to process transactions. Whereas OLTP systems process all kinds of queries (read, insert, update and delete), OLAP is generally optimized for read only and might not even support other kinds of queries. Effectiveness of such systems is measured by the response time.

Summary

The following table summarizes the major differences between OLTP and OLAP system design:

Characteristics	OLTP System Online Transaction Processing (Operational System)	OLAP System Online Analytical Processing (Data Warehouse)
Source of data	Operational data; OLTPs are the original source of the data	Consolidation data; OLAP data comes from the various OLTP databases
Purpose of data	To control and run fundamental business tasks	To help with planning, problem solving, and decision support
What the data	Reveals a snapshot of ongoing business processes	Multi-dimensional views of various kinds of business activities
Inserts and Updates	Short and fast inserts and updates initiated by end users	Periodic long-running batch jobs refresh the data

Characteristics	OLTP System Online Transaction Processing (Operational System)	OLAP System Online Analytical Processing (Data Warehouse)
Queries	Simple queries returning relatively few records	Often complex queries involving aggregations
Processing Speed	Typically very fast	Depends on the amount of data involved; batch data refreshes and complex queries may take many hours
Space Requirements	Can be relatively small if historical data is archived	Larger due to the existence of aggregation structures and history data; requires more indexes than OLTP
Database Design	Highly normalized with many tables	Typically de-normalized with fewer tables; use of star and/or snowflake schemas

SECTION 2

Paralelizm

Divide and conquer

In section 2: *Factors you should consider* of chapter 4: *Big Data paradigms* I tried to convince you to think parallel, to implement every solution as a set of jobs executed in parallel. Easy to say, harder to do because not every task or problem is open to parallel execution and very often you may have a difficulty deciding if this is possible or not.

You may find candidates of such an algorithms or ideas how to implement a new one for a new problem looking into algorithm gathered under the *divide-and-conquer* name. In computer science, divide and conquer is an algorithm design paradigm based on multi-branched recursion. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. Following you have some standard algorithms that share this divide and conquer nature:

- binary search,
- merge sort,
- quicksort,
- closest pair of points,

- Strassen algorithm for efficient matrices multiplication,
- Karatsuba algorithm for fast multiplication.

To be honest, divide-and-conquer approach is something you have to use, not exactly something you want to use. Without any doubts, in most cases it is much easier to implement data processing algorithm step by step than in "chunks". One problem with step by step approach is that you can not speed up this process infinitely. As you know, according to Amdahl's law, theoretical sped up is limited by the serial part of the program. For this reason, parallel computing with many processors is useful only for highly parallelizable programs.

The divide-and-conquer principle is generally achieved using one of the following approaches:

- **Parallel data processing**
  Parallel data processing involves the simultaneous execution of multiple sub-tasks that collectively comprise a larger task. The goal is to reduce the execution time by dividing a single larger task into multiple smaller tasks that run concurrently — in this way you make a gain in time.
- **Distributed data processing**
  Distributed data processing is closely related to parallel data processing however, it is always achieved through physically separate machines that are networked together.

Different faces of parallelizm

Among many different factors you can take into accounts talking about parallelizm is *what* is parallelised.

Task parallelism
Task parallelism refers to the parallelization of data processing by **dividing a task into sub-tasks** and running each sub-task on a separate node. Since the unit of division is task, each sub-task may executes (in parallel) a different algorithm, with its own copy of the same data or different data as its input.

Data parallelism
Data parallelism refers to the parallelization of data processing by **dividing a dataset into multiple datasets** and processing each sub-dataset in parallel. The sub-datasets are distributed across multiple nodes and are all processed using the same algorithm.

In both cases, at the end of the whole process, the output from multiple nodes is joined together to obtain the final set of results.

Cluster, grid, cloud, fog and edge

Cluster and grid

Both cluster and grid refer to a some kind of virtual super computer but they differ in essential details.

A cluster (of computers) is a set of loosely or tightly connected computers that work together so that, in many respects, they can be viewed as a single system. Cluster has each node set to perform the same task, controlled and scheduled by software.

Grids is composed of many networked loosely coupled computers acting together to perform large tasks. Grid computing combines computers from multiple administrative domains to reach a common goal, to solve a single task, and may then disappear if no need any more. Grid computers also tend to be more heterogeneous and geographically dispersed (thus not physically coupled) than cluster computers.

Grid is a "system" that manages resources under the direct control of different computers and connected computer networks. Grid resources can be administered by various organizations. Sharing resources follows the local resource management policy used in given organization.

The following table summarizes the major differences between cluster and grid:

Characteristics	Cluster	Grid
Range	Very local	Geographically dispersed
Type of resources	Unified hardware under the control of one system	Heterogeneous hardware and software
Number of resources	Constant	Very dynamic
Availability	Full control	Variable over time
Reliability	High	Low
Consistency	The same policies for all under the control of one organization	Various security and resource management requirements and policies

Whatever you choose, an additional benefit of these technologies is that they provide inherent redundancy and fault tolerance, as they consist of physically separate nodes.

Cloud

Cloud computing makes computer system resources, especially storage and computing power, *available on demand* without need of real access to hardware and direct active management by the user.

Edge

Edge computing is a distributed computing paradigm that brings computation and data storage closer to the sources of data – in close proximity to the physical location creating the data. Edge computing operates on "instant data" that is real-time data generated by sensors or users. With this technology, data is processed and transmitted to the devices instantly. In most cases edge nodes transmit all the data captured or generated by the device regardless of the importance of the data. This is not intended as a competitor of cloud but rather as an "input" layer separating cloud and big data tools from low level simple devices and protocols used to collect data. Both levels cooperate but require completely different hardware and software stacks. Such an approach is expected to improve response times and save bandwidth.

Fog

Edge and fog computing share a lot of similarities. Essentially, both are enablers of data traffic to the cloud. Edge is responsible for collecting data. Fog computing is a compute layer between the cloud and the edge. Where edge computing might send huge streams of data directly to the cloud, fog computing can receive the data from the edge layer before it reaches the cloud and then analyze and decide what's important and what isn't. The relevant data gets stored in the cloud, while the irrelevant data can be simply rejected, or analyzed locally at the fog layer to support or strengthen processing at this level for example improving learning models. You can say, fog computing acts as a mediator between the edge and the cloud for various purposes, such as data filtering, aggregating, transforming etc. In this way, fog computing saves a lot of bandwidth, processing and space in the cloud and transfers really expected data quickly.

Summary

Sometimes the term edge computing is used synonymously with fog computing. There is no clear consensus as to whether they are the same or not [TEC:1]:

"Fog computing and edge computing are effectively the same thing. Both are concerned with leveraging the computing capabilities within a local network to carry out computation tasks that would ordinarily have been carried out in the cloud," said Jessica Califano, head of marketing and communications at Temboo.

"Edge computing usually occurs directly on the devices to which the sensors are attached or a gateway device that is physically 'close' to the sensors. Fog computing moves the edge computing activities to processors that are connected to the LAN or into the LAN hardware itself so they may be physically more distant from the sensors and actuators." said Paul Butterworth, co-founder and CTO at Vantiq.

In my opinion it is reasonable to keep them separated as they logically separate totally different requirements about data acquisition, processing, storage etc.

Cloud layer

Operates on thousands of nodes. Implementing business logic. Use business analytics, business intelligence, big data processing, data warehousing.

Fog layer

Operates on millions of nodes (separated servers) in local networks. Responsible for preliminary data analysis and reduction.

Edge layer

Operates on billions of nodes, each of which is an embedded system responsible for real-time data processing. Processing understood as collecting data and sending them in chunks to save bandwidth. If it is relevant, also more sophisticated actions on data can be performed, however it is not very common in this layer.

Ways to process data

When it comes to big data, there are two main ways to process data. The first – and more traditional – approach is *batch processing*, the second is *real-time processing*.

Batch processing

As the term implies, batch-based data processing involves collecting a series of data, storing it until a given quantity of data has been collected, then processing all of that data as a one group named a *batch*.

It's much different from processing each piece of data right after it was collected.

Processing data in batches was a natural choice with older technologies when hardware, especially I/O operations, was a serious bottleneck of processing. To speedup you had to limit the time you needed to locate and read the data you wanted to use. Because common mass storages of that days were magnetic tapes the most effective way of getting data was sequential read without fast forwarding and rewinding the tape. To meet this requirements the data to be processed was arranged in advance in the correct order; the batch was prepared.

Doing so reduces the number of I/O events that need to take place but also help to save network bandwidth by compressing data within batches. Very often batch processing is the processing of a large volume of data, that is collected over a period of time, all at once. It means that

jobs are typically completed on well known data set (this data set is defined and known before you start processing) in non-stop, sequential order. Because the whole batch is completely ready before you start processing it can be used offline – hence very often *offline processing* term is used instead of *batch processing*. You have a complete control as to when to start the processing of this type. This way you can delay (postpone) processing till the computer is not executing very many tasks. It also helps balance the overall load and system utilization. Obviously, because of data size, it will take large amount of time for that data to be processed. Batch processing works well in situations where you don't need real-time analytics results, and when it is more important to process large volumes of data to get more detailed insights than it is to get fast analytics results. Another worth to mention characteristic is that because you have all data in a batch you can infer much more from that data than from processing single data record one by one.

Real-time

Although you probably intuitively understand the term real-time, it is not so simple to define it precisely. The meaning of *real* is highly application specific.

Sub-second real time

This type of real-time is typical for engineers – when they say *real-time*, they are usually referring to sub-second response time, very often in context of embedded systems. In this kind of real-time data processing, extreme levels of performance are key to success – even nanoseconds count. If this is what you think when you say *real-time data processing*, it means you need the data to come in, the condition for response to be evaluated, and the response to happen, all generally, in less than a

second. A lot of work to be done. And if someone else's system can do it a few nanoseconds faster, you might lose out. In this kind of real-time, pushing the limits of performance isn't a bonus; it's a necessity.

You think about this type of real-timing when you say:

- The sensors data is monitored in real-time to catch problems early.
- This stock exchange application has to bid in real-time or we'll lose money.

Human comfortable real-time response

This type of real-time is typical for ordinary users – when they say *real-time*, they are usually referring to response time that not bore or frustrate the users. The performance requirement for this kind of processing is usually a couple of seconds – performance matters, but it may not be the number one criteria. In some cases, a difference of a single second can be critical, but for the most part, as long as the application responds before the user decides to give up what he or she was going to do, then the performance requirement is met.

You think about this type of real-timing when you say:

- This website needs to respond to user requests in real-time or we'll lose sales.
- We need real-time visualizations for our business intelligence team, no matter how big the data.

Take an action as a response for some event

This type of real-time is typical for changes in the data or user actions. When you say *real-time* in this context, you are usually referring to response time opposite of scheduled. If something happen (an event occurs), do not wait but act – so called event-driven processing model. The performance requirement for this is generally before another event happens. Instead of happening in a particular time interval, event-driven data processing happens when a certain action or condition triggers it. You don't know precisely when you will need data processing done, but as soon as a certain thing happens, that's when the need for data processing is triggered.

There are actually two different performance requirements for event-driven data processing. First, the data processing system has to be finished working and ready to start again before the next event happens. So, if on average, the events happen no closer together than five minutes, a data processing time frame of 2-3 minutes is excellent. If the events tend to happen an average of 10 seconds apart, then clearly, a 2-3 minute processing time would be unacceptable.

You think about this type of real-timing when you say:

- As changes are made to the database, the replication process copies them out to the cluster in real-time.

Stream processing

This type of real-time is typical for processing the data as it flows in, one piece at a time. It's similar to event based real time processing.

Also here if something happen (you receive new data), do not wait but act – exactly as you do with events. The difference is that events may come with an unpredictable times, with varying frequency; simply speaking: event occurs, you process. In stream processing, very often,

you receive data with constant frequency are you can estimate the time it arrives. What is more important, once the data starts coming in, it generally doesn't end – just like a water in a river or stream; that is why *streaming data processing*. The performance requirement for streaming data processing is you must process data as fast as the data flows in, taking into account that because of its endless nature, there are no chances to postpone some processing and complete later.

In this type of real-time processing you have to be able to process the data continuously, without start-up or clean-up overhead. Streaming data processing also requires a way to deal with occasional system failures without massive data loss.

You think about this type of real-timing when you say:

- The server information in this data center is monitored in real-time to catch problems early.
- Weather conditions are recorded continuously to generate up-to-date weather forecast in real-time.

SUMMARY

So you have different meanings of *real-time* term, and all of them are correct. This ambiguity tends some people to use "*near real-time*" term in the sense of any one of the above definitions, maybe aside from sub-second, to emphasize the fact that something must happen "immediately" but we can wait a "moment" for results. Of course both "immediately" and "moment" are very fuzzy and imprecise and highly dependent on the specific case.

Very neat summary statement is: **respond before you lose the customer**. This, in some ways, is the best possible way to think about real-time when designing any data processing systems. Regardless of the level of performance your system has in any given situation, if you end up losing the customer, then it's simply too slow. This may be a premise to move up to (more restricted) real-time data processing.

Stream data processing

These days when people say *real-time data processing* in context of data processing, they are most likely referring to streaming data processing introduced in previous section. It refers to processing data as soon as it is collected, with results available virtually instantaneously. The individual event or stream considered in a given time interval is generally small in size, but its continuous nature results in very large datasets. Because of its nature, stream (real-time) processing is also known as *online processing*.

Why is stream processing needed?

- It is needed to provide the expected value in a strictly specified time window.
- Stream processing naturally fit with time series data and detecting patterns over time.
- Stream processing let you handle large data and retain only useful pieces.
  Sometimes data is huge and it is not even possible to store it. Working with streams you can examine a small portion of data and leave only what is important to you. You can get rid of data that does not carry

important information or aggregate them into more compact form. For example, if you observe that temperature sensor returns the same value for the last hour, it is enough to store this value only once along with information about its validity – timestamp when this value was recorded first time and when it was recorded last time.

- Stream processing let us handle large data with reasonable hardware/software stack.
  With streams you process data as they come in hence spread the processing over time. Hence stream processing can work with a lot less hardware than batch processing (you have less data to process compared to batch processing). With the almost instant flow, systems do not require large amounts of data to be stored.

Summary

The following table summarizes the major differences between batch and real-time processing:

Characteristics	Batch	Real-time (stream)
Known as	Offline processing	Online processing
Type of action	Retrospective	Reactive
Size of data	Large; huge collection of records	Small; individual records or micro batches of few records
Inference based on	All at once	One by one
Latency	Hight	Minimal
Use when	Processing needs multiple passes through full data	Processing can be done with a single pass over the data or has temporal locality

Characteristics	Batch	Real-time (stream)
Persistency	Data must be persisted to the persistent storage before it can be processed.	Data is processed in-memory as it is captured before being persisted to the disk
Data access	Sequential reads/writes	Random reads/writes
Setup	Easy	Hard
Cost	Low	High

SECTION 5

Map Reduce

No doubt MapReduce is the best known algorithm of batch processing type having its roots in divide-and-conquer principle. Putting into practice the idea of parallel and distributed computing realized over a set of commodity hardware nodes caused big data computing, so far reserved only for selected researchers and computer scientist, found its way to ordinary people, like you and me.

MapReduce does not require that the input data conform to any particular data model. Therefore, it can be used to process schema-less datasets. Although in many cases MapReduce returns result much faster than traditional algorithms, it is not expected, as all batch processing algorithms, to have low latency.

The MapReduce processing engine try to works differently compared to the traditional data processing paradigm.

Traditionally, data processing requires moving data from the storage node to the processing node that runs the data processing algorithm. This approach works fine for "small" datasets. The larger dataset is, the higher network bandwidth utilization and high latency you get and this in turn means that more time you waste. In consequence, in extreme cases, for large amount of the data moving it generates more overhead than the actual processing.

In MapReduce, an algorithm instead of the data is moved to the nodes that store the data, thereby eliminating the need to move

the data first. The real problem with this approach is how to spread your data so it will be on correct processing node from the very beginning when you start storing them. If this condition cannot be met, the algorithm will of course work, but less efficiently because data transfer will be necessary. Column families NoSQL databases helps to accomplish this.

Note: Sometimes I use the term *algorithm* referring to MapReduce, but it is discussable if it is an algorithm or rather a framework. See *MapReduce vs classic divide-and-conquer approach* section for detailed discussion.

At very high level of consideration a single processing run of the MapReduce is composed of a *map task* and a *reduce task*. Diving into details, each task consists of multiple stages:

- **map**: split, map, [combine], partition;
- **reduce**: shuffle, sort, reduce.

Suppose, you have a person list and your job is to count how many of them falls into age ranges: 0-9 years, 10-19 years, 20-29 years etc.

For simplicity assume the input consist of 10 persons: `Anna (12)`, `Bob (34)`, `Celine (63)`, `Diana (51)`, `Edmund (55)`, `Fracesco (67)`, `Giselle (15)`, `Henry (57)`, `Isabell (32)`, `Jane (39)`.

SPLIT STAGE

At this stage the input is divided into *splits*. This will distribute the work among all the map nodes. Each split is parsed into its constituent records as a key-value pair `(K1, V1)`. The key is usually the ordinal position of the record, and the value is the actual record.

Example:
For example MapReduce could divide exemplary personal data in three splits in the order in which they appear:

- **split 1**: Anna (12), Bob (34), Celine (63);
  set of pairs: `{"anna": 12, "bob": 34, "celine": 63}`

- **split 2**: Diana (51), Edmund (55), Fracesco (67);
  set of pairs: `{"diana": 51, "edmund": 55, "francesco": 67}`

- **split 3**: Giselle (15), Henry (57), Isabell (32), Jane (39);
  set of pairs: `{"giselle": 15, "henry": 57, "isabell": 32, "jane": 39}`

MAP STAGE

The map function executes user-defined logic on splits which are the output of previous stage. Each split generally contains multiple key-value pairs, and the *mapper* is run once for each key-value pair in the split. The mapper processes each key-value pair and in result generates a new key-value pair as its output `(K2, V2)`. The output key `K2` can either be the same as the input key `K1` or any other serializable user-defined object. Similarly, the output value `V2` can either be the same as the input value `V1` or any other serializable user-defined object.

When all records of the split have been processed, the output is a list of key-value pairs `list(K2, V2)` where multiple key-value pairs can exist for the same key. It should be noted that for an input key-value pair `(K1, V1)`, a mapper can generate one key-value pair, can generate multiple key-value pairs (demultiplexing), each possibly with different key and/or value, or may not produce any output key-value pair (filtering).

Example:
The mapping process remains the same on all the nodes. Continuing our example, in each of the mapper you give a hardcoded value (1) to each of the person. The rationale behind giving a hardcoded value equal to 1 is that every given person, in itself, will occur once. Next a list of key-value pair is created where the key is nothing but the age range and value is one. So, for our set of splits you have the following list of a key-value pairs:

- **node 1**: `{"10-19": 1, "30-39": 1, "60-69": 1}`
- **node 2**: `{"50-59": 1, "50-59": 1, "60-69": 1}`
- **node 3**: `{"10-19": 1, "50-59": 1, "30-39": 1, "30-39": 1}`

COMBINE STAGE

Generally, the output of the map function is handled directly by the reduce function. However, map tasks and reduce tasks are mostly run over different nodes. This requires moving data between mappers and reducers. This data movement can consume a lot of bandwidth and directly contributes to processing latency. With larger datasets, the time taken to move the data between map and reduce stages can exceed the actual processing undertaken by the map and reduce tasks. For this reason, the MapReduce engine provides an optional combine function that summarizes a mapper's output before it gets processed by the reducer.

A *combiner* is essentially a function run locally to group a mapper's output on the same node as the mapper. As a combiner you can use a reducer function, or you can define separate custom function.

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

Example:
First combine combines all values for a given key from the mapper output, creating multiple key-value pairs where the key is not repeated and the value exists as a list of all corresponding values for that key:

- **node 1**: `{"10-19": [1], "30-39": [1], "60-69": [1]}`
- **node 2**: `{"50-59": [1, 1], "60-69": [1]}`
- **node 3**: `{"10-19": [1], "50-59": [1], "30-39": [1, 1]}`

Next combiner works as a reducer function (see next stage) that locally groups a mapper's output on the same node as the mapper:

- **node 1**: `{"10-19": 1, "30-39": 1, "60-69": 1}`
- **node 2**: `{"50-59": 2, "60-69": 1}`
- **node 3**: `{"10-19": 1, "50-59": 1, "30-39": 2}`

PARTITION STAGE

The output from the previous stage is divided into partitions between reducer instances. Although each partition contains multiple key-value pairs, **all records for a particular key are assigned to the same partition**. The MapReduce engine guarantees a random and fair

distribution between reducers while making sure that all of the same keys across multiple mappers end up with the same reducer instance.

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

Example:
The output from the previous stage is divided into partitions between reducer instances. Although each partition contains multiple key-value pairs, the rule is that all records for a particular key are assigned to the same partition:

- **node 1**: `{"10-19": 1 -> P1, "30-39": 1 -> P1, "60-69": 1 -> P3}`
- **node 2**: `{"50-59": 2 -> P2, "60-69": 1 -> P3}`
- **node 3**: `{"10-19": 1 -> P1, "50-59": 1 -> P2, "30-39": 2 -> P1}`

SHUFFLE AND SORT STAGE

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

Next, the MapReduce engine automatically groups and sorts the key-value pairs according to the keys so that the output contains a sorted list

of all input keys and their values with the same keys appearing together. The way in which keys are grouped and sorted can be customized.

This merge creates a single key-value pair per group, where key is the group key and the value is the list of all group values.

Example:
Output from all partitioners is copied across the network to the nodes running the reduce task (shuffling):

- **partition on node 1**: `{"10–19": 1, "30–39": 1, "10–19": 1, "30–39": 2}`
- **partition on node 2**: `{"50–59": 2, "50–59": 1}`
- **partition on node 3**: `{"60–69": 1, "60–69": 1}`

Next, the MapReduce engine automatically groups data on nodes by key:

- **partition on node 1**: `{"30–39": 1, "30–39": 2, "10–19": 1, "10–19": 1}`
- **partition on node 2**: `{"50–59": 2, "50–59": 1}`
- **partition on node 3**: `{"60–69": 1, "60–69": 1}`

and sorts the key-value pairs according to the keys so that the output contains a sorted list of all input keys and their values with the same keys appearing together:

- **partition on node 1**: `{"10–19": [1, 1], "30–39": [1, 2]}`
- **partition on node 2**: `{"50–59": [2, 1]}`
- **partition on node 3**: `{"60–69": [1, 1]}`

REDUCE STAGE

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

Example:
The reducer job is to summarize its input. For each key-value pair that it receives, the list of values stored in the value part of the pair is processed:

- **node 1**: `{"10-19": 2, "30-39": 3}`
- **node 2**: `{"50-59": 3}`
- **node 3**: `{"60-69": 2}`

Finally the last key-value pair is written out:

```
"10-19": 2

"30-39": 3

"50-59": 3

"60-69": 2
```

MapReduce vs classic divide-and-conquer approach

"Dividing a problem to smaller ones until the individual problems can be solved independently and then combining them to answer the original question is known as the divide and conquer algorithm design technique.

Recently, this approach to solve computational problems especially in the domain of very large data sets has been referred to as MapReduce rather than divide and conquer.

My question is as follows: Is MapReduce anything more than a proprietary framework that relies on the divide and conquer approach, or are there details to it that make it unique in some respect?"

There are a lot of discussion, starting with the question similar to the above, wether is there (conceptual) novelty in MapReduce somewhere, or is it just a new implementation of old ideas useful in certain scenarios [MR:1-3] to mention only a few examples. MapReduce was hailed as revolution of distributed programming and it even got patented [MR:5] but there have also been critics [MR:4]. So, is MapReduce something new compared to well known divide-and-conquer approach? The answers varies – let's see some of them.

- There is no novelty. [...] nothing new in computation, or even distributed computing was discovered by MapReduce.

- It's not a super-sophisticated concept, but a very useful piece of infrastructure. MapReduce is a framework for implementing divide-and-conquer algorithms in an extremely scalable way, by automatically distributing units-of-work to nodes in an arbitrarily large cluster of computers and automatically handling failures of individual nodes by redistributing the unit-of-work to another node.

- If you're asking about the MapReduce architecture, then it is very much just a divide and conquer technique. However, any useful MapReduce architecture will have mountains of other infrastructure in place to efficiently "divide", "conquer", and finally "reduce" the problem set. With a large MapReduce deployment (1000's of compute nodes) these steps to partition the work, compute something, and then finally collect all results is non-trivial. Things like load balancing, dead node detection, saving interim state (for long running problems), are hard problems by themselves.

- The paradigm is named after Lisp's `map()` and `reduce()` functions. It was certainly a new idea to make a massively parallel version of `map` and `reduce`.

- Because Google did it. There's nothing particularly wrong with it, but it isn't particularly novel, either. People have been doing this kinds of parallel processing for decades.

- MapReduce is more about segregation and aggregation.

- MapReduce is not simply a divide and conquer technique, though it looks that way in many examples. In the mapping step you can and frequently want to do a one-to-many relation. Thus you're not simply dividing into cases.

- MapReduce diverges from most divide and conquer systems in a fairly fundamental way, but one that's so simple that many people almost miss it. The real genius of it is in tagging the intermediate results.

  In a typical (previous) divide and conquer system, you divide the work up serially, execute work packets in parallel, and then merge the results from that work serially again.

  In MapReduce, you divide the work up serially, execute work packets in parallel, and tag the results to indicate which results go with which

other results. The merging is then serial for all the results with the same tag, but can be executed in parallel for results that have different tags.

- MapReduce is not divide and conquer. It does not involve the repeated application of an algorithm to a smaller subset of the previous input. It's a pipeline (a function specified as a composition of simpler functions) where pipeline stages are alternating map and reduce operations. Different stages can perform different operations.

  MapReduce does not break new ground in the theory of computation – it does not show a new way of decomposing a problem into simpler operations. It does show that particular simpler operations are practical for a particular class of problem.

  The MapReduce paper's contribution was:

  - Evaluating a pipeline of two well understood orthogonal operators that can be distributed efficiently and fault-tolerantly on a particular problem: creating a text index of large corpus.
  - Benchmarking map-reduce on that problem to show how much data is transferred between nodes and how latency differences in stages affect overall latency.
  - Showing how to make the system fault tolerant so machine failures during computation can be compensated for automatically.
  - Identifying specific useful implementation choices and optimizations.

Some of the critiques fall into these classes:

- "Map/reduce does not break new ground in theory of computation." True. The original paper's contribution was that these well-

understood operators with a specific set of optimizations had been successfully used to solve real problems more easily and fault-tolerantly than one-off solutions.

- "This distributed computation doesn't easily decompose into map & reduce operations". Fair enough, but many do.

- "A pipeline of n map/reduce stages require latency proportional to the number of reduce steps of the pipeline before any results are produced." Probably true. The reduce operator does have to receive all its input before it can produce a complete output.

- "Map/reduce is overkill for this use-case." Maybe. When engineers find a shiny new hammer, they tend to go looking for anything that looks like a nail. That doesn't mean that the hammer isn't a well-made tool for a certain niche.

- "Map/reduce is a poor replacement for a relational DB." True. If a relational DB scales to your data-set then wonderful for you – you have options.

SECTION 6

SCV principle

One of the most important concept related to Big Data processing is called the *Speed, Consistency and Volume (SCV) principle*. If you have some basic knowledge about NoSQL concepts, you should have heard about the *CAP theorem*. Whereas the CAP theorem is primarily related to distributed data storage, the SCV principle is related to distributed data processing. As the CAP theorem states that a distributed data storage can be designed to support only two of the three requirements: *consistency*, *high availability* and *partition tolerance*, the SCV principle states that a distributed data processing system can be designed to support only two of the three requirements: *speed*, *consistency* and *volume*.

- **Speed** This refers to how fast the data can be processed once it is generated.

- **Consistency** This refers to the accuracy and the precision of the results. In common language both terms are used interchangeably as synonyms but they means different things. Results are deemed accurate if they are close to the correct value and precise if close to each other. For example, a more consistent system may use all present data, resulting in greater accuracy and precision as compared to a less consistent system that makes use of sampling techniques, which can result in lower accuracy with an acceptable level of precision.

- **Volume** This refers to the amount of data that can be processed.

NOTE

Accuracy and precision

To illustrate the fundamental difference between accuracy and precision, the analogy to a shooting target is often given:

Accurate and precise: Considering the case of a rifle with calibrated sighting scope in the hands of a professional marksman with a steady hand you will get accuracy and precision.

No accurate and precise: Considering the result for a professional marksman using a rifle whose sighting scope is not calibrated we will get no accuracy and precision.

Accurate and no precise: Considering the result for an amateur (with a shaky hand) using a calibrated rifle we will get accuracy and no precision.

No accurate and no precise: Considering the result for an amateur shooting an un-calibrated rifle we will get no accuracy and no precision.

The SCV principle states that:

- If speed (S) and consistency (C) are required, it is not possible to process high volumes of data (V) because large amounts of data slow down data processing.

- If consistency (C) and processing of high volumes of data (V) are required, it is not possible to process the data at high speed (S) as achieving high speed data processing requires smaller data volumes to ensure accuracy and precision.

- If high volume (V) data processing with high speed (S) is required, the processed results will not be consistent (C) since high-speed processing of large amounts of data limits the number of data you can process and may involves sampling the data, which in turn may reduce consistency.

In Big Data environments, making the maximum amount of data available is mandatory for performing in-depth analysis (as for a distributed data storage). Assuming that data (V) loss is unacceptable, generally a realtime data analysis system will either be S+V or C+V, depending upon whether speed (S) or consistent results (C) is favored. If you talk about real-time system than by definition you have S+V and consistency (C) will be compromised.

CHAPTER 7

Message queues

What you will **learn**:

- Why big data requires its own engineering

XXX

XX

xx

CHAPTER 8

Adoption issues and considerations

What you will **learn**:

- Why big data requires its own engineering

XXX

XX

xx

CHAPTER 9

Do we really need Big Data

What you will **learn**:

- Why big data requires its own engineering

XXX

XX

xx

CHAPTER 10

XXX

What you will **learn**:

- Why big data requires its own engineering

XXX

XX

xx

Chapter P1

Getting data

You will do:

In this part you will extend your knowledge with information on more advanced topics.

You will learn:

- What a **structure** is and how it differs from **class**.
- **Inheritance** with **type checking** and **access control**.
- How to handle **exception**al situation.

Text files

Text files? In XXI century? Are you kidding?

TODO

People forget about this but... It is very important !!!

SECTION 2

Working with files in Python

Text files

TODO

Binary files

TODO

The ability to work with files at the level of reading a single line of text or a block of bytes is particularly desirable as you should never assume you can read a whole file into memory or at least you should be prepared for this uncomfortable situation.

Working with JSON data in Python

Since its inception, JSON has quickly become the de facto standard for information exchange replacing in most cases much more verbose XML. Python supports JSON natively — it comes with a built-in package called `json` for encoding and decoding JSON data.

The process of writing JSON data to disk is usually called *encoding* or sometimes *serialization*, while *decoding* (*deserialization*) is for reading data into memory. In general *serialization* is any process converting a native object to a string so it can be saved on disk or transmitted across the network; and conversely *deserialization* includes any process that aims to convert a string to a native object (recreate object based on string).

Keep in mind that JSON is a semi-structured format contains tags (keys) to separate semantic elements and enforce hierarchies of records and fields within the data. Therefore, it may be difficult to read JSON file in chunks (which is required when available RAM is much smaller than file size), especially when its hierarchy is unknown.

Encoding JSON

Imagine you have a python object, for example something like this:

```
data = {
  "string": "text",
  "integer": 12,
  "real": 12.34,
  "list": [1, 1.2, "abc"],
  "dictionary": {"k1": "v1", "k2": "v2"}
}
```

After importing required library:

```
import json
```

you can save it with simple instruction:

```
with open("simple.json", "w") as output_file:
  json.dump(data, output_file)
```

Open the `simple.json` file or display its contents with for example `cat` command:

```
big_data:ch02 fulmanp$ cat simple.json
{"string": "text", "integer": 12, "real": 12.34, "list": [1,
1.2, "abc"], "dictionary": {"k1": "v1", "k2": "v2"}}
```

If for some reason you want to continue using this serialized JSON data in your program, you could write it to a native Python `str` object:

```
json_string = json.dumps(data)
print(json_string)
```

```
big_data:ch02 fulmanp$ python3 test_json.py
{"string": "text", "integer": 12, "real": 12.34, "list": [1,
1.2, "abc"], "dictionary": {"k1": "v1", "k2": "v2"}}
```

JSON is not very sophisticated format so in consequence saving it is not highly customizable.

You can use the `indent` keyword argument to specify the indentation size for nested structures:

```
with open("simple.json", "w") as output_file:
  json.dump(data, output_file, indent=2)
```

```
big_data:ch02 fulmanp$ cat simple.json
{
  "string": "text",
  "integer": 12,
  "real": 12.34,
  "list": [
    1,
    1.2,
    "abc"
  ],
  "dictionary": {
    "k1": "v1",
    "k2": "v2"
  }
}
```

Another formatting option is the `separators` keyword argument. This is a 2-tuple of the separator strings `(item_separator, key_separator)` by default defined as `(", ", ": ")`. If you want to save some space, you can use more compact separators with no spaces: `(",", ":")`. Take a look at the sample JSON again to see where these separators come into play:

```
with open("simple.json", "w") as output_file:
  json.dump(data, output_file, separators=(",", ":"))
```

```
big_data:ch02 fulmanp$ cat simple.json
{"string":"text","integer":12,"real":12.34,"list":
[1,1.2,"abc"],"dictionary":{"k1":"v1","k2":"v2"}}
```

You can sort by key data saved in JSON with `sort_keys` argument:

```
with open("simple.json", "w") as output_file:
  json.dump(data, output_file, indent=2, sort_keys=True)
```

```
big_data:ch02 fulmanp$ cat simple.json
{
  "dictionary": {
    "k1": "v1",
    "k2": "v2"
  },
  "integer": 12,
  "list": [
    1,
    1.2,
    "abc"
  ],
  "real": 12.34,
  "string": "text"
}
```

Saving custom data

Another two optional arguments of `dump` you can use to serialize custom objects — objects that aren't natively serializable:

- `default` defines a function that is called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`. If not specified, `TypeError` is raised.

- `cls` kwarg defines a custom `JSONEncoder` subclass (e.g. one that overrides the `default` method to serialize additional types). If not specified, `JSONEncoder` is used.

Consider the following custom data structures:

```
class Person:
  def __init__(self, name, address):
    self.name = name
    self.address = address
```

```
class Address:
  def __init__(self, country, city, street, number):
    self.country = country
    self.city = city
    self.street = street
    self.number = number


address = Address("AB", "Cd", "Ef", 12)
person = Person("Abcd", address)
```

Now you can implement function to translate a custom object `Person` into JSON. Without this you will get an error as in the following example:

```
person_json = json.dumps(person, indent=2)
print(person_json)
```

```
[...]
TypeError: Object of type Person is not JSON serializable
```

Serializing function can do anything you want and think is needed to represent your data in terms of the built-in types json already understands. Simply speaking, you translate the more complex object into a simpler representation, which the json module can translates into JSON. Every time you do this, remember to choose minimum representation sufficient to recreate this object. In case of `Person` object this could be:

```
def encode_person(person):
  if isinstance(person, Person):
    p = {
      "name": person.name,
      "address": {
        "country": person.address.country,
        "city": person.address.city,
        "street": person.address.street,
        "number": person.address.number,
      }
    }
    return p
  else:
```

```
    type_name = z.__class__.__name__
    raise TypeError(f"Object of type '{type_name}' is not JSON
serializable")
```

Now you can serialize any `Person` object:

```
person_json = json.dumps(
  person,
  indent=2,
  default=encode_person
)
print(person_json)
```

The other common approach is to subclass the standard `JSONEncoder` and override its `default` method:

```
class PersonEncoder(json.JSONEncoder):
  def default(self, person):
    if isinstance(person, Person):
      p = {
        "name": person.name,
        "address": {
          "country": person.address.country,
          "city": person.address.city,
          "street": person.address.street,
          "number": person.address.number,
        }
      }
      return p
    else:
      return super().default(z)
```

Making the following call, you will get the same result as before:

```
person_json = json.dumps(
  person,
  indent=2,
  cls=PersonEncoder
)
print(person_json)
```

Decoding JSON

TODO

file `simple.json` :

```
{
  "string": "text",
  "integer": 12,
  "real": 12.34,
  "list": [
    1,
    1.2,
    "abc"
  ],
  "dictionary": {
    "k1": "v1",
    "k2": "v2"
  }
}
```

xxx

```
with open("simple.json", "r") as input_file:
  data = json.load(input_file)
  print(data["dictionary"]["k2"])
```

```
v2
```

Loading data from JSON string:

```
json_string = json.dumps(data)
print(json_string)
data = json.loads(json_string)
print(data["dictionary"]["k2"])
```

```
{"string": "text", "integer": 12, "real": 12.34, "list": [1,
1.2, "abc"], "dictionary": {"k1": "v1", "k2": "v2"}}
v2
```

Deserializing function

TODO

xxx

```
def decode_person(dct):
  if "address" in dct:
    address = Address(
      dct["address"]["country"],
      dct["address"]["city"],
      dct["address"]["street"],
      dct["address"]["number"]
    )

    return Person(dct["name"], address)
  else:
    return dct


person = json.loads(person_json, object_hook = decode_person)
print(person.name)
```

```
Abcd
```

Subclass the standard `JSONDecoder`:

```
class PersonDecoder(json.JSONDecoder):
  def __init__(self, *args, **kwargs):
    json.JSONDecoder.__init__(
      self,
      object_hook=self.dict_to_object,
      *args,
      **kwargs
    )

  def dict_to_object(self, dct):
    if "address" in dct:
      address = Address(
        dct["address"]["country"],
        dct["address"]["city"],
        dct["address"]["street"],
        dct["address"]["number"]
      )
```

```
      return Person(dct["name"], address)
    else:
      return dct
```

Making the following call, you will get the same result as before:

```
person = json.loads(
  person_json,
  cls = PersonDecoder)
print(type(person))
print(person.name)
```

SECTION 4

Working with CSV data in Python

Although the CSV (Comma Separated Values) format is not a precise standard, it became the most common data exchange (import and export) format for spreadsheets and databases.

Because of CSV structure, where data are separated by delimiter and each "portion" of data takes exactly on line, there are no problems with processing files of size much greater than available RAM. You can read line by line in loop or at worst you can read data field by data field if data stored in fields are really huge.

TODO

XXX

importing required library

```
import csv
```

xxx

```
with open("test.csv", "w", newline="") as output_file:
  writer = csv.writer(output_file)
  writer.writerow(["Abc", 100])
  writer.writerow(["Bcd", 200])
```

```
  writer.writerow(["Cde", 300])
  writer.writerow(["Def", 400])
```

xxx

```
big_data:ch02 fulmanp$ cat test.csv
Abc,100
Bcd,200
Cde,300
Def,400
```

xxx

```
with open("test.csv", "w", newline="") as output_file:
  writer = csv.writer(output_file, delimiter=";")
  rows = [
    ["Abc", 100],
    ["Bcd", 200],
    ["Cde", 300],
    ["Def", 400]
  ]
  writer.writerows(rows)
```

xxx

```
big_data:ch02 fulmanp$ cat test.csv
Abc;100
Bcd;200
Cde;300
Def;400
```

xxx

```
with open("test.csv", newline="") as input_file:
  reader = csv.reader(input_file)
  for row in reader:
    print(*row, sep=",")
```

xxx

```
big_data:ch02 fulmanp$ python3 test_csv.py
Abc,100
Bcd,200
Cde,300
Def,400
```

xx

```
with open("test.csv", "w", newline="") as output_file:
  fieldnames = ["foo_string", "foo_integer"]
  writer = csv.DictWriter(output_file, fieldnames=fieldnames)
  writer.writeheader()
  writer.writerow({"foo_string": "Abc", "foo_integer": 100})
  writer.writerow({"foo_string": "Bcd", "foo_integer": 200})
  writer.writerow({"foo_string": "Cde", "foo_integer": 300})
  writer.writerow({"foo_string": "Def", "foo_integer": 400})
```

xx

```
big_data:ch02 fulmanp$ cat test.csv
foo_string,foo_integer
Abc,100
Bcd,200
Cde,300
Def,400
```

xx

```
with open('test.csv', newline='') as input_file:
  reader = csv.DictReader(input_file)
  for heading in reader.fieldnames:
    print(heading, end=' ')

  print('\n----------------------')

  for row in reader:
    print("{1},{0}".format(
      row['foo_string'],
      row['foo_integer']
    ))
```

xx

```
big_data:ch02 fulmanp$ python3 test_csv.py
foo_string foo_integer
----------------------
100,Abc
200,Bcd
300,Cde
400,Def
```

compact

```
res = [fields for fields in csv.reader(open("test.csv",
newline=''))]
print(res)
```

xx

```
big_data:ch02 fulmanp$ test_csv.py
[['foo_string', 'foo_integer'], ['Abc', '100'], ['Bcd', '200'],
['Cde', '300'], ['Def', '400']]
```

SECTION 5

Working with regular expressions

TODO

Regular expressions simplifies data extraction but requires a block of data to be loaded into RAM. In most cases it is not a problem as you may assume that size of extracted fields are much lower than RAM and you can load data in chunks.

Encoding JSON

TODO

SECTION 6

Summary

The ability to work with files at the level of reading a single line of text or a block of bytes is particularly desirable as **working with data sets which are big you should never assume you can read a whole file into memory** or at least you should be prepared for this uncomfortable situation.

RULE

Never assume you can read a whole file into memory.

JSON is a semi-structured format contains tags (keys) to separate semantic elements and enforce hierarchies of records and fields within the data. Therefore, it may be difficult to read JSON file in chunks (which is required when available RAM is much smaller than file size), especially when its hierarchy is unknown.

CSV is a structured format, where data are separated by delimiter and each "portion" of data takes exactly on line, which resembles rows and columns like structure. There are no problems with processing files of size much greater than available RAM. You can read line by line in loop or at worst you can read data field by data field if data stored in fields are really huge.

Regular expressions simplifies data extraction but requires a block of data to be loaded into RAM. In most cases it is not a problem as you

may assume that size of extracted fields are much lower than RAM and you can load data in chunks.

CHAPTER P2

Cleansing, transforming, and integrating data

You will do:

In this part you will xxx

You will learn:

- xxx.

XXX

TODO Short intro to the topic

cc

204B, (i.e. a size reduction of 62%)

Chapter P3

Multiprocessing

You will do:

In this part you will xxx

You will learn:

- xxx.

Log

TODO Short intro to the topic

At the top of log file you should save the checksum of the file for which this report is created. This will prevent you from accidentally using your log data to a similar but different file (which is not that hard to do).

Keep the scanning log in the form:

```
{"checksum": {"md5": "123", "sha1": "456"}}
{"line": 71, "errors": [{"column": 81, "type": "ERROR_1"}]}
{"line": 72, "errors": [{"column": 82, "type": "ERROR_2"}]}
{"line": 73, "errors": [{"column": 83, "type": "ERROR_3"}]}
{"line": 74, "errors": [{"column": 84, "type": "ERROR_4"}]}
```

Why so? Why not as valid JSON, e.g .:

```
{
  "meta": {"checksum": {"md5": "123", "sha1": "456"}},
  "log": [
    {"line": 71, "errors": [{"column": 81, "type": "ERROR_1"}]},
    {"line": 72, "errors": [{"column": 82, "type": "ERROR_2"}]},
    {"line": 73, "errors": [{"column": 83, "type": "ERROR_3"}]},
    {"line": 74, "errors": [{"column": 84, "type": "ERROR_4"}]}
  ]
}
```

This allows you to read data line by line without having to read the entire log file and put it into memory. Try to load the simplest, not

Bibliography

Bibliography

[BD] Big Data definitions, history and books

1. *Big data*, retrieved: 2021-04-02, https://en.wikipedia.org/wiki/Big_data

2. Thomas Erl, Wajid Khattak, Paul Buhler, *Big Data Fundamentals: Concepts, Drivers & Techniques*, Part of the The Prentice Hall Service Technology Series from Thomas Erl series. Published Jan 5, 2016 by Prentice Hall

3. *The Punched Card Tabulator*, retrieved: 2021-04-02, https://www.ibm.com/ibm/history/ibm100/us/en/icons/tabulator/

4. Dawn E. Holmes, *Big Data: A Very Short Introduction*, Oxford University Press, 2017

5. *Vulnerability – Introducing 10th V of Big Data*, retrieved: 2022-03-19, https://data-flair.training/blogs/vs-of-big-data-introduction/

6. Arockia Panimalar S., Varnekha Shree S., Veneshia Kathrine, *The 17 V's Of Big Data*, International Research Journal of Engineering and Technology (IRJET), Volume: 04, Issue: 09, Sep-2017, pp. 329-333 retrieved 2020-10-06, https://www.irjet.net/archives/V4/i9/IRJET-V4I957.pdf

7. Nawsher Khan, Arshi Naim, Mohammad Rashid Hussain, Quadri Noorulhasan Naveed, Naim Ahmad, Shamimul Qamar, *The 51 V's Of Big Data: Survey, Technologies, Characteristics, Opportunities, Issues and Challenges*, COINS '19: Proceedings of the International Conference on Omni-Layer Intelligent SystemsMay 2019, pp. 19-24, retrieved 2020-10-06, https://doi.org/10.1145/3312614.3312623

[BDC] Big Data Characteristics

1. *Big Data characteristics,* retrieved: 2019-02-16, https://fulmanski.pl/tutorials/computer-science/big-data/big-data-concepts-and-terminology/#big_data_characteristics

2. *Why are Data Lakes the Future of Big Data?,* retrieved: 2022-03-20, https://www.oracle.com/emea/a/ocom/docs/why-are-data-lakes-the-future-of-big-data-infographic.pdf

3. Zhamak Dehghani, *Data Mesh. Delivering Data-Driven Value at Scale*, O'Reilly 2022, retrieved: 2022-03-20

[BDP] Big Data problems

1. *Xerox scanners/photocopiers randomly alter numbers in scanned documents*, retrieved 2022-03-19 https://www.dkriesel.com/en/blog/2013/0802_xerox-workcentres_are_switching_written_numbers_when_scanning

[BDS] Big Data Sources

1. *How many Libraries of Congress does it take?*, retrieved 2021-04-02, https://blogs.loc.gov/thesignal/2012/03/how-many-libraries-of-congress-does-it-take/

2. *A "Library of Congress" Worth of Data: It's All In How You Define It,* retrieved: 2021-04-02, https://blogs.loc.gov/thesignal/2012/04/a-library-of-congress-worth-of-data-its-all-in-how-you-define-it/

3. *Transferring "Libraries of Congress" of Data*, retrieved: 2021-04-02, https://blogs.loc.gov/thesignal/2011/07/transferring-libraries-of-congress-of-data/

4. *Storage. What data to record*, retrieved 2021-04-02, https://home.cern/science/computing/storage

[COM] Computers and computing

1. Rodney Brooks, *A Quadrillion Mainframes on Your Lap . Your laptop is way more powerful than you might realize*, IEEE Spectrum, 21 Dec 2021, retrieved: 2022-03-18, https://spectrum.ieee.org/ibm-mainframe

2. Xeon, retrieved: 2021-01-28, https://en.wikipedia.org/wiki/Xeon#3000-series_"Conroe"

3. CPU Mega List, retrieved: 2021-01-28 https://www.cpubenchmark.net/CPU_mega_page.html

4. Year on Year Performance, Updated 28th of January 2021, https://www.cpubenchmark.net/year-on-year.html

5. Should you buy a laptop with 8th-gen or 9th-gen Core CPU? It's all about cores and clocks, retrieved: 2019-05-07, https://www.pcworld.com/article/3390994/intel-8th-gen-vs-9th-gen-core-cpu-which-should-you-buy.html

[CS] Computer Science

1. *Talk:Machine epsilon*, retrieved: 2021-04-03, https://en.wikipedia.org/wiki/Talk%3AMachine_epsilon

2. *Machine epsilon*, retrieved: 2021-04-03,
https://en.wikipedia.org/wiki/Machine_epsilon

[Data]

1. *CrowdFlower Data Scientist Report (2017)*, retrieved 2022-02-20
https://visit.figure-eight.com/rs/416-ZBE-142/images/CrowdFlower_DataScienceReport.pdf

[MR] MapReduce

1. *What is the novelty in MapReduce?*, retrieved 2022-06-13
https://cs.stackexchange.com/questions/3019/what-is-the-novelty-in-mapreduce

2. *Is MapReduce anything more than just an application of divide and conquer?*, retrieved 2022-06-13
https://softwareengineering.stackexchange.com/questions/98800/is-mapreduce-anything-more-than-just-an-application-of-divide-and-conquer

3. *Why is "MapReduce" so hyped?*, retrieved 2022-06-13
https://www.quora.com/Why-is-MapReduce-so-hyped

4. *DeWitt and Stonebraker's "MapReduce: A major step backwards*, retrieved 2022-06-13
http://craig-henderson.blogspot.com/2009/11/dewitt-and-stonebrakers-mapreduce-major.html

5. *US Patent 7,650,331: "System and method for efficient large-scale data processing*, retrieved 2022-06-13
https://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=/netahtml/PTO/srchnum.htm&r=1&f=G&l=50&s1=7,650,331.PN.&OS=PN/7,650,331&RS=PN/7,650,331

[TEC] Technology

1. *Edge Computing vs. Fog Computing: What's the Difference?*, retrieved: 2022-06-16,
https://www.cmswire.com/information-management/edge-computing-vs-fog-computing-whats-the-difference/

[XXX]

1. *Big Mac Index*, retrieved 2022-03-17
https://en.wikipedia.org/wiki/Big_Mac_Index

2. *The Economist. The Big Mac index*, retrieved 2022-03-17
https://www.economist.com/big-mac-index

3. *Coordinates converter*, retrieved 2022-03-23
https://coordinates-converter.com

4. *Convert between Latitude/Longitude & UTM coordinates / MGRS grid references*, retrieved 2022-03-23
https://www.movable-type.co.uk/scripts/latlong-utm-mgrs.html

5. Quality management best practices managing quality the 'toyota way'!, retrieved 2022-04-02
https://www.hpcimedia.com/images/PDF/NSF.pdf