Big Data Analytics

Mikael Svahnberg (Mikael.Svahnberg@bth.se)

This is part of a course on Applied Cloud Computing and Big Data, offered by Blekinge Institute of Technology. Version 1.0

Copyright © 2023 Mikael Svahnberg Mikael.Svahnberg@bth.se

MIT License

Copyright (c) 2023 Mikael Svahnberg

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Table of Contents

1 Document Overview

The course Applied Cloud Computing and Big Data is a 7.5 ECTS course offered by Blekinge Institute of Technology. The course is organised around three themes, and all three themes must be completed to complete the course:

- Cloud Provisioning and Deployment
- The Business Case for Cloud Computing
- Big Data Analytics

The course is divided across two source code repositories:

- `https://github.com/mickesv/ProvisioningDeployment.git` contains the instructions and source code for the Cloud Provisioning and Deployment, and the Business Case for Cloud Computing parts of the course.
- `https://github.com/mickesv/BigDataAnalytics.git` contains the instructions and source code for the Big Data Analytics part of the course.

This document covers the third of these themes, i.e. Big Data Analytics. The document is structured as a series of introductions to different topics mixed with hands-on tasks to exercise these topics.

The topic introductions and practical exercises are further punctuated by quizzes that (a) serve as checkpoints of important concepts that have been covered to that point, but also (b) introduce further reading material to deepen your understanding of the cloud computing business case. **These quizzes are not marked**, but serve as a learning aid; for you to summarise what you have learnt so far, and for us to keep track of your progress through the material.

The theme is examined through two assignments (See [assignmentCodeStream], page 21, and [assignmentAllAtOnce], page 37,), where you put what you have learnt into practice.

This document is available in different formats:

- as web pages on the course platform
- as a pdf file
- as a TEXInfo tutorial

The TEXInfo tutorial may require a bit more introduction. Access this version e.g. through the command `info Big-Data-Analytics` or in your favourite editor though `C-u C-h i Big-Data-Analytics.info` . Basic navigation in the document is done with the following keys (try `info info` for more details and further key commands):

- <spc> (space) scroll forward
- 'b' Beginning of current node
- 'e' End of current node
- 'n' Next node
- 'p' Previous node
- 'u' Up (usually (but not always) to the top index page)
- <enter> on a cross reference to open it.

2 A Note on Reading Research Articles

This course may suggest research articles for you to read. This may seem a daunting task for you, but there are some general guidelines that may help you:

- Start by reading the abstract and the conclusions. These will tell you what problem or area the article is addressing, and what the article contributed to solving the problem.
- The articles in this course have – to a large extent – been chosen because they summarise the topics in various ways. To find these summaries, look for *lists, figures, and tables* in the article. Read these. Read the text around them, or where the figures and tables were referenced, in order to get an explanation of how to interpret them.

Generally, interpret each section of an article as follows:

- The *introduction* section in an article puts the problem into context, and might give you clues as to how other researchers have solved it before. You can use this section to get a generic overview.
- The *related work* section should discuss in further detail what other researchers have done that is similar to, or relates to, the article you are reading. This may give you an idea of how others have addressed the problem, and may give you pointers to other articles that you would wish to read.
- The *methodology* section explains how the study was conducted. Once an article is published (which they naturally are in this course), this is mostly interesting if you distrust some of the results and want to see if anything has been missed when constructing the study.
- The *execution and results* sections describe how the study was executed (in particular discrepancies from the planned methodology), and what the raw results were. Most of the time, you can safely skip these sections. Sometimes however, the *results* section is combined with the *analysis* section.
- The *analysis* section "bakes" the results and tries to answer the research questions (address the identified problem). This is probably the most interesting section for you to read.
- The *discussion* section should raise the view and try to see what the results actually mean in a bigger context. What can you do with the results? "So what?". This is a tricky section to read, since the authors want you to believe that their results are the best thing invented since hot porridge, but at the same time they have to identify threats to the validity of the study. Validity threats are things that could have influenced the results instead of the sought after effect. This may be things that happened at the same time, inadequacies in the researchers skills and abilities, or inadequacies in the research design. Please remember, when (or if) reading this section, that most of the identified validity threats are minor obstacles (or the article would not have been published) that may impact the scientific view of the results more than the practical significance or usefulness of them.
- The *references* may give you ideas for other articles that you would like to read.

The bottom line is that when being told to read a lot of research articles, the trick is to learn how to not read them while still getting the gist of them. Hopefully, the "map" described above may provide some help in identifying the parts of an article that are important for you.

3 Introduction to Big Data

Big Data is Dead I am seeing more and more texts that argue that *Big Data is Dead* (see e.g. J. Tigani Big Data is Dead (`https://motherduck.com/blog/big-data-is-dead/`) (last checked 2023-03-30)). Their argument is essentially that while the technology to deal with big data is sound, few users actually need it; few users actually have enough data to be a problem for a modern computer.

The question that remains for us to study is thus: *What killed Big Data?*

To be able to answer this question, we must first understand what big data was (or is). Chen et al. 2014 discuss the "four V's" of Big Data. Crude as they are, they are still a decent starting point with which to understand what Big Data is. The following are the claims used to define Big Data:

Volume Big Data is, as the name implies, lots and lots of data.

Velocity Velocity refers to the rate and amount at which new data arrives and need to be ingested and analysed. Essentially, if you have more data generators than you have capacity to receive and organise the data, you're in trouble.

Variety Not all data arrives in the same format, which complicates your data reception.

Value/Density
: The signal-to-noise ratio is low in Big Data, meaning that there is a lot of data to process in order to pin down any meaningful information. Once you do find the information, however, the claim is that this information is valuable enough to make it worthwhile.

I repeat that these are the *claims* about what constitutes big data. It may be possible to deal with each one of these in isolation, but it is the combination of them and the scale of them that makes Big Data a difficult problem. But this also means that unless either of them (at least either of the first three) actually constitute a problem, then there is no Big Data. Finding the killer of Big Data is thus a matter of finding out *who isn't there anymore*. So, without further ado, let us invite the suspects into the library one by one and explore which of them might be the killer.

3.1 Volume : The Size of Big Data

We may go about defining the size of Big Data in two ways. One is to look at examples of large datasets and gather anecdotal evidence about what may be Big Data. The other is to look at storage and processing capacity, and establish a "lower limit" for what may be Big Data.

First, then, some examples. Starting with something simple, like text, Project Gutenberg (`https://www.gutenberg.org/`) contains over 70000 e-books with expired copyrights. Each book is usually available in a couple of different formats, primarily including plain text, but also html, epub, and PDF. According to Kiwix (`https://wiki.kiwix.org/wiki/Content`), it weighs in at around 75 GB. In comparison, the English Wikipedia appears to be around 100GB (I am too lazy to actually download this and figure out if this is all of Wikipedia, whether it includes images and other media, and whether the zim file format is compressed or not).

Source code is just another type of text, right? Github provides a dataset (`https://console.cloud.google.com/bigquery?p=bigquery-public-data&d=github_repos&page=dataset&project=vagrant-test-2022&ws=!1m4!1m3!3m2!1sbigquery-public-data!2sgithub_repos`) covering nearly 3 million public open-source licensed repositories; this repo is around 500 GB (which consumes 2.5 TB of disk space). As before, I have not checked whether this number contains all versions of all

files (GitHub is, after all, a configuration management utility), and whether binaries (such as images, generated pdf documents, etc.) are included in this as well.

What about image data? OpenStreetMap (`https://www.openstreetmap.org/`) is a community-driven collection of map data. Maps for the whole planet in XML format lands in at around 125 GB. Note that this is "just" the map data and does not include e.g. any satellite images, so the question is whether we should call this images or not. A more personal example of "unstructured" images can be found closer to home: Both my wife and myself are avid hobby photographers with two kids. Our personal photo database have grown to ~500 GB over the course of twenty years.

Audio and video? That ought to take up lots and lots of space. For commercial reasons, this type of data is difficult to get access to, but we may get some ideas based on some research datasets that *are* made available. For example, Spotify Research (`https://podcastsdataset.byspotify.com/`) has published a dataset of 100 000 hours worth of podcasts, with a size of ~2TB (See "100,000 Podcasts: A Spoken English Document Corpus" by Ann Clifton, Sravana Reddy, Yongze Yu, Aasish Pappu, Rezvaneh Rezapour, Hamed Bonab, Maria Eskevich, Gareth Jones, Jussi Karlgren, Ben Carterette, and Rosie Jones, COLING 2020 `https://www.aclweb.org/anthology/2020.coling-main.519/`). Youtube (`https://thinkcomputers.org/what-is-the-capacity-of-youtube-servers/`) is naturally coy about how much space they require, and I am not certain they even know or care themselves how much data they have stored. In their case, the defining factor is probably the velocity at which data arrives and the velocity at which data is requested, confident in that they have done as much as they can in order to save storage space.

With these examples it is clear that even a modern laptop has enough storage capacity to store even a couple of these datasets. Yes, there are cases where this becomes difficult, and finding a scalable backup solution for our photo collection is ever a problem, but it is still clear that in most cases *it is not the volume itself that is the problem, but what one desires to do with the data.*

Essentially, one can optimise for *writing* data, for *storing* data, or for *exploring* data.

Optimising for writing data implies that when new data arrives it should quickly be added to whatever data is already stored. ACID database qualities (Atomicity, Consistency, Isolation, Durability) are less important than just getting the data in there and becoming *eventually consistent*. It is often argued that NoSQL databases have excellent write performance simply because the usual database consistency is not an issue here. Whatever is received is stored as a single document.

In reality, this suffers from two major issues: The first is that even if the *database* support "free-form" data where everything is a document and each item in the database may have their own attributes, some form of structure is often imposed by the database access framework (e.g. Mongoose, which is used to access MongoDB databases from node.js programs). This is not such a big deal after all, since performance-wise it only means that data items need to be well behaved objects in your program before you push them to the database. When reading data, on the other hand, each data item needs to fit into the pre-defined structure.

The second issue is more of a problem, though. To accept any data in the form it comes is to abdicate from the responsibility of designing the data. This will (not may; it *will*) create maintainability issues when some items are added with some attributes and other items use other attributes. When reading data, one must carefully check to see if all relevant attributes are specified for each item before any processing of the items can be done. Your code-base gets littered with passive aggressive data verification every place where the database is queried. In other words; the "write anything" approach is truly only optimised for write speed, and may come with a significant cost for storing and exploring the data.

When optimising for storing data the first question to ask is always: "Do we *need* to store this data?" Any data that is stored also need to be maintained. As mentioned, managing backups is a constant worry with my personal photo collection, but this is only a small portion of what needs to be taken care of. Just the fact that new disk space needs to be constantly added to keep up with the arrival of new data, and the data need to be structured such that it can spread out across the available disk space. If you keep the data in a database, then the database need to be able to grow across multiple disks or even multiple computers. If you keep your data in a regular filesystem, you still need to index the data so that you can easily find the right file on the right disk on the right computer.

The second question is: "What is the *cost* of keeping data?" The information therein needs to be kept up to date and relevant, otherwise it is a nice historical archive but probably of lesser value. *Bit rot* refers to a physical degradation process where bits are accidentally flipped over time, eventually rendering data inaccessible, but *Data Rot* can be seen as a wider problem, where the data itself may remain intact but the usefulness of the data degrades over time. Today, it is also important to keep abreast with current legal requirements such as GDPR and whether data about citizens must be stored within their country of origin, etc. Even if you are fine now, there is a cost associated with ensuring that you comply even with future changes to legal requirements. If, on the other hand, you have decided to not store a particular piece of data, then you do not need to worry about whether you are in compliance.

The third storage decision is: "What are the most suitable formats in which to store the data?" One good example of this is Project Gutenberg mentioned earlier. They may receive books in any format, and they also serve the books in a number of different formats. Internally, however, they are only stored in a small set of formats. Plaintext (ASCII) because this format will remain readable even in the future, and HTML because it is easy to generate other formats based on this (and because it, too, is only a slightly more advanced version of plaintext). Plaintext is a good alternative since it is less vulnerable to the aforementioned bit rot, and for text storage it is also the smallest possible format, which reduces requirements on backup and disk space. In a pinch, it is also compressible (at the cost of becoming less future proof), meaning that even more data can be stored before running out of disk space.

Exploring the data includes executing queries on the data as well as retrieving complete data items. If this is the most frequent activity performed on the data, then you may need to compromise other concerns. For example, you might add an index to quickly find answers to common queries, but this comes at a cost of requiring more storage and that it will take longer to insert new items. Duplicating some data can also help avoiding more complex data queries, but again at a storage cost. Other "tricks" include to store and maintain searchable data in a relational database while keeping the actual data items outside, essentially stored as opaque blobs (binary large objects).

There are, however, things that can be done to optimise *most* of the queries without adding storage overhead to all items. For example, all of the examples discussed earlier have "active sets" of data where some items are requested more frequently. I am certain they also have more and less common search patterns for navigating the data. Among the data in the active set, I would also venture that only a portion of them are actually being updated – the rest must be available but can be statically served.

Mind you, organising data into an active set also implies a cost. At regular intervals the active set must be updated to ensure that it indeed consist of the most used data. If, for example, the active set consist of all entries from the last week or month, then an update needs to run at least once per week/month. If the active set is "last 30 days", then you need to update it every day. Moreover, you also need to decide what to do with data that is no longer in the active set; how available should it remain? What quality requirements do you have on e.g. accessibility? Security? Recoverability?

Summary In most cases, current volumes of data is manageable. It is not easy, and there are many design decisions to take in order to keep the data volumes manageable, but it is doable even with old fashioned tools such as relational databases and regular file systems. There are costs associated with increasing sizes, but with the right data design decisions, the technology is eminently able to keep up.

This is strike one for big data: *Volume is no longer an issue.*

3.2 Velocity : The Speed of Noise

There are two sides to velocity: keeping up with the rate with which data arrives, and keeping up with the rate with which data is requested. In both cases, a good starting point is to look at what classic software architecture design suggests. The approach by Bass et al. (Bass, L., Clements, P., & Kazman, R. (2012). Software architecture in practice, third edition. Reading MA: Addison-Wesley Publishing Co.) is to make use of *architecture tactics*, i.e. high level architectural solutions for how one may address different quality requirements. Looking at the quality attribute Performance, the tactics are split into two separate approaches, i.e. *control resource demand*, and *manage resources*.

Manage Resources focus on adding more computing resources to deal with the amount of data. In different variations, this is what one does by scaling up and scaling out in a cloud computing solution. Tactics such as *introduce concurrency*, *maintain multiple copies of computation*, and *maintain multiple copies of data* are all approaches to "throw more computers at the problem".

Control Resource Demand are slightly more advanced solutions to the problem, since they focus on making the best possible use of the resources that are available. *Manage sampling rate*, for example, implies that one should not try to take on more requests than there are resources available to process, whereas *limit event response* is about enqueuing requests for later processing when there is more time/resources available. In contrast, the tactics *reduce overhead* and *increase resource efficiency* suggest altering the processing algorithm to something more efficient.

How to manage the rate with which data arrives depends (of course) on what needs to be done with the data, and the requirements on the data stream itself. If, for example, the data stream needs to be processed so that actions can be taken immediately, there is only so much one can do in terms of increasing resource efficiency, after which it becomes a matter of managing resources instead.

An example to illustrate this is billing data in telephony systems. In the Good Old Days (TM), the billing systems would collect call data at regular intervals from all telephone switches (manage sampling rate) and process them so that every customer could get an invoice at the end of the month based on how much they had called. With the introduction of pre-paid, this was no longer sufficient. Instead, the moment a customer connected, data about this must be sent *post haste* through the billing system to figure out the price of the current call and the amount left to call for by this customer, etc., and ultimately terminate the call when the customer did not have any more pre-paid money to call for. Once the processing algorithms are optimised and overhead reduced, the only thing remaining is to add more computing resources.

That's not entirely true, though. Some more things can be done: work with the data generators (in this case the telephone switches) so that the data is more pre-processed even before it enters the billing system, and simplify the required processing e.g. by a flatter structure of what it should cost to call different types of numbers, regions, or countries.

Working with the data generators can often be fruitful, especially if the data is generated in client software under your own control. The data which are processed fastest by the server is the data not sent at all. This connects to the *value* aspect of Big Data; rather than collecting all the data and hope to at a later stage figure out which information is valuable, some up-front design to cull the initial data can often work miracles on the required resources.

I named this section "the speed of noise". If most of the data that arrive is without value, then it really does not matter whether algorithms and platforms can be improved sufficiently to take care of it: it is still junk.

Strike two for big data: *Lack of up-front planning doth not Big Data make.*

But what if the data *aren't* junk? The billing data above clearly has value and need to be processed. Yes, but does it need to be stored? Does all of it need to be stored? Does all of it need to be stored for all eternity? Does it need to be processed down to its final form within hard relative requirements? We still have the option to *limit event response* and *prioritise events* to ensure that we do the needful right now and the rest eventually.

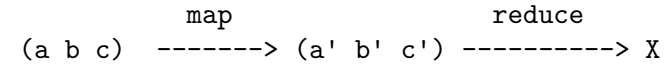
Analogous to the discussion above about optimising for collecting the *right* data to facilitate processing of new data, the next step is to optimise the storage of the data in order to facilitate querying of the data. We have grown comfortable with the idea that storage is cheap, and because we *can* generate new reports out of old data someone *might*, and thus we'd better hold on to all data for all eternity.

Fortunately (for big data), generating reports is only one small part of what one may do with data. Different types of data analysis that can be done include:

Linear Data Analysis, where each datum is studied once and perhaps transformed into some new state. In the end, an analysis across all data is produced. Examples of such queries may count how often a certain phenomenon occurs, or the correlation between different attributes of each datum. Even more concretely, one might wish to transform the data about each phone call to add the cost of it, or count of how often a particular number is called, or collecting information about the duration of all phone calls in order to create a new tariff.

Combinatory Data Analysis is similar, but include joining several datum into a single entity in order to conduct the analysis. If you remember your database theory, this is the realm of inner, left, right, and outer joins. With a bit of luck (and a certain amount of foresight in your database design), you may even be able to use database queries to perform much of this type of data analysis. In fact, this is really not that different from the linear data analysis except that it may take a bit longer to set up and structure the data for this type of queries. The join essentially creates a new data set, and if similar queries are going to be run frequently it may be worthwhile to store this new data set in some way or at least instruct the database to index the data so that similar queries can be run faster.

Some types of data analysis can be parallelised, and especially linear or combinatory data analysis lends itself well to splitting up the work in smaller parallelisable computations, followed by one or a series of merge steps to calculate a final result.



However, not all algorithms lend themselves to this type of independent '`map`' step. Optimisation problems in particular may require frequent synchronisation between the otherwise independent workers. Consider the following algorithm (This is a very crude reinterpretation of the Particle Swarm Optimisation (PSO) Algorithm):

Step 1: Distribute workers randomly across the entire search realm.

Step 2: For each worker, look around where they are and identify the nearest "best fit".

Step 3: Let all workers vote on who has the best "best fit".

Step 4: Move all workers closer to the best best fit.

Step 5: Unless all workers are in the same place, goto Step 2.

Step 3 is important in this algorithm since otherwise most workers will fall down in a local optimum and never go looking for anything better. By synchronising across all workers, this is – somewhat – avoided. But the key point here is that this synchronisation step is a common feature of many optimisation algorithms for that particular reason (to avoid local optima), and it presents an obstacle to large scale parallelisation of the algorithm. A few factors play a role here:

- How often the workers need to synchronise (synchronisation is often done as a part of every iteration in the original algorithms).
- How time-consuming the calculations are for each iteration and worker.
- How much memory each worker requires per iteration.
- How many workers there are.

The algorithms themselves usually include additional parameters such as how much to move each worker in each iteration, and how many iterations that are allowed (or whatever other end-criteria is used), but these affect the execution time in general but not the scalability of the algorithm. The scalability trade-off is whether the computations are so time consuming that it is worth the overhead effort of scaling out, synchronise the executables for the workers over the network, allocating memory to run each worker, etc., and then also synchronise all workers between each iteration. On the other hand, running things locally might be an option if the number of workers (and their collective execution time per iteration) is small enough to not warrant such overhead costs.

I will refer to these types of analyses as *Synchronisation-Intense Data Analysis*, and the issue with them is that while it may be possible to parallelise such algorithms they do not necessarily scale linearly.

The last type of analysis that I would like to name is *NP-Complete Data Analysis*. Note that whereas in computer science NP-Complete is a very specific and narrow class of problems with a mathematical definition of what makes them NP-complete, I am perhaps not adhering strictly to such formal definitions in my naming. What I mean is that in some analysis problems every new datum needs to be collated against all existing data, leading to an exponential increase of execution time rather than a polynomial increase as the dataset grows. For a one-off analysis this does not matter much if it takes a day instead of an hour to perform an analysis. However, for a continuous stream of data, it becomes difficult to both analyse the data in the stream and add it to the existing dataset. In these instances, it quickly becomes important to *redefine the problem* to avoid such pitfalls. Exactly how to redefine the problem depends on the problem itself and the analysis one wishes to perform on a given dataset.

Summary The velocity with which new data arrives is not necessarily a problem with modern hardware, but there are still some considerations. Firstly, one may wish to perform some early filtering of noise and valuable data, e.g. to keep the storage volumes down. Secondly, there is a difference between arrival velocity and request velocity, and the toolbox for addressing them differs significantly. Thirdly, if new data needs to be analysed at the same pace as it arrives there are several analysis types and some are absolutely not suitable for real-time analysis. The best option then is to redefine the problem so that it can be solved with more mundane analysis tools.

All in all, these considerations add up so that in the end even *small data can cause big challenges*. Thus, velocity remains in the room and cannot be accused of having killed Big Data.

3.3 Variety : Different Origins or Different Formats

The reason to include *variety* as one of the Big Data challenges is not necessarily because data from the same source may arrive in different formats. Of course this also happens, for example

when data arrives in the form of a text document. Advocates for NoSQL databases would like you to believe that this type of variety is one of the biggest challenges, and that is why you should want a *document store* rather than *column-based data* as you would have e.g. in a relational SQL database. In contrast, I would argue that this type of variety is quite trivial to deal with by a bit of up-front design to understand what the problem is, which data has a bearing on the problem, and how to analyse it.

Rather, the challenge with respect to data variety stems from the fact that the data analysis may require input from many different sources, and it may not be possible to know up-front which data entries from each source that will be necessary for the analysis. This means that it may not be possible to filter the data streams beforehand. Data acquisition thus has to resort to collecting and storing all the data from the sources. A certain amount of cleaning can be done on the data from each source (e.g. remove duplicate entries and entries that are obviously out of scope, and convert the data to an item in the database), but it is often not possible to perform any more advanced filtering. *This* is the real challenge with respect to data variety: it is impossible to know up front which data has value and which should be discarded, and this complicates many of the solutions previously discussed for dealing with volume and velocity (or makes them impossible to use).

Storage volume may no longer be an issue for Big Data, but we did argue that velocity may be a challenge depending on the type of analysis that needs to be done. The challenge with variety thus means that the dataset may grow faster than it would otherwise have to do, which may impact the analysis velocity.

The toolbox is the same as before, i.e. to try to filter as much as possible as early as possible, but with an added emphasis on regular culling of the already collected data. As before, the challenges with Big Data is not about finding a computer which is fast enough, but rather to intelligently analyse the data needs and devise strategies for dealing with this.

In fact, let's look a little closer at the cost of intelligent analysis. A simplified formula for calculating the power consumption of a cloud computer (based on "How can I Calculate CO_2-eq emissions for my Azure VM? (last checked 2023-04-11)) is: `E[Wh] = (n_cpu * P_cpu[W] * time[h])` where `n_cpu` is the number of CPUs and `P_cpu` is the power consumption per CPU. The formula suggested by S. Bergman also includes other factors such as memory power consumption and the power consumption of any GPUs, but for simplicity's sake these can be ignored for now. What remains now is to estimate the power consumption per CPU and how much of that CPU which is being used by us (as opposed to any other tenant on the same cloud server).

Thermal Design Power (TDP) is a measure listed by hardware manufacturers, so with a bit of matching between what type of computer and CPU is offered by the cloud provider and what the manufacturer of that specific CPU lists for their TDP, one may find out the energy consumption of one CPU running at full speed: `TDP_cpu[W]` . Multiply this with our load `l[percent]` and the time `t[h]`, and we may arrive at an estimated energy consumption for our application: `E[Wh]= (n_cpu * TDP_cpu * l * t)`.

Assume we want to reduce the load with for example 10% over a 24h period on a bog-standard dual CPU cloud machine, we may insert the following numbers: `E = (2*165*0.10*24) = 792 Wh` $\approx$ `0.8 kWh`. Being in south Sweden and thus tightly connected to the European power market (which, it should be noted, is likely *not* where the cloud provider is located), this would mean saving 100g of CO_2 equivalents per day. Run the server for a year to save 36kg of CO_2 equivalents.

Humans are part of a natural cycle where we only eat "fresh" organic material (unless you have a penchant for eating crude oil), and thus we are only re-releasing the CO_2 from this organic material back into the natural cycle that will feed the next crop. However, for the sake of this calculation, we *do* exhale roughly 1kg of CO_2 equivalents per day. Working 8h/day for 20

days/month a single developer can thus spend almost half a year thinking about how to reduce the load by 10% for the same amount of CO_2 equivalents.

Yes, this is extremely simplified. Yes, CO_2 equivalents does not necessarily mean that all equivalents are equal. Yes, humans are part of the natural cycle so their breath is not a net increase of CO_2 into the atmosphere. Yes, we should probably look closer at how they get to work (do they walk or ride a bike, or do they drive a Chelsea tractor) instead. No, they are not allowed to use a computer or anything else consuming energy while thinking. No, not even coffee. But you get the idea: There is a lot to be saved by analysing the problem before throwing more resources at it.

Defensive Programming Cycling back to the *variety* aspect of big data that we started this section with, the lure of not being forced to structure one's data is that one adopts a more *lassiez-faire* approach to processing data upon arrival. Over time, this leads to an extremely defensive programming style for data retrieval, a not at all extreme example of which is presented below:

```
return DocumentStore.find(query)
    .sort({time:1})
    .catch( err => {
        debug('Error while fetching data. \nQuery: %s \n Error: %s', query, err);
    })
    .then( result => {
      if (result) {
          if (result.field) {
              if (!isNaN(parseFloat(result.field))) {
                  if (0 >= parseFloat(result.field)) {
                      // Finally, here is some code
                      // that actually does something useful
                  } else {
                      throw 'result.field is negative';
                  }
              } else {
                  throw 'result.field is not a float';
              }
          } else {
              throw 'result.field does not exist';
          }
      } else {
          throw 'did not receive any result from query';
      }
    })
    .catch( err => {
        debug('Error while parsing data. \nQuery: %s \nError: %s', query, err);
    });
```

This not efficient programming, it is is a passively aggressive word salad! The question that someone forgot to ask in this project was whether data is written more frequently than it is read. Remember also that frequency in this case means both number of times per time unit as well as number of places in the code-base. If the latter is more than "one and only one place", this relaxed attitude to up-front planning of data formats have created a maintainability bomb just waiting to explode.

But I digress.

The point is that *this* type of variety (i.e. that the same data source produce data entries of varying quality) can be handled at different times; at arrival time, at storage time, by some

background process, at retrieval time, or not at all. Each of these times have different costs in terms of efficiency, reliability, and maintainability, and an application designer need to balance these against each other. If arrival velocity is the most important requirement and the data cannot be cleaned e.g. on the client-side, then maintainability nightmares such as above are perhaps the only possible solution. However, I suspect that modern computers can do a lot more cleaning up on arrival time than one may think, and so I'd rather ensure that once the data hits the database it is well formed and all fields are used as they should. Part of my suspicion is that the cost of maintaining a system (probably across several decades) will in most cases outweigh any performance benefits.

Summary Variety as such is likely not a problem anymore: modern computers are able to process and store data at a scale that covers most "big data" use cases. The complication with variety stems from when data from many sources need to be collated in order to determine *what* to store in the first place. The challenge then becomes how and when to perform data maintenance in order to clean up and cull the existing data to keep the volume down and streamline future data management and maintenance.

We could summarise this to: *Data Variety is mostly a design and management problem, not a computing power problem.*

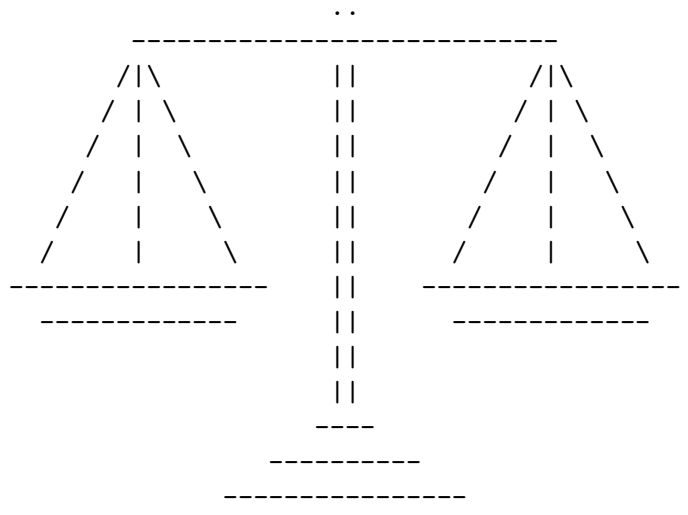
3.4 Value : The Cost of Noise

It is easy to confuse value with cost, and they are often seen as two sides of the same coin. So the first thing to realise is that value has *nothing* to do with the cost. What's more is that value is subjective. *Das Ding für Mich* is not the same as *das Ding für Euch* (as Immanuel Kant would have said if someone had given him the chance).

The claim, and the whole reason for the hype around Big Data, is that within these massive streams of data there are veins of gold just waiting to be mined by a resourceful company. So let's collect all the things and later on blame the developers for not finding the gold that you just know *must* be there. Why else would everyone be talking about Big Data? Eventually, and as with all buzzwords, simply stating that "*we work with big data!*" implies that *your* company are sitting on this vein of gold, and know how to tap into it, and often this is enough to increase the value of your company. Even if we, in the end, conclude that Big Data is indeed dead, there will still be pressure to claim that your company is working with Big Data.

But I digress. Again.

The real reason why value is included as one of the four 'V's of Big Data is because whatever value the data may have ought to outweigh all the costs related to Big Data analytics and Big Data Management.



Some would argue that just *having* data represent an untapped potential, especially if it is data about customers (or potential customers). Perhaps easier to understand is that the value of the data lies in what it *enables*. For example, collecting data about the health of a fleet of engines enable more apt maintenance, data may enable process improvements, sensor data may enable active interventions to save resources (e.g. cover up crops in anticipation of hail storms), or data about previous purchasing patterns may enable deeper market insights.

On the *cost* side, aside from the technical costs of collecting, storing, and analysing the data, there are also costs incurred to *curate* the data. Keeping the data updated with current events, fixing previous mistakes in the data, culling data that no longer have any value, adhering to legal requirements such as GDPR, maintaining data consistency, and keeping "proof of origin" of the data are all examples of activities that a data curator may work on.

Summary The question must always be whether the value outweighs the costs. To treat data as Big Data just to satisfy a marketing whim is one sure way to allow the costs to skyrocket. So be sure that you actually *have* to "go Big Data" before you do.

This is strike three for Big Data: *Data is not inherently valuable just because it is Big!* More analysis is needed to decide whether *you* are sitting on a potential goldmine or not.

3.5 Summary: From Big Data to Big Data Analytics

We've got three strikes against Big Data:

1. Volume is no longer an issue.
2. Lack of up-front planning doth not Big Data make.
3. Data is not inherently valuable just because it is Big!

Left in the room is *velocity*, which we declared as innocent to killing Big Data. Upon closer inspection, however, I would argue that the reason why velocity remains is not necessarily Big Data. It is, in fact, *Big Data Analytics*.

Whatever value one hopes to get out of the data is enabled through some sort of analysis, and this analysis may be more or less complicated to perform. Likewise, the speed and frequency at which the analysis needs to be made differs significantly from problem to problem – ranging from real time analysis to once-every-blue-moon analysis, and spanning everything from a simple query to a NP-complete regurgitation of the entire database.

Machine Learning and Artificial Intelligence is all the rage these days, and we all know that they require enormous training sets to become really good. Aren't they examples of Big Data? Well, they *can* be, even if there are plenty of smaller examples where you can download the entire dataset to your own laptop. Bear in mind also that *training* an artificial intelligence and *using* it are two separate activities. You may have all the time and resources in the world to train your AI, but will probably only need a fraction of the original dataset available for using the AI.

"Beautiful Evidence" by Edward Tufte is an interesting book about how to present data in graphics as well as in text, and how to do this while remaining true to actually available evidence (there is an entire chapter arguing about the dangers of PowerPoint). The following quote is apt with respect to machine learning:

> What the passive voice is for verbal reasoning, certain statistical methods are for data reasoning: anti-causal, a jumble of effects without causes. In particular, the techniques of data mining, factor analysis, and multi-dimensional scaling crunch and grind vast matrices down into small lumps but don't test causal models. These techniques are perhaps useful for those who have lots of data but no ideas. To be relevant for decisions and actions, whatever emerges from data crunching must somehow turn into evidence about causal processes.

"... Lots of data but no ideas." This pretty much sums it up, don't you think?

Later, Tufte reasons about the old adage "Correlation is not Causation", and concludes that the shortest accurate statement one can make about correlation and causation is that: *Empirically observed covariation is a necessary but not sufficient condition for causality.* When it comes to Big Data and much of the recent hype around machine learning, this is often forgotten. The fact that you have a lot of data does not mean that you are able to unveil any particular cause. You are certainly not going to do it just by using a machine learning algorithm; you need to construct an explanation model as well.

Given the way we often work with machine learning algorithms where we use one part of the data (after some magic hand-waving to fill in blank spots or deal with outliers) in order to train our algorithms in order to predict the remaining parts of the data, there may be thousands or millions of ways to fit a model, and the only measure we are certain of is that the model fits the rest of our data set. Once we unleash this model onto other data sets, we will likely find out that we have over-fitted the model to our one original data set.

And then what do we do?

3.6 Learning Outcomes

Relevant learning outcomes from the course syllabus are:

Knowledge and understanding On completion of the course, the student will be able to

- In depth be able to describe challenges with Big Data Analysis

Competence and skills On completion of the course, the student will be able to:

- Independently be able to set up a development environment consisting of local machine configurations and cloud based servers.
- Independently be able to implement and configure a big data analysis, including configuring the cloud platform and (if applicable) database.

Judgement and approach On completion of the course, the student will be able to:

- Be able to evaluate a problem description for a big data analysis and evaluate the potential to create a scalable cloud solution.

4 Quiz: Introduction to Big Data

TODO Please read the following articles:

1. Chen, Min, Shiwen Mao, and Yunhao Liu. "Big data: A survey." Mobile networks and applications 19 (2014): 171-209.
2. Fernández, Alberto, et al. "Big Data with Cloud Computing: an insight on the computing environment, MapReduce, and programming frameworks." Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 4.5 (2014): 380-409.
3. Curry, E. (2016). The Big Data Value Chain: Definitions, Concepts, and Theoretical Approaches. In: Cavanillas, J., Curry, E., Wahlster, W. (eds) New Horizons for a Data-Driven Economy. Springer, Cham.

TODO Summarise each article (no more than 1/2 page each) according to the following:

- Authors and Title of the article
- Briefly, what is the article about?
- What have they measured?
- What are their main findings?
- What can you learn from this article?

TODO Answer the following questions:

- Can you give examples of Big Data that you know of or have worked with? What makes them Big Data?
- Can you give examples of applications where the *volume* of data is an issue? Why, and what would you do to address these issues?
- Can you give examples of applications where data *velocity* is an issue? Why, and what would you do to address these issues?
- Can you give examples of applications where data *variety* is an issue? Why, and what would you do to address these issues?
- What is your opinion, is Big Data dead? Why or why not?

There is a Quiz on Canvas where you can submit your article summaries and answers to the questions.

NOTICE: *This quiz does not contribute to the grade in the course. We do, however, require of you to submit the quiz on time.* The purpose of this quiz is to serve as a learning aid allowing you to think about these concepts, and for us to keep track of your progress in the course. If you are unable to maintain the study pace required to submit this quiz on time, we want to be made aware of this so that you are able to re-plan your commitment for the remainder of the course.

5 Big Data Analytics in Practice: Problem Overview

The Qualitas Corpus (See `http://qualitascorpus.com/`) is a collection of software systems and their source code, created and maintained by Ewan Tempero, University of Auckland. It was originally created in order to have a curated collection of software systems to perform research studies on, but it is large enough to be useful for this course as well. As discussed in the introduction (See [introduction], page 3), it is perhaps not big enough to be called Big Data today, but it is still large enough to cause a bit of problems when trying to perform any form of analysis on it – as we shall see. For that reason, it does not matter that the most recent version of the corpus is getting a bit long in the tooth (the most recent version is from 2013): we can still use it for our analyses.

Using this dataset, we are going to go hunting for *Code Clones*. A code clone is a piece of code that is duplicated in one or more places within a set of source code files. Typically, code clones are created by copying pieces of code with little or no modifications in order to replicate some functionality. In general, there are four types of code clones that can be identified:

- Type-1: Identical code fragments except for variations in whitespace, layout, and comments.
- Type-2: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout, and comments.
- Type-3: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout, and comments.
- Type-4: Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

Obviously, the effort required to identify and match clones increases with the type, where type 1 clones can be detected using relatively simple string matching, whereas already type 2 clones require some knowledge about the compiled program's abstract syntax tree. Type 3 and type 4 clones require additional semantic understanding of what a code fragment performs.

Code clones, beside being examples of a bigger class of problems, are usually a code smell. In the simplest case they imply that if the code needs to be extended or bugs fixed, the developers need to find all places in the code where the change needs to be made. Such processes are usually error prone and mistakes are likely. Taking a step back, code clones imply that development resources are not spent as efficiently as possible, since obviously different developers (or the same developer) are working with the same task. Even further steps back, code clones can be used as a red flag that indicate that code has been copied from other places, e.g. StackOverflow, or code that may be covered under different licensing terms.

Keeping track of code clones is thus desirable. Once clones are identified, there are two main strategies on how to handle the problem:

- refactor the code such that the functionality is not duplicated
- maintain a registry of code clones that can be used to lookup clones when changes to the code are made

Please note that code clones are not *per se* a design flaw to avoid at all costs. As often in engineering, the decision on how to handle a problem depends on the particular situation. You might find the work by Kapser and Godfrey useful (Kapser, Cory J., and Michael W. Godfrey. "'Cloning considered harmful' considered harmful: patterns of cloning in software. Download Cloning considered harmful' considered harmful: patterns of cloning in software." Empirical Software Engineering 13.6 (2008): 645.) as they identify code clone patterns, describe their advantages/disadvantages, long term issues and provide advice on how to manage them.

Most Clone Detectors work according to the same algorithm, as outlined below:

1. Pre-processing: Remove uninteresting code, determine source and comparison units/granularities
2. Transformation: One or more extraction and/or transformation techniques are applied to the pre-processed code to obtain an intermediate representation of the code.
3. Match Detection: Transformed units (and/or metrics for those units) are compared to find similar source units.
4. Formatting: Locations of identified clones in the transformed units are mapped to the original code base by file location and line number.
5. Post-Processing and Filtering: Visualisation of clones and manual analysis to filter out false positives
6. Aggregation: Clone pairs are aggregated to form clone classes or families, in order to reduce the amount of data and facilitate analysis.

This process is the same for all types of clones, but the transformation and match detection can be made arbitrarily complex.

6 Task: Downloading and Describing the Qualitas Corpus

The first step is to ensure access to the Qualitas Corpus. For convenience, the corpus is available from an FTP server at BTH, but given the size we would rather not have you download it more often than you need to. Once you have downloaded it once, we am sure you will agree with us. Thus, we provide you with the basics to create a docker image with the sole purpose of downloading the Qualitas Corpus for you. Since the corpus should survive restarts of the docker container, we will be unpacking it in a separate docker volume, which we will keep and mount from different containers.

Tasks

1. Download the BigDataAnalytics repository:
   `git clone https://github.com/mickesv/BigDataAnalytics.git`
2. Build the '`CorpusGetter`' image (found in `Containers/CorpusGetter`) and tag it as `corpusgetter`
3. Create a volume for the Qualitas Corpus: `docker volume create qc-volume`
4. Run the `corpusgetter` image with the `qc-volume` mounted and instruct it to fetch the corpus: `docker run -it -v qc-volume:/QualitasCorpus --name qc-getter corpusgetter FETCH`
5. If all goes well, this should end by printing some statistics about the fetched corpus and then pause. If not, you can re-run the `corpusgetter` image without any commands, which will only print these statistics and then pause: `docker run -it -v qc-volume:/QualitasCorpus --name qc-getter corpusgetter` . Note that pressing any key will terminate the script and end the container, please do not do this yet.
6. In a separate terminal, enter the running container with `docker exec -it qc-getter bash` and browse around. Do spend some time to acquaint yourself with how the QualitasCorpus is structured, where to find the actual source code for the systems in it, and what else you can find for each system.

7 Quiz: Code Clones and Big Data

TODO Please read the following articles:

1. C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach Download Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," Science of Computer Programming, vol. 74, no. 7, pp. 470–495, 2009.
2. Kapser, Cory J., and Michael W. Godfrey. "'Cloning considered harmful' considered harmful: patterns of cloning in software. Download Cloning considered harmful' considered harmful: patterns of cloning in software." Empirical Software Engineering 13.6 (2008): 645.

TODO Summarise each article (no more than 1/2 page each) according to the following:

- Authors and Title of the article
- Briefly, what is the article about?
- What have they measured?
- What are their main findings?
- What can you learn from this article?

TODO Answer the following questions:

- Why is the Qualitas Corpus nearly 10GB in size, but only 1.5GB of these are Java files? What other types of files consume most space?

There is a Quiz on Canvas where you can submit your article summaries and answers to the questions.

NOTICE: *This quiz does not contribute to the grade in the course. We do, however, require of you to submit the quiz on time.* The purpose of this quiz is to serve as a learning aid allowing you to think about these concepts, and for us to keep track of your progress in the course. If you are unable to maintain the study pace required to submit this quiz on time, we want to be made aware of this so that you are able to re-plan your commitment for the remainder of the course.

8 Assignment Task: Working with a Stream of Data

The Code Stream Clone Detector consists of two parts, a *CodeStreamGenerator* and a *CodeStreamConsumer*. Below, a brief overview of each is provided.

CodeStreamGenerator The generator is just a placeholder for whencesoever new files may be submitted to the clone detector for inspection. This may, for example, be through a website where users submit suspicious code, or via a hook in the configuration management system that automatically send all commits for clone detection. The CodeStreamGenerator expects the previously generated `qc-volume` to be mounted under `/QualitasCorpus` so that it has some code to "generate". You could, of course, mount any code repository in this place, and CodeStreamGenerator would be none the wiser.

The source code for the CodeStreamGenerator is found under `Containers/CodeStreamGenerator/`. It consists of a Dockerfile, a bash script, and two test files that will be sent first if you give the `TEST` flag to the container. Please take a moment to study the Dockerfile and the bash script so you understand what they do.

CodeStreamConsumer The consumer is a `node.js / express.js` app, the source code of which is found under `Containers/CodeStreamConsumer/` , that waits for a HTTP '`POST`' request with a file in it. Each file is processed in turn using the `processFile()` function, clones are detected, and statistics are collected. Opening a web browser to `http://localhost:8080` (Assuming you've instructed docker to forward port 8080 on the host to port 3000 in the CodeStreamConsumer container) displays the time it took to process the last file, the code clones found so far, and the names of the thus far processed files.

In the `src` directory, you will find a small set of files:

```
$ tree
.
 CloneDetector.js # This is the actual Clone Detector, discussed further below
 Clone.js         # The Clone class represents one identified Clone
 CloneStorage.js  # A Singleton Class to keep track of already identified Clones
 FileStorage.js   # A Singleton Class to keep track of already processed files
 index.js         # Start file for the system.
                     # Sets up the express app and contains
                     # the ~processFile()~ function.
 SourceLine.js    # The SourceLine class represent a single line
                     # (including its original line number) in a file.
 Timer.js         # This class is used to start and stop timers as needed.
```

The `processFiles()` function follows the previously outlined overall process for code detection, with pre-processing, transformation, match detection, post-processing, and storage. Primarily, it is the `CloneDetector` class that implements these steps.

In the CodeStreamConsumer a `file` is the central concept through all of these steps. Each processing step may add new fields to the `file` object. Just before storage, it will contain the following fields:

```
file.name       // Full path and name of the original file.
file.contents   // Original contents as a single string.
file.lines      // An array of SourceLines.
                // These are generated by CloneDetector::#filterLines(),
                // which replaces empty lines and comment lines with ''
                // so they can be easily cleared later.
file.chunks     // An array of Chunks, which are just arrays of
                // SourceLines, each having the length CHUNKSIZE.
```

```
file.instances  // An array of Clones. The processing steps will
                // refine this list so that only true and
                // as-long-as-possible Clones remain.
```

Tasks With these introductions out of the way, it is time to actually fix and run the application.

0. Generate images for the generator and the consumer, tag them `csgenerator` and `csconsumer`, respectively.
1. Start the application using `docker compose` with the given `stream-of-code.yaml` . You may wish to modify the `DELAY` to something bigger during your first tests so you can see what is going on. You will also notice that the `cs-consumer` container introduces a bind-mount to the source code of the app so that you can edit the code locally and it will restart automatically.
2. Study the source code of the CodeStreamConsumer so that you understand it. You will notice that there are two implementation tasks, identified by '`TODO`' statements in the code.
3. The first task is to complete the implementation of the `CloneDetector`. The smallest possible edit for you is to implement the three methods as suggested, but feel free to be more creative here.
4. The second task is to add a new landing page in `index.js` for more detailed timing statistics (not just data on the last processed file). Your own creativity sets the boundaries here, but it is useful to be able to see trends over time for how long it takes to process each new file. You may also wish to normalise the processing time for each file with the number of lines it contains. Make use of the test files `A.java` and `B.java` to see if you are able to correctly identify clones (feel free to write a new CodeStreamGenerator that just send these two files on repeat).
5. Once you are satisfied with your implementation, run the app for a longer period of time so you can get some proper statistics out of it. Make note of what happens, when it happens, and try to reason about causes and remedies.

Summary We have now:

1. Created and used a docker image/container to fetch a very-nearly-Big-Data repository of source codes, i.e. the Qualitas Corpus.
2. Stored the QualitasCorpus in a docker volume, `qc-volume`, for use in subsequent steps
3. Created a docker image to simulate source code being generated in a continuous stream.
4. Created a node.js app to consume this stream and look for code clones in any of the previously submitted files.

9 Assignment Submission: Code Stream Clone Detector

Please answer the following questions:

1. Are you able to process the entire Qualitas Corpus? If not, what are the main issues (think in terms of data processing and storage) that causes the CodeStreamConsumer to hang? How can you modify the application to avoid these issues?
2. Comparing two chunks implies `CHUNKSIZE` comparisons of individual SourceLines. What can be done to reduce the number of comparisons?
3. Studying the time it takes to process each file, do you see any trends as the number of already processed files grow? What may be the reasons for these trends (think in terms of the data processing algorithms)?
4. Please upload your code for the CodeStreamConsumer (zip the folder for the whole Container).

Once delivered, *a meeting will be scheduled* where you are expected to run and explain your application and its deployment. Be prepared to answer questions about:

- Your implementation of the CloneDetector
- Your implementation's ability to accurately identify clones.
- Timing statistics and what they may imply
- Possible future improvements

There is a Quiz on Canvas where you can submit your article summaries and answers to the questions.

NOTICE: This is a *marked* assignment that contributes to the grade on the course. Specifically, the assignment contribute to your grade on the following parts of the course:

- Big Data Analytics

10 Improving the Clone Detecting Algorithm

The CodeStreamConsumer suffers from a few issues that makes it less suitable for Big Data. The most obvious issue is of course that it hangs (at least for me) before it finishes processing all the files even in the relatively small Qualitas Corpus. This really isn't very surprising, considering that the `FileStorage` class keeps all the processed files in memory. Not only that, to speed up processing it keeps a chunkified version of the file *also* in memory, essentially doubling the memory footprint of every file.

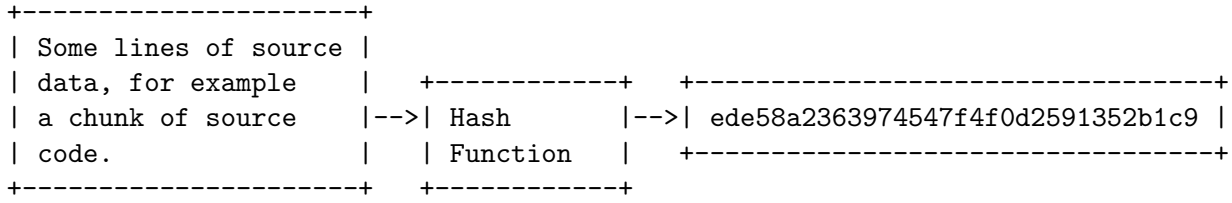
In fact, most of the time the original file is not needed once the initial processing is done, so we can and should offload all of that into a database. Studying the `FileStorage` class, it can easily be seen how this might be done.

I leave it as an exercise to the avid student to implement and measure what retrieving every file from the database each time a new file arrives for processing will do for the processing times. A middle ground would be to store the original file but keep the chunks in memory. This would at least reduce the memory footprint down to just one instance of each file. This is an improvement, but still not very Big Data of us. We can do better.

10.1 Hashing

A seemingly unrelated issue is that every chunk comparison requires `CHUNKSIZE` comparisons of SourceLines. As it turns out, the next improvement solves both of these problems, i.e. fewer comparisons and smaller memory footprint. One approach to reduce complex data to a small fixed-size comparable unit is through *hashing*. For example the music identification service Shahzam `https://www.shazam.com/` uses hashing to store and match music (Nicely explained in a Youtube video *How Shahzam Works*: `https://youtu.be/kMNSAhsyiDg`).

The trick with hashing (which, by the way, the `String.hashCode()` method in Java fails at) is that for every unique input the hash function produces a single unique fingerprint. Thus, the same input always produce the same output, and no two inputs (which are different) produce the same output. So rather than having to compare data of some arbitrary size (`CHUNKSIZE` of source code, `x` seconds of noise, or the DNA of an extinct reptile), it is enough to compare the generated hash code to determine if the original data is the same.



Learn more about hash functions:

- Hashing: Why & How `https://www.youtube.com/embed/yXmNmckX4sI`
- Hash Tables and Hash Functions `https://www.youtube.com/embed/KyUTuwz_b7Q`
- Hashing: The Greatest Idea in Programming `https://www.i-programmer.info/babbages-bag/479-hashing.html`

10.2 $O(n^2)$

In the current implementation, every file, indeed *every chunk*, is compared to every other file and chunk. This is the equivalent of using bubble sort, which is never a good idea and absolutely positively definitely never ever *ever* a good idea if you are planning to process Big Data. With hash functions we have introduced the idea of a "fingerprint", and the logical next step is to keep a list of all fingerprints *and where they came from.* If the same fingerprint have two (or more) origins, it is a clone, which is what was to be shown. The data structure, for you algorithms and

data structure nerds out there, is a *Dictionary*, where the key is the hash value of the chunk of code and the values are the locations where this key can be found.

```
    Key        - Values
|
+-- AEF623...  - File GH/line 4-8
|
+-- 1ED322...  - File AK/line 25-29, File AK/line 48-52
|
+-- GJHT55...  - File OB/line 12-16
|
+-- MH6555...  - File NJ/line 56-60
|
+-- 7878GF...  - File KI/line 4-9, File KI/line 88-92, File OO/line 4-9
|
+-- 0D21HG...  - File MK/line 89-93
|
```

The overall process for each file now becomes:

1. Generate chunks and transform each chunk into a hash value.
2. Add the tuple `<hash value, file location>` to the Dictionary Of All Chunks
3. Filter the Dictionary Of All Chunks so that only entries with more than one value remains. These are candidate Clones.
4. For each remaining `file location`, find all adjacent file locations and expand the candidate Clones accordingly.
5. Repeat until no more adjacent file locations. What remains are unique and expanded Clones.
6. For each Clone, use the `file location` to extract the original source code for the clone; add this to the Clone.
7. Add to list of identified Clones.
8. Repeat.

There is an opportunity to optimise in steps 2 and 3 so that we keep track of only the currently processed file. Otherwise, the expand function in step 4 will take much longer.

The implementation of hash tables such as this are usually quite optimised for finding elements (or where to write a new element), bringing these operations down to a best case Big-O complexity of `O(log n)` (which assumes that all your trees are balanced and the wind is blowing in the right direction. The bottom line is that we can't really improve the *writing* any further).

10.3 From File-by-File to All-at-Once Analysis

The elephant in the room remains. We are still processing one file at the time, so the overall complexity is still stuck to the bubble-sort `O(n^2)` quagmire. This is kind of the whole point with this exercise: In order to get any further, we need to re-think the problem. The *file* is no longer the most appropriate data "atom", even if we still have to start (and end) there. The algorithm we are looking for now would be to:

1. For All Files At Once, generate chunks and transform each chunk into a hash value.
2. For All Chunks (now only a hash value and a location), Add the tuple `<hash value, file location>` to the Dictionary Of All Chunks.
3. Filter the Dictionary Of All Chunks so that only entries with more than one value remains. These are candidate Clones.

4. For each remaining `file location`, find all adjacent file locations and expand the candidate Clones accordingly.
5. Repeat until no more adjacent file locations. What remains are unique and expanded Clones.
6. For each Clone, use the `file location` to extract the original source code for the clone; add this to the Clone.
7. Add to list of identified Clones.

It turns out we were almost there! Steps 1 and 2 are modified to work on all files at once, and the expand in step 4 remains a challenge.

Map and Reduce Before continuing, we need to introduce the functional programming concept of MapReduce (Also mentioned as a Cloud Architecture Pattern in the Cloud Provisioning and Orchestration part of the course). The basic trick is to stop thinking about iterations and procedures where you *first* do something and *then* do something else. Instead, with functional programming you spend more time describing what the input data looks like, and how you want the output data to look like. Once you know this, you write a function to get from the input data to the output data and *map* it onto your entire collection.

- `map()` applies a given function element by element to a collection of elements and returns a collection of new (or modified) elements.
- `reduce()` aggregates the list of results from `map()` into a single result.

These two simple principles (coupled with a bit of `filter()`, `find()` , `exists()` , etc.) are surprisingly versatile for data analysis problems. Sometimes (as with the clone detection algorithm discussed so far), the problem or the required input and output data formats need to be rephrased a bit, but once you do the world's your oyster. Even better, `map()` can easily be parallelised since each element should be transformable by itself. The `reduce()` can also be parallelised, but this requires a bit more work. Being so very versatile, it is no surprise that you will find implementations of MapReduce in places such as databases and in many programming languages (such as lisp, of course, but also Java 8's Streams API `http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/`, and in JavaScript `https://www.sitepoint.com/map-reduce-functional-javascript/`).

Let's try to turn the previous algorithm into something MapReduceable. While we are at it, we also need to acknowledge that not all things should be solved by *our* programming. Some tasks are better left to a database.

Map	Transform	
	Input	Array of files
	Output	Array of Chunks containing `{filename, startLine, endLine, contentHash}`
	Comment	Technically, the output is an Array of Arrays of Chunks, but this is quickly flattened into an Array of Chunks.
	Side Effect	Files is stored somewhere accessible (e.g. a database)
Reduce	Merge output from Transform map	
	Input	Array(s) of Chunks
	Output	Dictionary of `<contentHash , Array of Chunks>`
	Comment	Just storing the chunks to a database will take care of this step for us.

Filter	Match Detection	
	Input	Dictionary of `<contentHash, Array of Chunks>`
	Output	Dictionary of `<contentHash, Array of Chunks>` where Array of Chunks has length > 1.
	Comment	This is a simple database search.
Reduce	Expand	
	Input	Dictionary of `<contentHash, Array of Chunks>` where Array of Chunks has length > 1.
	Output	Array of Clones containing `Array of cloneInstances {filename, startLine, endLine}`
	Comment	As we will see, this is the real killer. We will study two ways of addressing this step.
Map	Format	
	Input	Array of Clones AND Access to files
	Output	Array of Clones containing `{ Array of cloneInstances, sourceCode }`
	Comment	This is strictly not needed; the clones are already identified by the previous reduce, but can be solved with a database join.
Reduce	Merge output from Format map	
	Input	Array(s) of Clones
	Output	Array of Clones
	Comment	Only needed if the previous map is done in parallel.

A comment on the '`Filter`' Step: If we do decide to implement this ourselves instead of relying on a database, this can be done as a Map (returning null if `Array.length == 1`) or (better but less parallelisable) as a Reduce.

When parallelising a `map()` operation, the input is typically split into several parts, each of which is executed by a separate computing unit. In the extreme case there is one computing unit per entry in the input (e.g. when using Azure Functions or Amazon Lambdas). However, to minimise overhead costs of distributing data, running a function, and collecting the results, it may be desirable to instead distribute a block of data to each computing unit. With many frameworks, optimising the '`split size`' is an important step of fine-tuning the performance of a big data analysis.

With several computing units, each executing their own `map()` on a section of the data, it becomes evident that the outputs from each computing unit need to be merged. Not all MapReduce frameworks leave the data in a nice state; expect to see a series of Arrays simply concatenated together even if the next step requires the input data to be at least sorted.

10.4 Wisely, and slow. They stumble that run fast.

Shakespeare be damned, we *do* want this to run fast. And in parallel. And be distributable. So which programming language should we choose for this?

Until now, JavaScript have served us well. Map and Reduce are part of the standard feature set for JavaScript, and with Promises it is also possible to run the `map()` in parallel, but not distributed (granted, there are some open source packages for distributed MapReduce, but most

of the ones I could find were last modified around 2018). Moreover, the general consensus of The Internet is that one does not reach for JavaScript when building a high performance computing solution. So let's explore the options.

We *could* go down the rabbit hole and implement everything in C or C++ (`std::transform` and `std::reduce` were added in C++17), but I am not in the mood to go hunting for memory leaks, and distribution remains an issue.

Java? With Java 8 and the Streams API, there is a native implementation of MapReduce, as well as support for working with infinite data streams. Writing in Java would also have the benefit of opening up to many of the Apache Software Foundation's data processing tools such as Hadoop `https://hadoop.apache.org/`, Spark `https://spark.apache.org/`, or Storm `https://storm.apache.org/index.html`.

The downside with all three of these, in my humble opinion, is that none of them are ready to be released outside of a university lab. Setting them up and actually getting the master node and worker nodes to communicate is *painful*. There is no reliable list of which network ports that need to be open on each type of node in the network, the used network ports change between versions (even minor versions), and by default you need to essentially keep all ports above 10000 open (and hope that this is sufficient; the port to use for a particular job is assigned at random and sometimes you get a port number lower than 10000). Older versions communicated via unencrypted http requests without any authentication. And even when you do have everything up and running, Hadoop MapReduce will fail three times out of ten (I know: I have tested).

However.

The only other reasonable distributed computing alternative I have found is Disco `http://discoproject.org/`. And then you have Python on your hands.

So Java it is. Or Clojure, since I feel adventurous. The benefit of using Clojure for this is that infinite data streams are inherent to the programming language instead of added as an afterthought, so the program code can focus on the problem at hand rather than juggling data to fit the right Java Enterprise Object model. And before you start complaining about your lack of experience in Clojure, consider the following "Master Foo and the Recruiter" Koan (more Unix Koans (some of which are hilariously outdated) can be found at: `http://catb.org/~esr/writings/unix-koans/`):

> A technical recruiter, having discovered that that the ways of Unix hackers were strange to him, sought an audience with Master Foo to learn more about the Way. Master Foo met the recruiter in the HR offices of a large firm.
>
> The recruiter said, "I have observed that Unix hackers scowl or become annoyed when I ask them how many years of experience they have in a new programming language. Why is this so?"
>
> Master Foo stood, and began to pace across the office floor. The recruiter was puzzled, and asked "What are you doing?"
>
> "I am learning to walk," replied Master Foo.
>
> "I saw you walk through that door" the recruiter exclaimed, "and you are not stumbling over your own feet. Obviously you already know how to walk."
>
> "Yes, but this floor is new to me." replied Master Foo.

Upon hearing this, the recruiter was enlightened.

11 cljDetector: All-at-Once Clone Detection

In this section we go through the implementation for our All-at-once Clone Detector. Please do spend some time to read this section and the included discussion of various design decisions. More importantly, do spend some time with the source code to acquaint yourself with it. Understanding this implementation helps you understand a key component of Big Data Processing: how to work with seemingly infinite data streams. While the overall clone detection algorithm has been described earlier, the same map/reduce concepts are applied at every step along the way to filter and transform data streams by treating each individual item in isolation (as much as possible; there are occasions where some form of state need to be kept, and the implementation introduces a few different solutions for these occasions).

In `Containers/cljDetector` you will find the All-at-once clone detector. To start this on the full QualitasCorpus, you may use the `all-at-once.yaml` file in the root directory as well. The source code for the program is found in `Containers/cljDetector/src/cljdetector`:

```
$ tree
.
 core.clj
 process
    expander.clj
    source_processor.clj
 storage
     storage.clj
```

where `core.clj` is the starting point (containing the `-main` function), and the other three files are packages dealing with (as their names imply) storage, processing of source files, and clone expansion. The `-main` function reads as follows:

```
(defn -main
  "Starting Point for All-At-Once Clone Detection
  Arguments:
   - Clear clears the database
   - NoRead do not read the files again
   - NoCloneID do not detect clones
   - List print a list of all clones"
  [& args]

  (maybe-clear-db args)
  (maybe-read-files args)
  (maybe-detect-clones args)
  (maybe-list-clones args)
  (ts-println "Summary")
  (storage/print-statistics))
```

which corresponds to the previously discussed MapReduce chain as follows:

- `(maybe-clear-db args)` No correspondence, this is just there to optionally clear the database from previous attempts.
- `(maybe-read-files args)` This is the first map "Transform" and the first reduce "Merge output from Transform map".
- `(maybe-detect-clones args)` The Filter "Match Detection" and the "Expand" reduce.
- `(maybe-list-clones args)` The final "Format" map to join clones with their original files.

So far, so good. Let's look at the `maybe-read-files` function next (since the others are shaped in more or less the same way this is as good an example as any):

```
(defn maybe-read-files [args]
```

```
(when-not (some #{"NOREAD"} (map string/upper-case args))
  (ts-println "Reading and Processing files...")
  (let [chunk-param (System/getenv "CHUNKSIZE")
        chunk-size (if chunk-param (Integer/parseInt chunk-param) DEFAULT-CHUNKSIZE)
        file-handles (source-processor/traverse-directory source-dir source-type)
        chunks (source-processor/chunkify chunk-size file-handles)]
    (ts-println "Storing files...")
    (storage/store-files! file-handles)
    (ts-println "Storing chunks of size" chunk-size "...")
    (storage/store-chunks! chunks))))
```

If the list of command line arguments `args` , when transformed into upper-case, does *not* contain the word "NOREAD", define some local variables (which will only be available inside the `let` statement):

- `chunk-param` is just a step along the way to get a usable `chunk-size`
- `file-handles` is a set of all files, i.e. the result of calling `source-processor/traverse-directory` . This is a lazy sequence, so no handles are actually created until needed (which will be in the call to `source-processor/chunkify` or maybe even in the `storage/store-files!` call).
- `chunks` is a set of all chunks for all files. This is also a lazy sequence, so it will not be evaluated until it needs to be (in the call to `storage/store-chunks!`). By convention, function names ending in an exclamation mark have side effects, in the case of `storage/store-chunks!` this side-effect is the storing of the results to the database.

Once the variables are defined, most of the processing have (at least nominally) been done and what remains is to store the results to the database. I say "nominally" because in fact none of the actual processing has happened yet; because of lazy sequences, the processing only happens in these last two calls to `storage/store-files!` and `storage/store-chunks!`.

The final piece of this function are the interspersed calls to our `ts-println` function to print timestamped status messages, and a whole lot of closing parentheses to back out of the `let`, the `when-not`, and the `defn` expressions.

11.1 First Run on full Qualitas Corpus

The results of running the first version of this program on the full Qualitas Corpus yields the following data on my laptop:

Processing Step	**Database Collection**	**#Items**	**Processing Time**
Reading and Processing Files			0 min
- Storing Files	files	132000	1 min
- Storing Chunks	chunks (size 20)	$18 * 10^6$	50 min
Identifying Clone Candidates	candidates	586000	15 min
Expanding Candidates			fails after +2h
- Storing Clones	clones		

Yes, I ran this on my not at all very super impressive laptop that was still able to manage without any permanent damage other than to the CPU fan. I mention this just to forestall any arguments that your computer is too tiny to cope with big data.

An observation when correlating this data with the processes running on the computer during each step is that the database is using all available CPUs whereas Clojure (and its execution as a java virtual machine) is limited to two main processes. One of which I strongly suspect is the garbage collector. For cloud deployment, this implies that the database node will benefit from having multiple CPUs whereas the cljDetector can cope with a much smaller node. It also means that the database is most likely able to cope with the increased load that any parallelisation of

the cljDetector can incur, and that it is worth studying ways of parallelising at least the 'map' steps in the cljDetector.

Storing the files is quick, I suspect that Clojure can do a lot to help out with simple data transfers behind the scenes. Storing chunks is much slower, and I willingly grant that the source-processor/chunkify-file function can probably be much improved. The real improvement would however be to reduce the number of chunks per file, and one might be led to believe that the CHUNKSIZE parameter may help with this. However, since there is essentially one chunk created for each line in each file, the only difference is at the end of the file where you may create CHUNKSIZE fewer chunks per file.

```
Chunk 1: File lines 1 -- 1 + CHUNKSIZE
Chunk 2: File lines 2 -- 2 + CHUNKSIZE
@dots{}
Chunk n: File lines n -- n + CHUNKSIZE
Chunk n+1 : File lines n+1 -- n+1 + CHUNKSIZE
```

Thus, CHUNKSIZE only determines the size of the smallest possible clone. What *may* be impacted by the CHUNKSIZE is the hash encoding. I leave it as an exercise to the reader to investigate the effect of MD5-encoding chunks of different sizes.

However, the real killer is the "Expanding Candidates" step – not only because it fails to complete, but because it takes a long time to get to the point where it fails. This is discussed further below (see [elegant-expansion], page 30, and [beer-expansion], page 31,).

11.2 Expanding Elegantly but Inefficiently

The *elegant* way to expand clones is by using a reduce operation like so:

```
(defn expand-clones-elegantly [candidates]
  (reduce (fn [clones candidate]
            (let [{overlapping true remaining false} (group-by
                                                      #(overlaps? candidate %)
                                                      clones)]
              (if (empty? overlapping)
                  (conj clones candidate)
                  (conj remaining (reduce merge-clones candidate overlapping)))
            )) [] candidates))
```

In plaintext: with every candidate, split the already found clones into two piles (This is what the group-by does; One that overlaps the new candidate (all elements where overlaps? evaluates to true) and one with all the remaining clones (where overlaps? evaluates to false). If there are no overlapping clones, this is a new clone and is conjoined to the existing list of clones. Otherwise, we want to remove the overlapping clones (conveniently already done with the remaining list) and just store a new clone that merges all of the overlapping clones into one. Beautiful, isn't it? This is the version that fails above.

As it turns out, this introduces a performance antipattern called "The Ramp". Every new candidate that does not overlap any existing clones will add one more item to the list of clones. Sometimes a few items are removed, but most of the time the list of clones just grow. The time it takes to group-by thus gets longer and longer all the time like a ramp: it starts out small but grows continually. Eventually something has to break:

> Execution error (MongoCursorNotFoundException) at com.mongodb.operation.QueryHelper/translateCommandException (QueryHelper.java:27). bigdataanalytics-clone-detector-1 | Query failed with error code -5 and error message 'Cursor 7063045859606288607 not found on server dbstorage:27017' on server dbstorage:27017

Why does the *database* throw an error in the `expand-clones` method? There are no database calls there. Hint: whenever possible Clojure tries to use *lazy collections*, meaning they are not generated straight off but only on an as-requested basis. This is part of why and how it is possible to work with infinite data series. The other clue is that the database connection defaults to recycling open cursors after 10 minutes, if a new candidate is not requested in more than 10 minutes the database is going to kill the connection. We could force Clojure's hand and wrap the call to `storage/identify-candidates!` in a `doall` block, making sure that all candidates are in memory before we start the `expand`. However, we would still have the ramp antipattern in there, so the time per candidate is still going to grow and ultimately something else is going to break for us if we *do* try to throw Real Big Data (TM) onto this algorithm. Most likely our sanity will be the first to go.

Just to prove that there is nothing technically *wrong* with this algorithm, here is the execution data for only a part of the QualitasCorpus, i.e. only the Eclipse SDK:

Processing Step	**Database Collection**	**#Items**	**Processing Time**
Reading and Processing Files			0 min
- Storing Files	files	22634	< 1 min
- Storing Chunks	chunks (size 20)	$3.3 * 10^6$	10 min
Identifying Clone Candidates	candidates	39689	1 min
Expanding Candidates			5 min
- Storing Clones	clones	2515	< 1 min

So there. :-P

11.3 Expand with "Aleph-null Bottles of Beer"

The alternative to being elegant is inspired by an ancient song (who knew that *this* was Billy Bones' true treasure!):

> Lots and lots of candidates on the wall!
> Take one down,
> stretch it abound,
> store it as a clone,
> and remove what we found.

Translate this to Clojure, and we get the following:

```
(defn maybe-expand [dbconnection candidate]
  (loop [overlapping (storage/get-overlapping-candidates dbconnection candidate)
         clone candidate]
    (if (empty? overlapping)
      (do
        (storage/remove-overlapping-candidates! dbconnection (list candidate))
        clone)
      (let [merged-clone (reduce merge-clones clone overlapping)]
        (storage/remove-overlapping-candidates! dbconnection overlapping)
        (recur (storage/get-overlapping-candidates dbconnection merged-clone)
               merged-clone)))))

(defn expand-clones []
  (let [dbconnection (storage/get-dbconnection)]
    (loop [candidate (storage/get-one-candidate dbconnection)]
      (when candidate
        (storage/store-clone! dbconnection (maybe-expand dbconnection candidate))
        (recur (storage/get-one-candidate dbconnection))))))
```

Where:

- In `expand-clones` we "take one down" as long as we can find one. `maybe-expand` "stretches it abound" and then we store it as a clone.
- In `maybe-expand` we find any candidates that overlap and stretch our clone to encompass these too (in `merged-clone`). Removing all the "used" candidates from the database as we go, we then loop back to see if the newly stretched clone happen to overlap with any new candidates.
- The two clojure commands `loop` and `recur` go hand in hand to overcome the principle of immutable data by replacing iterations with recursion.

The Execution time for the '`Expanding Candidates`' step with Eclipse SDK is now down to slightly over two minutes. *me adjusts his tie and brushes off an imaginary speck of dust from his sleeve.*

```
 ///   +-------------------------------------------------------------------------+
 o o  <  I tried to crawl through the code and get an approximation of a mental |
  -    | map of what is going on in the code. I think I get the FP (elegant)    |
       | expander but you lost me with the beer. I understand the ramping       |
       | problem, but how is that actually solved? I can see that beer moved    |
       | the overlap detection from clojure space to a mongo query (in          |
       | get-overlapping-candidates ) and the clones are stored in the database |
       | instead of in memory. But is that the actual core of the solution, or  |
       | where is it?                                                           |
       +-------------------------------------------------------------------------+

  _
_/_\_  +-------------------------------------------------------------------------+
 o o  <  There are a number of factors that help us out here. First is that     |
  v    | only one candidate (and possible expansions) is kept in memory. Second |
       | is that we shift processing from clojure to the database, so the       |
       | database connection does not time out as easily.                       |
       |                                                                        |
       | But the biggest factor is, I think, that the bundle of joy             |
       | get-overlapping-candidates is essentially a series of filters (which   |
       | databases are really good at) where already the first $match quickly   |
       | reduces the number of elements to continue looking at.                 |
       |                                                                        |
       | Also, for every successfully expanded clone the set of candidates      |
       | shrinks, whereas in the "elegant" solution we just get more and more   |
       | clones to match every new candidate with. If the database can execute  |
       | the query for the first clone candidate (and why shouldn't it, it is   |
       | after all just a set of filters), life is only going to get easier for |
       | every subsequent clone candidate.                                      |
       +-------------------------------------------------------------------------+
```

11.4 Expand with Aggregation Pipeline

A third option, which we will not explore further, is to use an *aggregation pipeline* in MongoDB to do the expansion. We are already using an aggregation pipeline for parts of the expansion, but not for all of it:

```
(defn get-overlapping-candidates [conn candidate]
  (let [db (mg/get-db conn dbname)
        collname "candidates"
        clj-cand (from-db-object candidate true)]
```

```
(mc/aggregate db collname
              [{$match
                {"instances.fileName"
                 {$all (map #(:fileName %) (:instances clj-cand))}}}
               {$addFields {:candidate candidate}}
               {$unwind "$instances"}
               {$project
                {:matches
                 {$filter
                  {:input "$candidate.instances"
                   :cond
                   {$and
                    [{$eq
                      ["$$this.fileName" "$instances.fileName"]}
                     {$or
                      [{$and
                        [{$gt ["$$this.startLine" "$instances.startLine"]}
                         {$lte ["$$this.startLine" "$instances.endLine"]}]}
                       {$and
                        [{$gt ["$instances.startLine" "$$this.startLine"]}
                         {$lte ["$instances.startLine" "$$this.endLine"]}
                         ]}]}]}}}
                 :instances 1
                 :numberOfInstances 1
                 :candidate 1
                 }}
               {$match {$expr {$gt [{$size "$matches"} 0]}}}
               {$group {:_id "$_id"
                        :candidate {$first "$candidate"}
                        :numberOfInstances {$max "$numberOfInstances"}
                        :instances {$push "$instances"}}}
               {$match {$expr
                        {$eq
                         [{$size "$candidate.instances"} "$numberOfInstances"]}}}
               {$project {:_id 1 :numberOfInstances 1 :instances 1}}]))
```

An aggregation pipeline is (these days) MongoDB's preferred way of doing map/reduce-like processing. How to read this is as a series of processing steps, e.g `{$match}` , `{$addFields}`, `{$unwind}`, `{$project}`, and `{$group}` where each step filters or transforms the data stream in some way. There is much to say about what can and can't be done with an aggregation pipeline, a good starting point is the official documentation: `https://www.mongodb.com/docs/manual/aggregation/`. Eventually it boils down to *as long as the problem is defined in the right way, the data processing step becomes "easy"*.

This option will not be explored further in this course, but it does deserve mentioning. Any data processing we can offload to the database is one thing less for our application to worry about, the database is excellent at juggling data streams, and has built-in support for parallelism (and even horizontal scaling).

If someone more well versed in the ins and outs of database wrangling than yours truly is willing to attempt this path, I shall be more than happy to update this document with how you did it.

```
 ----------------------------
( SELECT COUNT(id) FROM Sheep )
```

```
----------------------------
       o   ^__^
        o  (oo)\_______
           (__)\       )\/\
               ||----w |
               ||     ||
```

11.5 Second Run on full Qualitas Corpus

Using the "Beer Expansion Algorithm" on the full Qualitas Corpus yields the following numbers:

Processing Step	**Database Collection**	**#Items**	**Processing Time**
Reading and Processing Files			0 min
- Storing Files	files	$\simeq$ 132000	1 min
- Storing Chunks	chunks (size 20)	$\simeq 18 * 10^6$	~1h
Identifying Clone Candidates	candidates	$\simeq$ 586100	20 min
Expanding Candidates			
- Storing Clones	clones	$\simeq$ 25000	6h 20 min

Notice 1: I don't *think* it matters much since I had CPUs to spare but I had to participate in a Zoom call during the candidate expansion step, which may have slowed things down for me. To remember when it is time for you to run this exercise is to plan it so that you do not have to use the computer yourself at the same time.

Notice 2: During the expand-step it is now MongoDB processes that are consuming most CPU resources, but only on two CPUs. The Beer Expansion Algorithm has thus managed to push the workload to the database, but is still limited by how many and how quickly requests can be dispatched from the Clojure/Java processes. The need to parallelise the expansion algorithm thus remains. In fact, the expansion algorithm is already parallelisable by adding a second `reduce` step that consolidates any overlapping clones into one, much in the same way it is done for candidates today.

12 Assignment Task: Implementing Monitoring

It is time for you to contribute to this project. As it stands, the cljDetector prints status updates together with timestamps to the terminal (ignore, if you please, that the timestamps are usually off by some hours; you can fix this with the Dockerfile but it is not necessary).

However, it does not provide any way to monitor the progress within each processing step, which processes that are currently consuming most CPU, or any fine-grained statistics such as the time to create a chunk or the time to expand each clone. Moreover, if cljDetector were to run at a cloud provider somewhere, it may not be possible to monitor terminal output either.

This is where you come in. Since we *have* the database, and since each processing step incrementally add data to this database, we can use this to monitor progress from a separate process. In fact, you can do this already now in a separate terminal by running `mongosh` on the database node and query the database directly. With a small modification, the status updates (which all use the `cljDetector.core/ts-println` method) can also be written to a collection in the database so that we have access to them too.

Tasks

1. Modify `cljDetector.core/ts-println` to also add the timestamp and message to the database with a call to a new method `storage/addUpdate!` . Implement the `addUpdate!` method in `storage.clj` so that messages are stored in the new collection `statusUpdates`.
2. Implement a separate Docker Container – in your own choice of programming language – that monitors the database and regularly updates with the current count of `[files chunks candidates clones]` , as well as any new `statusUpdates`. Call this Container `MonitorTool` .
3. Implement monitoring so that you regularly sample the `[files chunks candidates clones]` and calculate statistics of the processing time at each time interval. You want to be able to answer questions such as whether the processing time to create a new unit (e.g. a chunk or a clone) increases or decreases with the number of already processed units, and whether the processing time changes linearly or exponentially.
4. Visualising these collected statistics can be done as you please, e.g. text-only in a terminal or by implementing a tiny web app. Do note that identifying trends in processing times will be more challenging with text-only output; graphs can help here.
5. Modify `all-at-once.yaml` so that your container starts alongside with cljDetector and attaches to the database.
6. Run `all-at-once.yaml` on the full Qualitas Corpus and collect statistics through your MonitorTool.

Optional (not mandatory) Tasks

- Collect statistics from inside the cljDetector container to see which processes are consuming most CPU each time-frame. Correlate this with the execution stage and the rest of your collected statistics.
- Since the clone expansion works with the database it is possible to pause execution (at the right moment - just before a new candidate is fetched from storage for processing) and resume later. This requires at least (a) a mechanism for signalling the cljDetector process (e.g. through an open network socket), (b) modifications to `expander/expand-clones` to poll if a signal has been received before recurring with a new candidate, and (c) modifications to `core/-main` to check for a command line flag "Resume" that overrides the usual execution and jumps directly to calling `expander/expand-clones` again. In addition, adding timestamped status updates so that total execution times can still be calculated would be helpful.

TODO Please read the following articles:

1. C. U. Smith and L. G. Williams. Software performance antipatterns. In Proceedings of the 2nd international workshop on Software and performance, pages 127–136, 2000.
2. C. U. Smith and L. G. Williams. New software performance antipatterns: More ways to shoot yourself in the foot. In Computer Measurement Group Conference, pages 667–674, 2003.
3. C. U. Smith and L. G. Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In Computer Measurement Group Conference, pages 717–725, 2003.

TODO Summarise each article (no more than 1/2 page each) according to the following:

- Authors and Title of the article
- Briefly, what is the article about?
- What have they measured?
- What are their main findings?
- What can you learn from this article?

13 Assignment Submission: All at Once Clone Detector

TODO Answer the following questions:

1. Are you able to process the entire Qualitas Corpus? If not, what are the main issues (think in terms of data processing and storage) that causes the cljDetector to fail? How may you modify the application to avoid these issues?
2. Please upload a report of your collected statistics and your analysis thereof. The report shall contain summaries of the collected data, along with answers to the following questions. Please *discuss* your answers and try to find reasons for why you are getting these results. Are there, for example, any particular characteristica in way the analytics is undertaken or the particular implementation/choice of tools/database that may cause or influence these results and trends?
   - Is the time to generate chunks constant or does it vary with the number of already generated chunks?
   - Is the time to generate clone candidates constant or does it vary with the number of already generated candidates?
   - Is the time to expand clone candidates constant or does it vary with the number of already expanded clones or the number of remaining candidates?
   - What is the average clone size? How can this be used to predict progress during the expansion phase?
   - What is the average number of chunks per file? How can this be used to predict progress during the read/chunkify/identify candidates stages?
3. Please make the raw data of your statistics accessible (either as an appendix to your report or via a linked web page).
4. Please upload your code for the modified cljDetector and your MonitorTool (zip the whole folders with the Containers. Please remember to also include the yaml-file used to start your application). Please highlight any modifications you have made in cljDetector to make it easier for us to review your submission.

Once delivered, *a meeting will be scheduled* where you are expected to run and explain your application and its deployment. Be prepared to answer questions about:

- Your modifications to cljDetector
- Your implementation of your MonitorTool
- The collected statistics and what they may imply
- Possible future improvements

There is a Quiz on Canvas where you can submit your article summaries and answers to the questions.

NOTICE: This is a *marked* assignment that contributes to the grade on the course. Specifically, the assignment contribute to your grade on the following parts of the course:

- Big Data Analytics

14 Summary

Well, here we are; the end of this part of the course. Please revisit the Section 3.6 [Learning Outcomes], page 13, to verify whether you now meet them. Starting with the four V's of Big Data (Volume, Velocity, Variety, and Value), we have explored what these imply for how we may solve Big Data Analytics tasks. Indeed, we have also discussed whether Big Data really is that big anymore. The conclusion is (not surprisingly) "it depends".

As a concrete example, we introduce the analytics task of identifying code clones in a relatively large collection of source code (i.e., the Qualitas Corpus), and discuss what makes this task a Big Data Analytics task. Identifying code clones is a relevant industry challenge, but it is also a representative for a larger class of problems to identify matches between '`some data`' and '`some large repository of other data`'.

Specifically, we explore three of the four V's:

Variety Each file that is processed contain a mix of code and non-code (empty lines and comments), so we preprocess the files to remove any non-code from further analysis. The challenge here is to maintain the locations in the original file so that any found clones can be visualised as they were originally submitted. We focus on Type-I clones, but further variety challenges can be explored if we enhance the analysis to also work with Type-II (which may be possible with the help of an abstract syntax tree for each file) or Type-III clones (which is a completely different ballgame, and require semantic understanding as well as syntactic agnosticism).

Velocity In this scenario each file arrives separately as a stream of data. For example, this can be seen as simulating a code repository where each new file represent a new commit to the repository. We explore a solution that incrementally builds up a collection of existing files and clones and match every new file against what was previously available. The challenge here is to process each new file quickly, and so the data need to be represented in a way that enables this. Some performance antipatterns such as the Ramp are difficult to avoid here, so the focus shifts to mitigating the effects of any unavoidable antipatterns.

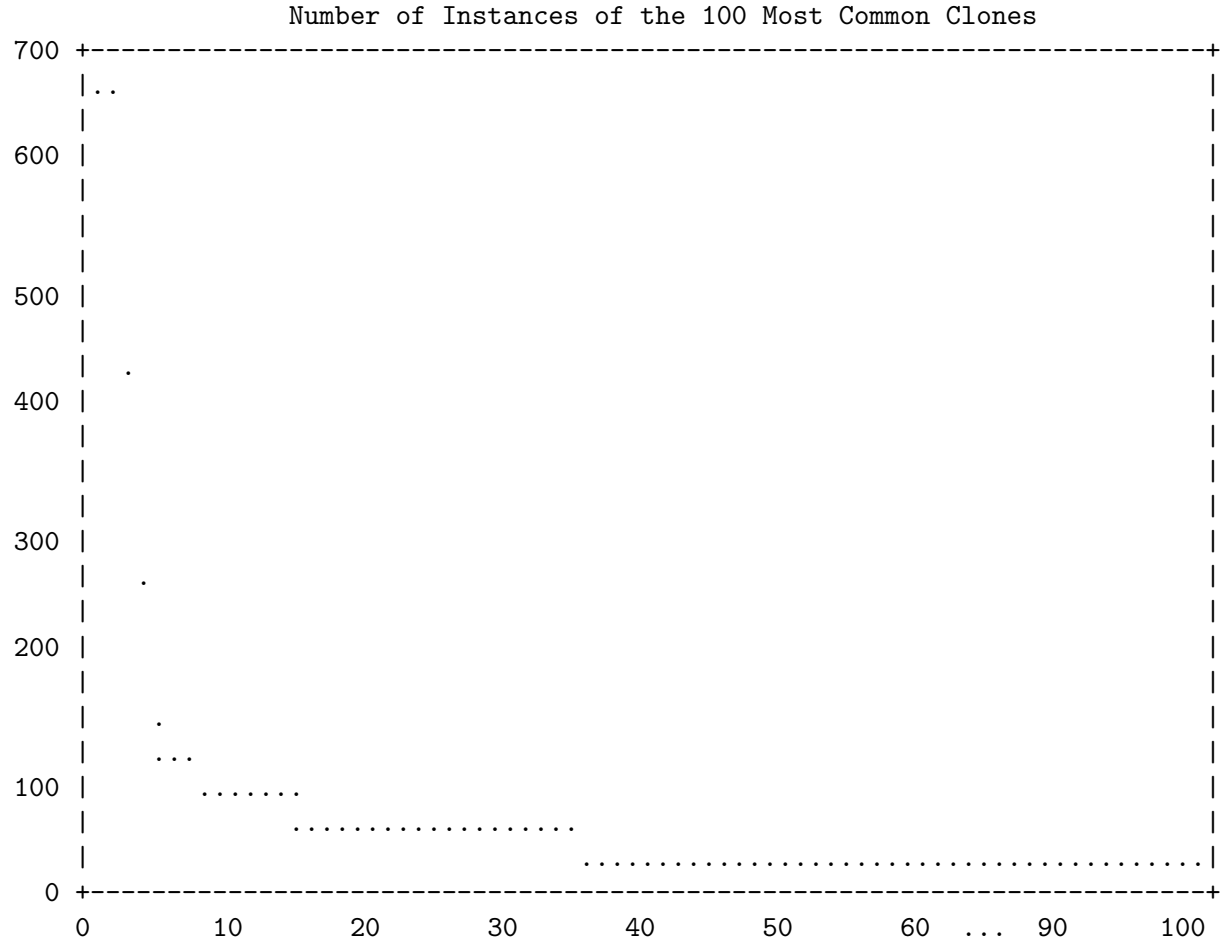
Volume In this scenario we use the entire QualitasCorpus as an existing dataset, and the challenge is not necessarily to process the data at speed (even if this is still desirable), but to process the data within given resource constraints (CPUs, number of threads, and heap size in particular). It becomes especially important to precisely match the required resources with the most cost-efficient deployment when running on a cloud provider. In order to process large volumes of data we introduce Map/Reduce algorithms, and explore their impact on data representations.

In practice, a code clone detection solution would consist of a combination of these two approaches, using an initial set of data (volume) and then continuously add new commits as they are made (velocity). Also, in practice, it should be possible to bulk-add a new block of data without having to reprocess everything.

One point that J. Tigani made in their "Big Data is Dead" article is that most of the time, most of the data is rarely touched after a week, a month, or a year. Code clones are different, since every new piece of code always need to be matched against everything seen before. Since I've already argued that code clone detection is representative of a bigger class of analysis tasks, this implies that big data is not dead. It is just resting.

But what if we don't need to check against all previously seen code all the time? Let's form the hypothesis that *a few clones stand for most of the occurrences.* This would be true if, for example, there are specific programming patterns that are more frequent than others (a more glib explanation is that some answers on StackOverflow are more useful than others). It is even possible to verify this through the data already collected by cljDetector (see graph below). If this is true, then one way to streamline the *velocity* aspect is to only check new code against

the list of previously found clones, and schedule a full analysis once every night, week, or even month. The conclusion of *this* is that data does not need to be big data just because it is plenty, it all boils down to what you want to do with it.



Studying this graph, we see that there are indeed a few clones that are very frequent, but it quickly tapers off to less than 50 instances per clone. Thus, it would make sense to check against these "popular" clones first, but we would eventually have to check against the entire codebase anyway.

With respect to which technology to use, we have explored a variety of programming languages and tools, including microservices in the form of Docker, shell scripts to perform simple tasks (we are seriously under-utilising shell scripts here: we have once implemented the full clone detection in a 160-odd lines script just to prove that you do not need fancy tools), and a choice of two programming languages to perform the clone detection. We have deliberately *not* used any of the much touted Big Data Tools such as Hadoop, Spark, or even Disco, since the underlying principles are much more important than the specific tool. The choices of programming languages is not a coincidence; they have inherent support for many of the same functional programming mechanisms that the Big Data tools try to massively scale, and we make intensive use of these features in our solutions.

One tool used which deserves much more credit than usually given is the database. As seen in the assignments, the database tool is able to optimise and parallelise to a much greater extent than anything we can hope to accomplish with our user-space code within a reasonable budget. We use a NoSQL database, i.e. MongoDB, but there is nothing inherently NoSQL-y with the data we have; a traditional relational database would work just as well and today they can be horizontally scaled just as easily across multiple hardware nodes. With a well used database, bigly data need to be much larger to become Big Data.

One of the reasons why we picked Clojure for the all-at-once analysis is because in my experience this language forces you to pay attention to the structure of the data you are processing rather than the algorithm with which you are going to process it, and when the data structures come into focus you start paying attention to which problem you are actually trying to solve. In that spirit, we'd like to end with a precis of a quote from D. Adams in "The Hitchhikers Guide to the Galaxy":

> Loonquawl: "Do you have... er, that is..."
>
> Deep Thought: "An answer for you? Yes I have." "Though I don't think that you're going to like it"
>
> "The answer to the Great Question... is..."
>
> "Forty-two."
>
> Loonquawl: "Forty-two! Is that all you've got to show for seven and a half million years' work?"
>
> Deep Thought: "I checked it very thoroughly, and that quite definitely is the answer. I think the problem, to be quite honest with you, is that you've never actually known what the question is."

– THE END –

Concept Index

Program Index

Files and Data Types Index

C

Q

T

Functions and Commands Index