Towards Big Data in Education: the case at the Open University of the Netherlands

Citation for published version (APA):

Vogten, H., & Koper, R. (2018). Towards Big Data in Education: the case at the Open University of the Netherlands. In M. Spector, V. Kumar, A. Essa, Y-M. Huang, R. Koper, R. Tortorella, T-W. Chang, Y. Li, & Z. Zhang (Eds.), *Frontiers of Cyberlearning: Emerging Technologies for Teachingand Learning* (pp. 125-143). Springer. Lecture Notes in Educational Technology https://doi.org/10.1007/978-981-13-0650-1_7

DOI:
10.1007/978-981-13-0650-1_7

Document status and date:
Published: 05/10/2018

Document Version:
Peer reviewed version

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:

https://www.ou.nl/taverne-agreement

Take down policy
If you believe that this document breaches copyright please contact us at:

pure-support@ou.nl

providing details and we will investigate your claim.

Downloaded from https://research.ou.nl/ on date: 07 Mar. 2021

Towards Big Data in Education: the case at the Open University of the Netherlands

Hubert Vogten
Open University of the Netherlands
Valkenburgerweg 177
6419 AT Heerlen
hubert.vogten@ou.nl
+31 45 5762126

Rob Koper
Open University of the Netherlands
Valkenburgerweg 177
6419 AT Heerlen
rob.koper@ou.nl
+31 45 5762657

Introduction

When reviewing technology developments over the past centuries a pattern emerges: the rate of these developments is not evenly spread over time, but rather, there seem to be pivotal moments in time when some key developments and discoveries accelerate and fuel a whole range of derived advancements. Some examples of this flywheel effect are the harnessing of steam power, the introduction of electrical power, the discovery of the transistor or the visionary work on user interfaces by Douglas Engelbart (Engelbart and English 1968) and his team.

We argue that we have reached such a pivotal moment in time again, although this time the field is data science. Data science is the emerging intersection of various disciplines such as social science, statistics, and information and computer science. The internet, social networks, new devices such as mobile devices and more recently the internet of things are responsible for an explosion of digital data, which is increasing exponentially each year. Some forecast predict we will produce and consume 40 Zetta bytes by 2020 (Gantz and Reinsel 2012). Data science is all about making sense of these vast amounts of partly unstructured data, so called ‘big data’. There have been three key developments, which are intertwined, that spurred on the data science field.

Firstly, there is the rise of cloud computing, which makes data storage increasingly cheap and ubiquitous while at the same time it provides us with cheap, on-

demand and virtually endless processing power. Cloud computing is also a double bladed knife, as it not only is the backbone for services that are the source of the big data in the first place, but it also provides the computing resources, processing and storage, needed for the data science services themselves. Secondly, there are the recent advances and developments in distributed computing technologies. Google's paper on their MapReduce algorithm (Jeffrey and Sanjay 2008), resulted in a whole range of distributed software systems, libraries and services with the common denominator that they scale very well and therefore are very suitable for processing big data. Thirdly, there have been impressive advancements in field of machine learning. In fact, to such a degree that nowadays artificial intelligence and machine learning are considered to be synonymous. Especially deep learning, which in fact builds on the relative old idea of neural networks reaching back as far as the 1950's with the Perceptron project (Rosenblatt 1958), has shown great promise because large amounts of data combined with ample processing power made this old idea viable albeit with some essential twists on the original idea.

All these developments, glued together via the internet, provide the necessary means to do 'clever stuff' with these big data or phrased more eloquently, they enable the development of smart services. These smart services will affect all of our society and hence also education. The idea of educational smart services is not entirely new. Educational datamining or learning analytics have been around for a while. However, in practice, the data are primarily stemming from the learning management system and are relative limited. Solutions often use traditional and proven technologies, such as learning record stores that depend on relational databases. This approach may be appropriate for now but is in our view is too limited for the next generation of smart services, as relevant data continues to grow exponentially and are not restricted to the LMS. We can expect that data will not merely be the result of human interactions but also will be generated by smart devices such as wearables and the internet of things. Research carried out at OUNL on the relation between some biometric variables and learning effectiveness, showed that traditional learning record stores could not cope with the large data streams produced in the experiment (Di Mitri et al. 2016)

The Open University of the Netherlands (OUNL) launched in 2016 a new project called 'Data Sponge' (DS) with the ambition to research and develop an enterprise level big data infrastructure for OUNL that will enable and stimulate the development of educational smart services. OUNL is in a relative good position to do so, as in 2015 OUNL completed a major step in restructuring their educational model (Schlussmans et al. 2016), moving from a guided self-study model for distance education towards an activated learning model for distance education. This model change was accompanied by the introduction of a complete new learning management system (LMS) (Koper 2014) (Vogten and Koper 2014). The combination of this new educational model and new LMS was also a major step towards a fully digital university and as a result, OUNL has access to a fair amount data. Several departments at OUNL are already making use of these data: the data warehouse of OUNL captures data from various administrative systems mainly to produce information for the management; faculties use the LMS which incorporates a proprietary data store to monitor student's and tutor's progress; the Welten Institute research center has developed an infrastructure for learning analytics that

captures biometric data using Google services. What becomes clear from this is that these efforts are dispersed and therefore are not as effective as they could be. Furthermore, these initiatives are bounded by their respective departments and as a result, data is only sparsely available throughout the wider organization. In other words OUNL has no “single integrated version of the truth” with respect to their data.

DS should overcome typical obstructions when trying to get hold of the dispersed data across various source systems and departments. DS has the ambition to be the single integrated version of the ‘truth’ for researchers, developers of smart services and OUNL’s management. As a consequence, DS should collect as much data as possible even though these data may be not used yet. One could argue that it makes no sense to store these unused data as they can be retrieved later from their respective source systems. This is a faulty assumption however, as we have to be aware that the vast majority of today’s databases reflect designs from decades ago, when memory and disks were very small and very expensive. Databases could simply not afford to keep track of a so called change log. Rather, these databases typically only contain the last known state of an entity which is the result of consecutively applying all incoming changes. As a consequence, if we don’t take any measures, the history of these changes is lost forever. This change log can be essential when developing new smart services. So we need an infrastructure that keeps track of all these changes, for a variety of data sources.

Furthermore, some event data are currently not stored in any of OUNL’s systems but are still very relevant when developing smart services. Examples are mouse clicks, browsing behavior, biometric data etc. DS should be capable to capture these fine grained event data as well, which will not only result in large amounts of data, but will also impact the throughput requirements and characteristics of DS. The DS architecture should be capable to deal with the backpressure arising from sudden bursts of vast amounts of incoming data.

These immutable event and changelog data resemble journal entries in a ledger for the enterprise. Obviously, as these data are immutable, the amount of data will therefore only grow and therefore DS should be capable of dealing with a very large ledger. Such a ledger for the whole enterprise is also known as an Enterprise Data Lake. This ledger can be suitable for some statistical analytics, but most likely, it is not very suitable for most smart services to be used directly. The ledger data have to be transformed into different, more suitable formats, sub-selections and aggregations for an effective processing by most smart services. The prompt transformation of the event data in the ledger is an essential requirement for DS. The term ‘prompt’ is relevant here as some of the smart services may have to provide virtual instantaneous feedback, using the most recent data, while others are much more lenient and are perfectly fine working with data that is maybe a couple of days old. DS must be suited for both real-time and more batch oriented smart services. The resulting transformed data of this transformation that can be queried by the smart services is called the data factory.

Obviously, development of such smart services is an ongoing process. New smart services will be developed while existing smart services have to be maintained because, for example, the provided data formats have changed as a result of alterations in one of the source systems. Furthermore, it must be possible to repair

bugs in the data transformations without losing any data as a result. The DS architecture should have provisions for updating existing and adding new smart services without the risk of losing any data or producing incorrect results.

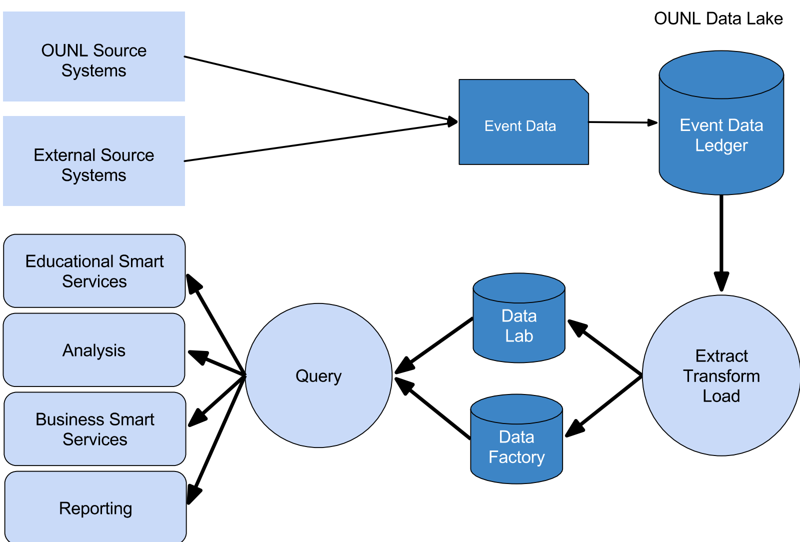

Fig.7.1: high level Data Sponge architecture**.**

In addition DS should facilitate the discovery and the development of new smart services. For it is crucial that data analysts can get a good understanding of the nature of the available data so they can develop new hypothesis and questions that could be answered via new smart services. It should also be possible to develop prototypes validating these assumptions in a very agile way. These are typically functions of a Data Lab. The DS architecture must provide the required agility to support this functionality as well.

Figure 7.1 depicts a high level view on the resulting DS architecture. OUNL has partnered with SURFsara, which will provide the infrastructure for DS, through its high performance computing cloud platform (SURFsara 2017). In the remainder of this chapter we will derive the main none functional requirements of DS and see how we can meet these requirements. Finally, we will describe the resulting DS architecture in more detail.

Data Sponge requirements

From the discussion in the previous sections we can derive a set of non-functional requirements which DS and its underlying architecture has to meet. We define four major requirements: *scalability*, *availability*, *reliability* and *flexibility*.

Scalability is the term we use to describe a system's ability to cope with increased load. Load can be parametrized by the size of the data and the amount of data packages. We shall define 'coping' as being able to deliver similar performance even when the load metrics change. Performance can be measured by throughput, that is: how much data can be processed on average within a certain time period. This is a good indicator for the extent that the data are up to date. For near real-time systems, such as online systems, latency is a very important performance indicator. We define latency as the time it takes from the start of the request until the delivery of the requested data. DS must guarantee scalability of both aspects: DS performance should not deter when more, potentially much more, data is produced. DS latency should not deter when data load increases.

Availability will be rather intuitively defined as the ratio of the total time DS is operational during a given interval to the length of that interval. High availability of DS is of utmost importance as downtime would lead not only to inaccurate data for various smart services, but also to a potential permanent loss of data as incoming data cannot be processed. This is especially the case when these data are not stored by any other source system of OUNL or are exclusively fed into DS. The architecture of DS should take into account that disturbances, such as hardware failures, will not impact its availability.

Reliability is the measure of how far we can trust the data in DS to be correct and up to date. There is an obvious relationship with scalability and availability. However, a scalable and highly available DS does not in itself guarantee that data are correct. We must expect incoming data to be erroneous from time to time for example due to human error. The DS architecture is considered to be reliable when it provides means to correct such errors once they have been detected.

Flexibility is a measure to what extent DS can handle changes in the system. Data fed into DS will change over time as the source systems evolve. This not only applies for new data types, but also for changes in existing data types. Similarly smart services may require different data types as the services evolve over time. DS should be able to cope with these changes, without compromising availability and reliability.

In the next sections we will discusses different architectures that can meet these requirements and we look in more detail at the proposed architecture for DS. We will address each requirement in more detail in the next section and discuss how these requirements influence the architectural choices. Finally, we will present a high level overview of the DS architecture.

Consequences for Data Sponge architecture

The DS architecture must meet the scalability, reliability, availability and flexibility requirements. The first requirement, scalability, will impact the DS architecture most. A major decision is what type technology stack we will use to meet the scalability requirement, which for a large part determines the DS architecture. One option is to use, what we will call 'traditional' technologies, which typically include a relational database and one or more application and/or web servers. Such a three tiered approach is very well understood as it is applied in numerous systems over last decades. An ACID compliant database (Haerder and Reuter 1983), usual SQL compatible, is essential in this type architecture, as the upper layers very much depend on the transactions typically provided by these database management systems. This type of architecture typically will scale well up to a certain point, when the underlying database system becomes too slow. For incoming data it will cause back pressure issues and as a consequence eventually could lead to permanent data loss. This typically occurs when the amount of input data is greater than the system can handle for a prolonged period of time. Another consequence is that database latency will be high and this could also lead to a potentially unacceptable increase in overall system latency, simply because the data cannot be retrieved in due time. Both situations, back pressure on the input data and high latency in the data throughput are obviously undesirable. We could fix such a situation by upgrading the underlying database hardware, which is known as vertical scaling. Vertical scaling only goes so far as what the best hardware has to offer, while at the same time hardware costs increase exponentially when squeezing the last bit of performance out the server hardware. However, there are alternative approaches that could help alleviate the database bottleneck. Probably the first step would be to shard the database, which basically is dividing the database into partitions which are hosted on different database servers. But there is a high price to pay when sharding a relational database. A lot of the logic behind this sharding has to be handled by the application layer and ordinary operational tasks such as backing up, schema changes become much more difficult. An example of the increased complexity introduced by sharding of the database is the multi write problem. As data will be distributed over multiple database servers, the application becomes responsible for the data integration, meaning it must keep the databases up to date with the correct data. This data integration problem is complex and race conditions can lead to faulty data which is very hard to detect and correct. In other words, we have lost the benefits of having an ACID compliant database. Alternatively, we could also introduce additional data caches and alternative storages to increases data throughput. However, such architecture will become very complex very quickly, which is ultimately very difficult to manage, maintain and understand. In conclusion, using a 'traditional' three tier approach has the advantage that the underlying technologies are very well understood and have proven to work well. Nevertheless at a certain point the underlying database technology will not scale anymore without additional measures, which in turn will quickly lead to an architecture that is very complex, messy and very difficult to maintain.

An alternative to these 'traditional' technologies are distributed data systems, which are relative new and received a lot of attention when Google published their paper 'MapReduce: Simplified Data Processing on Large Clusters'. Since, an explosion of environments has emerged including many NoSQL databases and numerous variations on the original MapReduce data processing model. What these applications have in common is the way they approach scalability. Rather than relying on more powerful computer hardware to address scaling as is typical in vertical scaling, they are built around the concept of horizontal scaling. Horizontal scaling is achieved by adding additional computing resources to a cluster of connected nodes which allows the nodes in the cluster to work in parallel at the same tasks. The processing and data load is spread amongst the available nodes in the cluster by one or more supervisor nodes. This approach, theoretically, should scale limitless as long as additional computing resources are available. Cloud computing fits very nicely into this model as it provides the means to increase and decrease the number of computing resources in the cluster as needed.

Distributed data systems, having horizontal scalability in their DNA, are very well suited to process large amounts of heterogeneous data. However, this does not also imply that they are automatically suitable for real time applications typically having low latencies. For example, many MapReduce implementations are rather batch oriented and therefore have not the required low latencies for near real-time processing of data. We will discuss two different approaches that will address this latency problem of batch oriented distributed data processing frameworks. The first approach is known as the 'Lambda architecture' which we will discuss next.

The Lambda architecture in a nutshell

In 'Big Data: Principles and best practices of scalable realtime data systems' (Marz and Warren 2015) Marz and Warren describe an architecture that they dubbed "Lambda Architecture'. This architecture not only addresses the issue of meeting the low latencies requirements with batch oriented distributed data processing frameworks such as Hadoop, but also addresses the reliability and flexibility requirements.

This architecture is made up by three distinct layers: a batch layer, a speed layer and finally a serving layer. The serving layer combines the outcomes of the batch layer and speed layer into multiple up to date views on the input data. Up to date means that the latency of the serving layer is sufficiently low so data in the views can act as input for real-time systems. Figure 7.2 depicts a high level overview of the Lambda architecture.

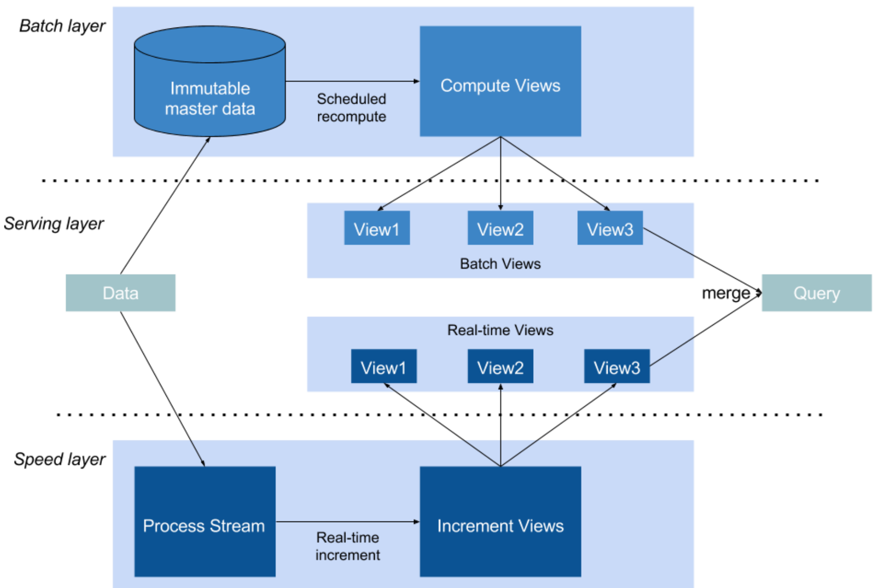

Fig.7.2: the Lambda Architecture**.**

The batch layer uses an immutable master data set as input to re-compute, on regular intervals, the data in views of the batch layer. This processing of the data may take minutes or even hours. Clearly the computed batch views are out of date by the time this processing has been completed as under while new data has been pouring into the master data set. For this reason the architecture also includes the speed layer. The speed layer is responsible for calculating exactly the same views as the batch layer does, but with the distinction that the serving layer only processes the input data that is not already processed by the batch layer. Because the batch layer regularly catches up with the speed layer, the amount of data to be processed by the speed layer at any given moment in time is fairly limited. This limited data set can easily be processed with sufficiently low latencies. The speed layer can use a variety of sub-architectures such as micro batch jobs, micro batched streams or single item streams.

Finally, the serving layer is responsible for merging the outcomes of the batch views and the real-time views into up-to-date views on the input data. The Lambda architecture solves two major problems. First, it provides the low latencies required by near real-time applications, whilst at the same time allows the use of batch oriented distributed technologies such MapReduce to do the majority of the data processing. But maybe as important, the architecture introduces the necessary resilience against faults in the data processing which could be caused for example by changing requirements, modified data formats or programming errors. The key to this resilience is keeping the original input data in an immutable data store. This ensures that no original data is lost and each view can be recomputed at any time.

Updating both the programming for the batch and speed layer with the necessary changes and or fixes, followed by the reprocessing of all input data in the master dataset will return the system in a valid and correct state again. This meets our reliability and flexibility requirement as it allows us to deal with faults and changed requirements.

Although this architecture solves the low latency demands of our scalability requirement, it also introduces additional complexity. First, we need to synchronize the speed layer with the batch on regular intervals, by unloading data from the speed layer once the batch layer views have been updated. Secondly, and more importantly, the speed layer does use a different technology stack from the batch layer and as a consequence the programming code of the batch layer cannot directly be reused in the speed layer. Having two code bases increases the likelihood of interpretation differences and programming errors, while maintenance efforts are at least doubled because every piece of code has to be programmed twice.

The architecture and technologies used in the speed layer differs depending on whether the real-time views are updated synchronously or asynchronously. In case the speed layer views are updated synchronously, the updating process is stopped until all processing has been completed. In most cases this is undesirable, and an asynchronous approach is therefore preferred in which a stream processor acts as buffer avoiding back pressure in the data providers. The data provider will continue immediately after the data is queued by the stream processor. This way, peaks and sudden bursts of data can be easily accommodated. There are many stream processing frameworks available, but in combination with big data processing Apache Kafka (J Kreps et al. 2011) is a very popular choice. Kafka provides a unified, high-throughput, low-latency platform for handling real-time data feeds. The persistent multi-subscriber message queue is built as a distributed transaction log. These features make Kafka an appealing choice as streaming framework for the speed layer.

Interestingly, it is the main architect of Kafka, Jay Kreps who questions the Lambda architecture (Jay Kreps 2014) and proposes an alternative architecture exploiting the unique properties of Kafka, while maintaining the resilience offered by the Lambda architecture.

The Kappa architecture, in a nutshell

Jay Krepps argues in ‘I Heart Logs’ (J Kreps 2014) that streaming micro services using Kafka’s distributed persistent messagebus, could replace the batch layer of the Lambda architecture. By doing so, one of the main drawbacks of the Lambda architecture, the need to maintain two different application environments for the batch and speed layer, can be overcome. This approach is dubbed ‘Kappa architecture’ with an obvious wink to the ‘Lambda Architecture’. Kreps recognizes that one of the strong points of the Lambda architecture is it resilience to cope with changes and bugs by exploiting its immutable master data set. The proposed ‘Kappa’ architecture also provides this resilience, albeit in a slightly different and

more implicit fashion, by using Kafka's unique persistent multi-subscriber message streams.

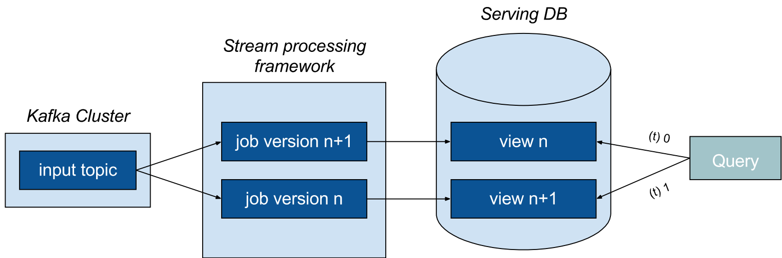

Fig. 7.3: the 'Kappa' architecture

Figure 7.3 depicts the 'Kappa architecture' based on Kafka. It becomes immediately obvious that the batch layer has disappeared in this architecture. A stream processing framework converts all input data, persisted through Kafka input topics into the required views. This approach very much resembles a speed layer of the Lambda architecture that is tuned for asynchronous data processing. However, in the case of the Kappa architecture, all input data will be processed by the stream infrastructure and not only the most recent data as it is the case with the Lambda architecture.

But how does this architecture achieve the resilience of the Lambda architecture? To answer this question we have to look a little closer at the Kafka architecture. Kafka is a distributed messaging system, a real-time stream processor and distributed data store in one closely integrated package. Kafka retains messages, by topic, as an immutable log. The retention period can be configured by topic and may be indefinite. Each topic can have multiple independent subscribers, meaning that each subscriber is receiving all messages of the topic. Each subscriber maintains a pointer to the last read message, which is simply the index of the last processed message by that subscriber. The collection of immutable topic logs very much resembles the immutable master data set of the Lambda architecture. So if we must recalculate our output views as a result of programming errors or perhaps emerging requirements, we can feed the complete topic log again to the stream processing system by simply resetting the last read index of the relevant topic subscribers. While this reprocessing is taking place, which may take many hours, the system would be producing out of date, albeit correct, data. Depending on the type of defect being fixed, it could be preferable to serve more up to date, but less correct, data as long as the reprocessing has not yet had time to catch up. It therefore makes sense not to overwrite the existing output views right away, but instead rename the updated stream processes and the resulting output views by adding a version number to them. This way the old views and the new corrected views coexists for a period of time. Once the new streams are up to date, the consumers of the old views can be configured to start using the latest versions of the output views containing the corrected data. Because both versions of the stream processes

and resulting views are constantly being updated with the latest input there is no immediate pressure to switch all consumers simultaneously, which is essential in real life situations where a centralized release management of various sub-systems is at best undesirable and more likely unrealistic. Once all consumers have been adapted and configured to use the latest versions of the streams and views, we can delete the old version with its corresponding data and thereby free the used computing resources. This way the Kappa architecture achieves a similar resilience against erroneous data and programming bugs as the Lambda architecture. Hence, the Kappa architecture also meets the reliability and flexibility criteria of DS.

In the previous section we did not address another major difference between the two architectures which has to do with scalability. Although both architectures can use a distributed message broker such as Kafka, the scalability demands of this message broker are very different in the Lambda architecture compared to the Kappa architecture. The Lambda architecture has a message broker in the speed layer, if it has one at all. This speed layer only processes data not yet processed by the batch layer and therefore the required low latency is relative easily achieved when compared to the Kappa architecture where the message broker is responsible for processing all incoming data. In other words, the Kappa architecture depends much more on the scalability of the message broker compared to the Lambda architecture. Is Kafka up to this task? Because Kafka is a distributed message broker it will allow vertical scaling by adding additional nodes to the cluster. Kafka is also a persistent message broker. The persistence of the message streams is achieved via a distributed NoSQL key/value store, which implementation can be changed via configuration. This store will scale vertically as well. In fact the developers of Kafka claim that the system is capable of handling millions of message per second in a properly configured Kafka cluster with very low latencies. This should be ample to meet the scalability requirement of DS. This leaves the availability requirement which we will discuss next.

Kafka addresses the availability requirement by introducing a failover mechanism for each topic in the Kafka cluster. A Kafka topic is split into one or more partitions, and each partition is responsible for processing a shard of the total message stream. The distribution is determined by the hash value of a unique message key. The partitions themselves are distributed as evenly as possible over the available Kafka nodes in the cluster. Each partition is replicated across a configurable number of Kafka nodes for fault tolerance and each partition has one node which acts as the ‘leader’ and zero or more nodes which act as ‘followers’. The leader handles all read and write requests for the partition while the followers passively replicate the leader. If the leader fails, one of the followers will automatically become the new leader. Each node acts as a leader for some of its partitions and as follower for others so the load and risks are well balanced within the cluster guaranteeing the availability of the services provided by the cluster should one or more nodes in the cluster fail. In case of a catastrophic failure where none of the replicas are available two alternative recovery scenarios are available. Either wait for a synchronized replica to come back to life and choose this replica as the leader or alternatively choose the first replica that comes back to life, as the leader, which is not necessarily fully synchronized. This is a tradeoff between availability and reli-

ability. Kafka can be configured either way, but by default reliability is sacrificed over availability.

So when properly configured we may conclude that Kafka also meets the reliability requirement of DS and thereby meets all four requirements. This combined with the advantages of the reduced complexity through a single technology stack makes is an appealing choice for DS. However, the message broker is only one, although very important, part of overall Kappa architecture. The stream processing system is the other part and it must meet the scalability, availability, flexibility and reliability requirements as well.

The stream processing system

We didn't pay much attention to the stream processing system so far, but it is an essential component of the Kappa architecture. The stream processing system is focused around so called micro services, which are responsible for small parts of the transformation of the data, very similar to pipelines known from Unix (Kleppmann and Kreps 2015). There are various implementations of these stream processing frameworks such as Apache Storm, Apache Samza, Spark Streams and more recently Kafka Streams (KS). Having a native stream processing framework integrated in Kafka makes an interesting proposition for DS, as this reduces the learning curve and ensures optimal integration. Next we will have a more detailed look at KS and review how KS meets our requirements.

Kafka stream processing applications are ordinary Java applications that can be run everywhere without any special requirements. For packing and deployment KS relies on external specialized tools such as Puppet, Docker, Mesos, Kubernetes or even YARN. So KS does not rely on a proprietary deployment manager. From a deployment perspective, a Kafka stream is just another service that may have some local state on disk, which is just a cache that can be recreated at any time if it is lost or if the streaming application is moved to another node. Kafka will partition and balance the load over the running instances of the streaming application. This partitioning is what enables data locality, scalability, high performance, and fault tolerance.

So KS meets the scalability and availability requirements of DS, given it has been be properly configured. How do KS meet our reliability and flexibility requirements? To answer this question, we must have a closer look at a concept known as 'Stream Table Duality'. We have seen that Kafka threats messages as an immutable changelog. This changelog would therefore only be growing, which could become problematic. To keep the changelog manageable, Kafka has a feature called log compaction. Log compaction determines the most recent version of a changelog entry for every key and discards all other changelog entries for that key. The compacted changelog effectively can be regarded as a traditional state table. KS uses this duality of the changelog to the fullest by interpreting a stream as a changelog of a table and tables as a changelog of a stream.

Stream as Table: A stream can be considered a changelog of a table, where each data record in the stream captures a state change of the table. A stream is thus a table in disguise, and it can be easily turned into a 'real' table by replaying the changelog from beginning to end to reconstruct the table.

Table as Stream: A table can be considered a snapshot, at a point in time, of the latest value for each key in a stream. A table is thus a stream in disguise, and it can be easily turned into a 'real' stream by iterating over each key-value entry in the table.

Because of this duality, the Kafka message broker can used to replicate the local state stores across nodes in the cluster for fault-tolerance. It also provides a mechanism to correct mistakes, as the streaming applications also maintain an index to the last processed changelog entry. Recalculating results is a matter of deleting some intermediate topics and resetting the corresponding indexes. The framework will handle the rest automatically and after some time it takes to catch up, the results will be up to date again. So probably not unsurprisingly, KS fits well in the Kappa architecture and meets the reliability and flexibility requirements of DS.

Cold Start Problem, CDC to the rescue

Now that we have determined a basic architecture and corresponding implementation framework for DS that meets our global requirements, we focus on something we will call the cold start problem. The cold start problem refers to initial lack of data that can directly be fed into DS. In an ideal world all of OUNL's source systems would be extended with triggers, event listeners and so forth that would provide DS with all event data from these systems. However, this is not very realistic as this would require a tremendous effort. More realistically, the required modifications will be implemented as these source systems develop over a prolonged period of time. This process could take years to fully complete. How can we survive this data drought in the meantime?

The most practical and least invasive approach is to develop applications that monitor changes in the databases of the source systems and thus in effect creating a simulated change log on these databases. The advantage of this approach is that the source systems do not have to be affected by this at all, while some of the most relevant data becomes available for DS straight away with a minimum of effort. This approach is also known as Change Data Capture (CDC).

How we monitor DB changes very much depends on the available database technologies and the characteristics of the data involved. For example, some database management systems have out of the box support for an actual changelog, which is also used for replicating the databases for backup purposes. In these cases developing a proprietary change listener feeding directly into DS is a realistic approach. If the used database systems do not have support changelogs other scenarios are possible as well. If data is not very volatile and relative limited in size, such as student course registrations for example, it is possible to create a batch job

that determines the delta of the table values on a daily basis and sends its results to DS.

Obviously, CDC cannot capture data that is not stored in any of the databases and this approach will eventually miss relevant data. So besides implementing CDC, efforts must go towards capturing event data in the various systems as well. However, by establishing a basic DS infrastructure solely based on CDC data, we can showcase DS and make a more informed case to emphasize the importance to make changes to various source systems to capture the missing data.

The Confluent platform extends Kafka with a number of very useful additions among which there is a framework for implementing our CDC requirements, called Kafka Connect (KC). KC defines two basic interfaces: source connectors which are producers that feed Kafka with new data and sink connectors which are consumers that export data from Kafka to various other formats and systems. With this framework it is possible to develop proprietary connectors. However, the Confluent platform also ships a number of standard connectors, among which is a JDBC source and sink connector. These KC connectors can be configured to work in stand-alone or in distributed mode. Distributed mode obviously is targeted at scalability and availability. Whether this is a requirement depends very much on the characteristics of the data, such a volume and volatility. DS will make use of these connectors to overcome the cold start problem by implementing a CDC solution for some of OUNL's most essential source systems.

Data Formats and schemas

The format and semantics of data will change over time as systems continue to develop. This is a major challenge for any data transformation process and therefore also for DS. Semantic changes can be very hard to track and failure to do so can lead to erroneous and unpredictable results in downstream consumers. Unfortunately, besides very tight change management procedures, there is very little in terms of technology that can be offered to overcome this situation. However, there are some solutions that can help to keep track of changes in the data formats used.

Various standards have evolved that allow the formal definition of data structures in a programming language independent manner. Up until recent years XML and more specifically XML DTD's and XML schema's where the representations of choice. More recently, JSON has become very popular and is replacing XML as format of choice. While XML schemas or XML DTDs allow to formal definition of the data structures, JSON does not have any possibility to define data structures out of the box. Furthermore, both formats are very verbose and therefore not very suitable when processing and streaming large amounts of data. To overcome this issue several data language and format independent serialization frameworks have emerged. Probably the best known ones are Apache AVRO, Apache Thrift and Protocol Buffers. These frameworks provide ways to compact rich data structures into an efficient binary format and describe the rich data structures by some sort of schema. Schemas not only play an important role in the definition of the data

structures, but also in the evolution of these data structures. When applications evolve, the data structures change and thereby the schemas must evolve as well. Merely detecting that data structures have changed is useful by itself as it can trigger an alert that producers and consumers are not compatible anymore. However, by designing these schemas cleverly, we can achieve compatibility between older and newer versions of these data structures. Schemas can be backward compatible, meaning that the consumers using the latest version of the schema can process data from producers using an older version. This can for example be achieved by defining default values for data elements that are added in the new version of the schema. Forward compatibility is achieved when a consumer using an older schema version can still process data from a producer that uses a newer schema version. This can be achieved by simply ignoring data elements introduced by the newer schema. Forward compatibility is very important when data is changed upstream and the downstream consumers can't be updated simultaneously. Forward compatibility helps to avoid the need of a big bang release of the entire stack of stream processing applications. In addition, schemas can also be both forward and backward compatible at the same time, which is obviously the most flexible situation. Figure 7.4 depicts the four cases of producer and consumer compatibility or the lack of it.

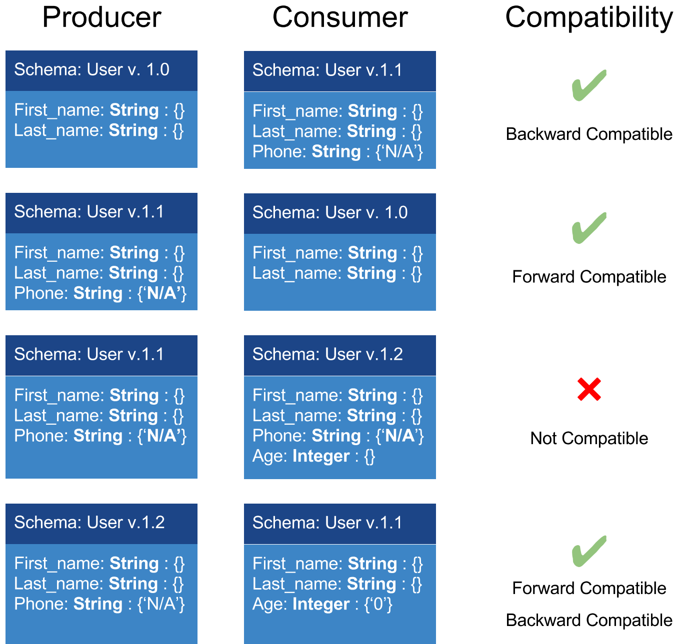

Fig. 7.4: Schema evolution and compatibility

Kafka does not support any of the aforementioned serialization frameworks out of the box. However, Kafka supports some basic stream serializers and de-

serializers (SERDE), which can be extended. The Confluent platform extends Kafka s standard SERDEs with an Apache AVRO SERDE. In addition, the Confluent platform also provides a schema registry that allows the versioned storage of AVRO schemas. This allows the efficient serialization and deserialization of message data into their appropriate formats, while also guaranteeing data compatibility between producer and consumer. Incompatible data automatically trigger an error.

Schema compatibility and more specific forward schema compatibility is essential component to satisfy our flexibility and reliability requirements. The data structures in the source systems will evolve over time, and the downstream processing applications should regardless be able to keep performing their task correctly. This allows for a gradual upgrade of the downstream applications enabling them to start benefiting from the new schema.

Data Sponge Architecture

In the previous sections we discussed the general requirements DS has to meet concerning scalability, availability, reliability, flexibility. We saw that the distributed data systems can overcome scalability issues of more traditional multi-tier systems. The low latency issue, a scalability requirement for near real time systems can be overcome by incorporating a distributed streaming server into our architecture. We reviewed two architectural approaches to overcome the low latency issue and concluded that the Kappa architecture using a Kafka only solution will meet our DS requirements. We argued that sticking to a single framework solution is enticing as it reduces the learning curve and simplifies operations. We also concluded that DS is facing a cold start problem and that is not realistic to expect OUNL systems to be adapted on the short term so they feed their data into DS. CDC using data connectors can help overcome this cold start problem in a fairly elegant manner. Finally we reviewed schemas and schema evolvement and compatibility as a means to guarantee data correctness for producer and consumers.

For the first implementation of DS we will restrict ourselves by merely integrating the most crucial of OUNL source systems in DS. This first implementation will act as a proof of concept and will be a technical validator and pioneering platform on the one hand and a means for generating awareness of the importance of data science within OUNL on the other hand.

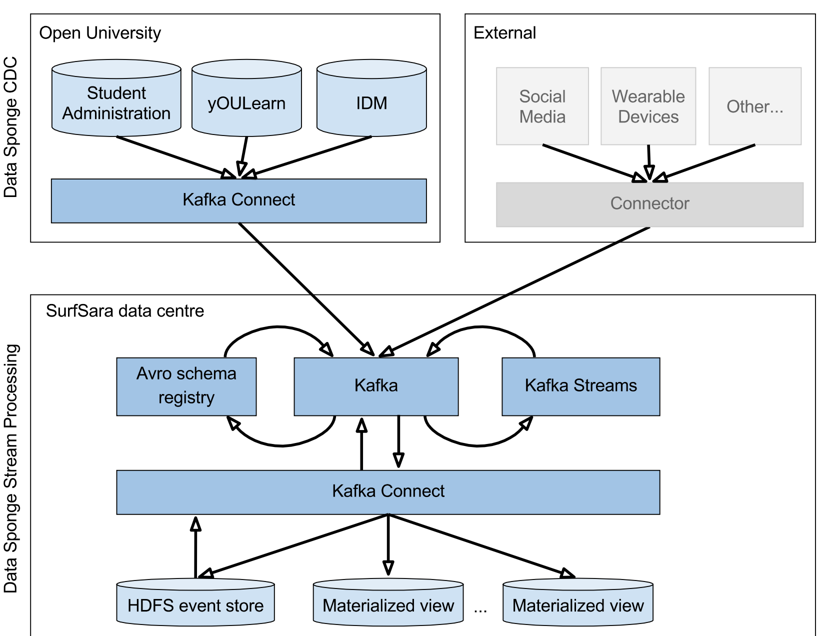

Fig. 7.5: the Data Sponge Architecture

Figure 7.5 depicts the resulting DS architecture. The architecture is divided into two distinct layers. The first layer contains the CDC infrastructure which is using Kafka Connect to keep track of changes three source systems of OUNL:

- Student Administration: the administrative system of OUNL known as SPIL is the source for student enrollments, course registrations, and student grades.
- yOUlearn: OUNLs proprietary LMS. It handles all in course processes and interactions between tutors and students;
- IDM: OUNLs identity management system and provides all users with a single identity across various OUNL subsystems. It also incorporates an access manager handling the log-in and log-out to the OUNL.

Integration of these three systems should provide DS with a first solid data set data that can be for some interesting analyses. At a later stage other systems can be included in the CDC layer as well. The connectors will be hosted by OUNL itself as the required hardware for running these connectors is fairly limited and available. Another part of the first layer is handling user data stemming from external systems and devices such as social networks and wearables. These systems will be connected through their proprietary connectors. Although these external systems are important, they will be out of scope for the first implementation iteration of DS.

The second layer of the architecture is formed by the stream processing framework at which's core is Kafka with some of the Confluent extensions. The Kafka

messaging component is the hub via which all other components communicate. The Kafka message broker cluster is extended with a cluster of nodes that run the Kafka stream processing jobs. Both clusters will be hosted by SURFsara as part of their Big Data Services. An Avro schema registry acts as schema service for the various data formats used. After the necessary processing of the incoming data, the results are exported to views that act as inputs for the smart services. These views are referenced as 'materialized views' because they contain data from several sources that are combined into a denormalized data storage. A materialized view might also contain aggregates or data stemming from some business logic. The consumer of a materialized view determines which data should be available and the stream processing framework will be responsible for a continuous, low latency, delivery of these data to that view. A special materialized view will be an event store that will basically capture all input events into a standardized data format, which is not necessarily the original format of data. This event store can act as input for the event streams in case of cataclysmic failure of the total system. In theory we should be able to rebuild all materialized views, based on this event store.

Next steps

The proposed DS architecture is a result of a journey investigating various solutions for establishing an enterprise level version of the data 'truth' for various target groups at OUNL. Practical experience so far is limited to a set of prototypes that have shown the feasibility of various platforms. In this chapter we have presented the background and motivations for the proposed DS architecture. A prototype has been built that connects to the copy of the yOULearn database via the standard JDBC source connector. This resulting input stream has been processed by a stream processing service that does some very basic joins and counts. However, the proof of the pudding is in the eating. We are in process of launching a Kafka/Confluent cluster on the SURFsara big data infrastructure. The first streaming applications will process some basic data from OUNL's source systems via Kafka connector, similar to the prototype and will produce some basic materialized views. We intent to use the data from the materialized view to construct an appealing info graphic of all learning and teaching activities that are happening at OUNL. This graphic will be projected on the OUNL's information screens present in several buildings for all passing staff, students and visitors to see. This serves a twofold purpose. Firstly, for the first time in OUNL's history, it will provide a feeling of activity at OUNL campus, that otherwise is a somewhat desolate environment characterized by a total lack of students. Remember that OUNL is a distance teaching university and students do not reside on the campus. The secondary goal is raising awareness of the importance and relevance of the DS project within OUNL itself.

Real life experience will tell if the proposed architecture is up to the task, or whether new insights will lead to adaptations. The whole data science field is still

very in turmoil at the moment as generally accepted practices are just start to come into place. Time will tell.

References

Di Mitri, D., Scheffel, M., Drachsler, H., Börner, D., Ternier, S., & Specht, M. (2016). Learning Pulse: Using Wearable Biosensors and Learning Analytics to Investigate and Predict Learning Success in Self-regulated Learning. In *Proceedings of the First International Workshop on Learning Analytics Across Physical and Digital Spaces,* (pp. 34–39). CEUR.

Engelbart, D., & English, W. (1968). A research center for augmenting human intellect. *Proceedings of the December 9-11, 1968,*. http://dl.acm.org/citation.cfm?id=1476645. Accessed 4 May 2017

Gantz, J., & Reinsel, D. (2012). The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*. https://www.emc-technology.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf. Accessed 4 May 2017

Haerder, T., & Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*. http://dl.acm.org/citation.cfm?id=291. Accessed 4 May 2017

Jeffrey, D., & Sanjay, G. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*.

Kleppmann, M., & Kreps, J. (2015). Kafka, Samza and the Unix Philosophy of Distributed Data. *IEEE Data Engineering Bulletin*, 1–11. https://www.cl.cam.ac.uk/research/dtg/www/files/publications/public/mk428/streamproc.pdf

Koper, R. (2014). Towards a more effective model for distance education. *eleed*, (10). https://eleed.campussource.de/archive/10/4010

Kreps, J. (2014). Questioning the Lambda Architecture. *O'Reilly*. https://www.oreilly.com/ideas/questioning-the-lambda-architecture

Kreps, J. (2014). *I Heart Logs: Event Data, Stream Processing, and Data Integration*. O'Reilly Media. https://books.google.nl/books?hl=en&lr=&id=gdiYBAAAQBAJ&oi=fnd&pg=PR3&dq=I+heart+logs,+I+Heart+Logs,+Event+Data,+Stream+Processing,+and+Data+Integration&ots=3wV748ShbL&sig=-GnFj2Rq7vuy-1hBamtw3NF0izo. Accessed 4 May 2017

Kreps, J., Narkhede, N., & Rao, J. (2011). Kafka: A distributed messaging system for log processing. *Proceedings of the NetDB*. http://people.csail.mit.edu/matei/courses/2015/6.S897/readings/kafka.pdf. Accessed 4 May 2017

Marz, N., & Warren, J. (2015). *Big Data: Principles and best practices of scalable realtime data systems* (1st ed.). Greenwich: Manning Publictions Co. http://dl.acm.org/citation.cfm?id=2717065. Accessed 4 May 2017

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*. http://psycnet.apa.org/journals/rev/65/6/386/. Accessed 4 May 2017

Schlussmans, K., Van den Munckhof, R., & Nielissen, R. (2016). Active online

education: a new educational approach at the Open University of the Netherlands. In *The Online, Open and Flexible Higher Education Conference* (pp. 19–21). Rome.

SURFsara. (2017). Big Data Services. https://www.surf.nl/en/services-and-products/big-data-services/index.html. Accessed 4 May 2017

Vogten, H., & Koper, R. (2014). Towards a new generation of Learning Management Systems. In *Proceedings of the 6th International Conference on Computer Supported Education* (Vol. 1, pp. 513–519). Barcelona: CSEDU. doi:10.5220/0004955805140519