Towards a big data reference architecture

Maier, M.

Award date:
2013

Eindhoven University of Technology

Department of Mathematics and Computer Science

Master's thesis

Towards a Big Data Reference Architecture

13th October 2013

Author: Markus Maier
m.maier@student.tue.nl

Supervisor: dr. G.H.L. Fletcher
g.h.l.fletcher@tue.nl

Assessment committee: dr. G.H.L. Fletcher
dr. A. Serebrenik
dr.ir. I.T.P. Vanderfeesten

Abstract

Technologies and promises connected to ‘big data’ got a lot of attention lately. Leveraging emerging ‘big data’ sources extends requirements of traditional data management due to the large volume, velocity, variety and veracity of this data. At the same time, it promises to extract value from previously largely unused sources and to use insights from this data to gain a competitive advantage.

To gain this value, organizations need to consider new architectures for their data management systems and new technologies to implement these architectures. In this master’s thesis I identify additional requirements that result from these new characteristics of data, design a reference architecture combining several data management components to tackle these requirements and finally discuss current technologies, which can be used to implement the reference architecture. The design of the reference architecture takes an evolutionary approach, building from traditional enterprise data warehouse architecture and integrating additional components aimed at handling these new requirements. Implementing these components involves technologies like the Apache Hadoop ecosystem and so-called ‘NoSQL’ databases. A verification of the reference architecture finally proves it correct and relevant to practice.

The proposed reference architecture and a survey of the current state of art in ‘big data’ technologies guides designers in the creation of systems, which create new value from existing, but also previously under-used data. They provide decision makers with entirely new insights from data to base decisions on. These insights can lead to enhancements in companies’ productivity and competitiveness, support innovation and even create entirely new business models.

Preface

This thesis is the result of the final project for my master's program in Business Information Systems at Eindhoven University of Technology. The project was conducted over a time of 7 months within the Web Engineering (formerly Databases and Hypermedia) group in the Mathematics and Computer Science department.

I want to use this place to mention and thank a couple of people. First, I want to express my greatest gratitude to my supervisor George Fletcher for all his advice and feedback, for his engagement and flexibility. Second, I want to thank the members of my assessment committee, Irene Vanderfeesten and Alexander Serebrenik, for reviewing my thesis, attending my final presentation and giving me critical feedback. Finally, I want to thank all the people, family and friends, for their support during my whole studies and especially during my final project. You helped my through some stressful and rough times and I am very thankful to all of you.

Markus Maier, Eindhoven, 13th October 2013

1 Introduction

1.1 Motivation

Big Data has become one of the buzzwords in IT during the last couple of years. Initially it was shaped by organizations which had to handle fast growth rates of data like web data, data resulting from scientific or business simulations or other data sources. Some of those companies' business models are fundamentally based on indexing and using this large amount of data. The pressure to handle the growing data amount on the web e.g. lead Google to develop the Google File System [119] and MapReduce [94]. Efforts were made to rebuild those technologies as open source software. This resulted in Apache Hadoop and the Hadoop File System [12, 226] and laid the foundation for technologies summarized today as 'big data'.

With this groundwork traditional information management companies stepped in and invested to extend their software portfolios and build new solutions especially aimed at Big Data analysis. Among those companies were IBM [27, 28], Oracle [32], HP [26], Microsoft [31], SAS [35] and SAP [33, 34]. At the same time start-ups like Cloudera [23] entered the scene. Some of the 'big data' solutions are based on Hadoop distributions, others are self-developed and companies' 'big data' portfolios are often blended with existing technologies. This is e.g. the case when big data gets integrated with existing data management solutions, but also for complex event processing solutions which are the basis (but got further developed) to handle stream processing of big data [1].

The effort taken by software companies to get part of the big data story is not surprising considering the trends analysts predict and the praise they sing on 'big data' and its impact onto business and even society as a whole. IDC predicts in its 'The Digital Universe' study that the digital data created and consumed per year will grow up to 40.000 exabyte by 2020, from which a third [2] will promise value to organizations if processed using big data technologies [115]. IDC also states that in 2012 only 0.5% of potentially valuable data were analyzed, calling this the 'Big Data Gap'. While the McKinsey Global Institute also predicts that the data globally generated is growing by around 40% per year, they furthermore describe big data trends in terms of monetary figures. They project the yearly value of big data analytics for the US health care sector to be around 300 billion $. They also predict a possible value of around 250 billion € for the European public sector and a potential improvement of margins in the retail industry by 60% [163].

[1] e.g. IBM InfoSphere Streams [29]

[2] around 13.000 exabyte

With this kind of promises the topic got picked up by business and management journals to emphasize and describe the impact of big data onto management practices. One of the terms coined in that context is ‘data-guided management’ [157]. In MIT Sloan Management Review Thomas H. Davenport discusses how organisations applying and mastering big data differ from organisations with a more traditional approach to data analysis and what they can gain from it [92]. Harvard Business Review published an article series about big data [58, 91, 166] in which they call the topic a ‘management revolution’ and describe how ‘big data’ can change management, how an organisational culture needs to change to embrace big data and what other steps and measures are necessary to make it all work.

But the discussion did not stop with business and monetary gains. There are also several publications stressing the potential of big data to revolutionize science and even society as a whole. A community whitepaper written by several US data management researchers states, that a ‘major investment in Big Data, properly directed, can result not only in major scientific advances but also lay the foundation for the next generation of advances in science, medicine, and business’ [45]. Alex Pentland, who is director of MIT’s Human Dynamics Laboratory and considered one of the pioneers of incorporating big data into the social sciences, claims that big data can be a major instrument to ‘reinvent society’ and to improve it in that process [177]. While other researchers often talk about relationships in social networks when talking about big data, Alex Pentland focusses on location data from mobile phones, payment data from credit cards and so on. He describes this data as data about people’s actual behaviour and not so much about their choices for communication. From his point of view, ‘big data is increasingly about real behavior’ [177] and connections between individuals. In essence he argues that this allows the analysis of systems (social, financial etc.) on a more fine-granular level of micro-transactions between individuals and ‘micro-patterns’ within these transactions. He further argues, that this will allow a far more detailed understanding and a far better design of new systems. This transformative potential to change the architecture of societies was also recognized by mainstream media and is brought into public discussion. The New York Times e.g. declared ‘The Age of Big Data’ [157]. There were also books published to describe how big data transforms the way ‘we live, work and think’ [165] to a public audience and to present essays and examples how big data can influence mankind [201].

However the impact of ‘big data’ and where it is going is not without controversies. Chris Anderson, back then editor in chief of Wired magazine, started a discourse, when he announced ‘the end of theory’ and the obsolescence of the scientific method due to big data [49]. In his essay he claimed, that with massive data the scientific method - observe, develop a model and formulate hypothesis, test the hypothesis by conducting experiments and collecting data, analyse and interpret the data - would be obsolete. He argues that all models or theories are erroneous and the use of enough data allows to skip the modelling step and instead leverage statistical methods to find patterns without creating hypothesis first. In that sense he values correlation over causation. This gets apparent in the following quote:

> *Who knows why people do what they do? The point is they do it, and we can track and measure it with unprecedented fidelity. With enough data, the numbers speak for themselves.* [49]

Chris Anderson is not alone with his statement. While they do not consider it the ‘end of theory’ in general, Viktor Mayer-Schönberger and Kenneth Cukier also emphasize on the importance of correlation and favour it over causation [165, pp. 50-72]. Still this is a rather extreme position and is questioned by several other authors. Boyd and Crawford, while not denying its possible value, published an article to provoke an overly positive and simplified point of view of ‘big data’ [73]. One point they raise is, that there are always connections and patterns in huge data sets, but not all of them are valid, some are just coincidental or biased. Therefore it is necessary to place data analysis within a methodological framework and to question the framework's assumptions and the possible biases in the data sets to identify the patterns, that are valid and reasonable. Nassim N. Taleb agrees with them. He claims that an increase of data volume also leads to an increase of noise and that big data essentially means 'more *false* information' [218]. He argues that with enough data there are always correlations to be found, but a lot of them are spurious [3]. With this claim Boyd and Crawford, as well as Talib, directly counter Anderson's postulations of focussing on correlation instead of causation. Put differently those authors claim, that data and numbers do not speak for themselves, but creating knowledge from data always includes critical reflection and critical reflection also means to put insights and conclusions into some broader context - to place them within some theory.

This also means, that analysing data is always subjective, no matter how much data is available. It is a process of individual choices and interpretation. This process starts with creating the data[4] and with deciding what to measure and how to measure it. It goes on with making observations within the data, finding patterns, creating a model and understanding what this model actually means [73]. It further goes on with drawing hypotheses from the model and testing them to finally prove the model or at least give strong indication for its validity. The potential to crunch massive data sets can support several stages of this process, but it will not render it obsolete.

To draw valid conclusions from data it is also necessary to identify and account for flaws and biases in the underlying data sets and to determine which questions can be answered and which conclusions can be validly drawn from certain data. This is as true for large sets of data as it is for smaller samples. For one, having a massive set of data does not mean that it is a full set of the entire population or that it is statistically random and representative [73]. Different social media sites are an often used data source for researching social networks and social behaviour. However they are not representative for the entire human population. They might be biased towards certain countries, a certain age group or generally more tech-savvy people. Furthermore researchers might not even have access to the entire population of a social network [162]. Twitter's standard APIs e.g. do not retrieve all but only a collection of tweets, they obviously only retrieve public tweets and the Search API only searches through recent tweets [73].

As another contribution to this discussion several researchers published short essays and comments as a direct response to Chris Anderson' article [109]. Many of them argue in line with the arguments presented above and conclude that big data analysis will be an additional and valuable instrument to conduct science, but it will not replace the scientific method and render theories useless.

While all these discussions talk about 'big data', this term can be very misleading as it puts the focus only onto data volume. Data volume, however, is not a new problem. Wal-Mart's corporate data warehouse had a size of around 300 terrabyte in 2003 and 480 terrabyte in 2004. Data warehouses of that size were considered really big in that time and techniques existed to handle it [5]. The problem of handling large data is therefore not new in itself and what 'large' means is actually scaling as performance of modern hardware improves. To tackle the 'Big Data Gap' handling volume is not enough, though. What is new, is what kind of data is analysed. While traditional data warehousing is very much focussed onto analysing structured data modelled within the relational schema, 'big data' is also about recognizing value in unstructured sources[6]. These sources are largely uncovered, yet. Furthermore, data gets created faster and faster and it is often necessary to process the data in almost real-time to maintain agility and competitive advantage.

[3] e.g. due to noise

[4] note that this is often outside the influence of researchers using 'big data' from these sources

[5] e.g. the use of distributed databases

[6] e.g. text, image or video sources

Therefore big data technologies need not only to handle the volume of data but also its velocity[7] and its variety. Gartner comprised those three criteria of Big Data in the 3Vs model [152, 178]. Coming together the 3Vs pose a challenge to data analysis, which made it hard to handle respective data sets with traditional data management and analysis tools: processing large volumes of heterogeneous, structured and especially unstructured data in a reasonable amount of time to allow fast reaction to trends and events.

These different requirements, as well as the amount of companies pushing into the field, lead to a variety of technologies and products labelled as 'big data'. This includes the advent of NoSQL databases which give up full ACID compliance for performance and scalability [113, 187]. It also comprises frameworks for extreme parallel computing like Apache Hadoop [12], which is built based on Google's MapReduce paradigm [94], and products for handling and analysing streaming data without necessarily storing all of it. In general many of those technologies focus especially on scalability and a notion of scaling out instead of scaling up, which means the capability to easily add new nodes to the system instead of scaling a single node. The downside of this rapid development is, that it is hard to keep an overview of all these technologies. For system architects it can be difficult to decide which respective technology or product is best in which situation and to build a system optimized for the specific requirements.

1.2 Problem Statement and Thesis Outline

Motivated by a current lack of clear guidance for approaching the field of 'big data', the goal of this master thesis is to functionally structure this space by providing a reference architecture. This reference architecture has the objective to give an overview of available technology and software within the space and to organize this technology by placing it according to the functional components in the reference architecture. The reference architecture shall also be suitable to serve as a basis for thinking and communicating about 'big data' applications and for giving some decision guidelines for architecting them.

As the space of 'big data' is rather big and diverse, the scope needs to be defined as a smaller subspace to be feasible for this work. First, the focus will be on software rather than hardware. While parallelization and distribution are important principles for handling 'big data', this thesis will not contain considerations for the hardware design of clusters. Low-level software for mere cluster management is also out of scope. The focus will be on software and frameworks that are used for the 'big data' application itself. This includes application infrastructure software like databases, it includes frameworks to guide and simplify programming efforts and to abstract away from parallelization and cluster management, and it includes software libraries that provide functionality which can be used within the application. Deployment options, e.g. cloud computing, will be discussed shortly where they have an influence onto the application architecture, but will not be the focus.

Second, the use of 'big data' technology and the resulting applications are very diverse. Generally, they can be categorized into 'big transactional processing' and 'big analytical processing'. The first category focusses on adding 'big data' functionality to operational applications to handle huge amounts of very fast inflowing transactions. This can be as diverse as applications exist and it is very difficult, if not infeasible, to provide an overarching reference architecture. Therefore I will focus on the second category and 'analytical big data processing'. This will include general functions of analytical applications, e.g. typical data processing steps, and infrastructure software that is used within the application like databases and frameworks as mentioned above.

[7] Velocity refers to the speed of incoming data

Building the reference architecture will consist of four steps. The first step is to conduct a qualitative literature study to define and describe the space of ‘big data’ and related work (Sections 2.1 and 2.3.2) and to gather typical requirements for analytical ‘big data’ applications. This includes dimensions and characteristics of the underlying data like data formats and heterogeneity, data quality, data volume, distribution of data etc., but also typical functional and non-functional requirements, e.g. performance, real-time analysis etc. (Chapter 2.1). Based on this literature study I will design a requirements framework to guide the design of the reference architecture (Chapter 3).

The second step will be to design the reference architecture. To design the reference architecture, first I will develop and describe a methodology from literature about designing software architectures, especially reference architectures (Sections 2.2.2 and 4.1). Based on the gathered requirements, the described methodology and design principles for ‘big data’ applications, I will then design the reference architecture in a stepwise approach (Section 4.2).

The third step will be again a qualitative literature study aimed to gather an overview of existing technologies and technological frameworks developed for handling and processing large volumes of heterogeneous data in reasonable time (see the 3 V model [152, 178]). I will describe those different technologies, categorize them and place them within the reference architecture developed before (Section 4.3). The aim is to provide guidance in which situations which technologies and products are beneficial and a resulting reference architecture to place products and technologies in. The criteria for technology selection will again be based on the requirements framework and the reference architecture.

In a fourth step I will verify and refine the resulting reference architecture by applying it to case studies and mapping it against existing ‘big data’ architectures from academic and industrial literature. This verification (Chapter 5) will test, if existing architecture can be described by the reference architecture, therefore if the reference architecture is relevant for practical problems and suitable to describe concrete ‘big data’ applications and systems. Lessons learned from this step will be incorporated back into the framework.

The verification demonstrates, that this work was successful, if the proposed reference architecture tackles requirements for ‘big data’ applications as they are found in practice and as gathered through a literature study, and that the work is relevant for practice as verified by its match to existing architectures. Indeed the proposed reference architecture and the technology overview provide value by guiding reasoning about the space of ‘big data’ and by helping architects to design ‘big data’ systems. that extract large value from data and that enable companies to improve their competitiveness due to better and more evidence-based decision making.

2 Problem Context

In this Chapter I will describe the general context of this thesis and the reference architecture to develop. First, I will give a definition of what 'big data' actually is and how it can be characterized (see Section 2.1). This is important to identify characteristics that define data as 'big data' and applications as 'big data applications' and to establish a proper scope for the reference architecture. I will develop this definition in Section 2.1.1. The definition will be based on five characteristics, namely data volume, velocity, variety, veracity and value. I will describe these different characteristics in more detail in Sections 2.1.2 to 2.1.6. These characteristics are important, so one can later on extract concrete requirements from them in Chapter 3 and then base the reference architecture described in Chapter 4 on this set of requirements.

Afterwards in Section 2.2, I will describe what I mean, when I am talking about a reference architecture. I will define the term and argue why reference architectures are important and valuable in Section 2.2.1, I will describe the methodology for the development of this reference architecture in Section 2.2.2 and I will decide about the type of reference architecture appropriate for the underlying problem in Section 2.2.3. Finally, I will describe related work that has been done for traditional data warehouse architecture (see Section 2.3.1) and for big data architectures in general (see Section 2.3.2).

2.1 Definition and Characteristics of Big Data

2.1.1 Definition of the term 'Big Data'

As described in Section 1.1, the discussion about the topic in scientific and business literature are diverse and so are the definitions of 'big data' and how the term is used. In one of the largest commercial studies titled 'Big data: The next frontier for innovation, competition, and productivity' the McKinsey Global Institute (MGI) used the following definition:

> *Big data refers to datasets whose size is beyond the ability of typical database software tools to capture, store, manage, and analyze. This definition is intentionally subjective and incorporates a moving definition of how big a dataset needs to be in order to be considered big data.* [163]

With that definition MGI emphasizes that there is no concrete volume threshold for data to be considered 'big', but it depends on the context. However the definition uses size or volume of data as only criterion. As stated in the introduction (Section 1.1), this usage of the term 'big data' can be misleading as it suggests that the notion is mainly about the volume of data. If that would be the case, the problem would not be new. The question how to handle data considered large at a certain point in time is a long existing topic in database research and lead to the advent of parallel database systems with 'shared-nothing' architectures [99]. Therefore, considering the waves 'big data' creates, there must obviously be more about it than just volume. Indeed, most publications extend this definition. One of this definitions is given in IDC's 'The Digital Universe' study:

> *IDC defines Big Data technologies as a new generation of technologies and architectures, designed to economically extract value from very large volumes of a wide variety of data by enabling high-velocity capture, discovery, and/or analysis. There are three main characteristics of Big Data: the data itself, the analytics of the data, and the presentation of the results of the analytics.* [115]

This definition is based on the 3V's model coined by Doug Laney in 2001 [152]. Laney did not use the term 'big data', but he predicted that one trend in e-commerce is, that data management will get more and more important and difficult. He then identified the 3V's - data volume, data velocity and data variety - as the biggest challenges for data management. Data volume means the size of data, data velocity the speed at which new data arrives and variety means, that data is extracted from varied sources and can be unstructured or semistructured. When the discussion about 'big data' came up, authors especially from business and industry adopted the 3V's model to define 'big data' and to emphasize that solutions need to tackle all three to be successful [11, 178, 194][231, 9-14].

Surprisingly, in the academic literature there is no such consistent definition. Some researchers use [83, 213] or slightly modify the 3V's model. Sam Madden describes 'big data' as data that is '*too big, too fast, or too hard*' [161], where '*too hard*' refers to data that does not fit neatly into existing processing tools. Therefore '*too hard*' is very similar to data variety. Kaisler et al. define *Big Data as the amount of data just beyond technology's capability to store, manage and process efficiently*', but mention variety and velocity as additional characteristics [141]. Tim Kraska moves away from the 3 V's, but still acknowledges, that 'big data' is more than just volume. He describes 'big data' as data for which '*the normal application of current technology doesn't enable users to obtain timely, cost-effective, and quality answers to data-driven questions*' [147]. However, he leaves open which characteristics of this data go beyond '*normal application of current technology*'. Others still characterise 'big data' only based on volume [137, 196] or do not give a formal definition [71]. Furthermore some researchers omit the term at all, e.g. because their work focusses on single parts [1] of the picture.

Overall the 3V's model or adaptations of it seem to be the most widely used and accepted description of what the term 'big data' means. Furthermore the model clearly describes characteristics that can be used to derive requirements for respective technologies and products. Therefore I use it as guiding definition for this thesis. However, given the problem statement of this thesis, there are still important issues left out of the definition. One objective is to dive deeper into the topic of data quality and consistency. To better support this goal, I decided to add another dimension, namely veracity (or better the lack of veracity). Actually, in industry veracity is sometimes used as a 4th V, e.g. by IBM [30, 118, 224][10, pp. 4-5]. Veracity refers to the trust into the data and is to some extent the result of data velocity and variety. The high speed in which data arrives and needs to be processed makes it hard to consistently cleanse it and conduct pre-processing to improve data quality. This effect gets stronger in the face of variety. First, it is necessary to do data cleansing and ensure consistency for unstructured data. Second the variety of many, independent data sources can naturally lead to inconsistencies between them and makes it hard if not impossible to record metadata and lineage for each data item or even data set. Third, especially human generated content and social media analytics are likely to contain inconsistencies because of human errors, ill intentions or simply because

[1] e.g. solely tackling unstructuredness or processing streaming data

there is not one truth as these sources are mainly about opinion and opinion differs.

After adding veracity, there is still another issue with the set of characteristics used so far. All of them focus on the characteristics of the input data and impose requirements mainly on the management of the data and therefore on the infrastructure level. 'Big data' is however not only about the infrastructure, but also about algorithms and tools on the application level that are used to analyse the data, process it and thereby create value. Visualization tools are e.g. an important product family linked to 'big data'. Therefore I emphasize another V - value - that aims at the application side, how data is processed there and what insights and results are achieved. In fact, this is already mentioned in IDC's definition cited above, where they emphasize the '*economic extraction of value*' from large volumes of varied and high-velocity data [115].

One important note is that, while each 'big data' initiative should provide some value and achieve a certain goal, the other four characteristics do not need to be all present at the same time for a problem to qualify as 'big data'. Each combination of characteristics (volume, velocity, variety, veracity) that makes it hard or even impossible to handle a problem with traditional data mangement methods may suffice to consider that problem 'big data'. In the following I will describe the mentioned characteristics in more detail.

2.1.2 Data Volume

As discussed in Chapters 1.1 and 2.1.1 handling volume is the obvious and most widely recognized challenge. There is however no clear or concrete quantification of when volume should be considered 'big'. This is rather a moving target increasing with available computing power. While 300 terrabyte were considered big 10 years before, today petabyte are considered big and the target is moving more and more to exabyte and even zettabyte [115]. There are estimates that Walmart's processes more than 2.5 petabytes per hour, all from customer transactions [166]. Google processes around 24 petabytes per day and this is growing [92, 165]. To account for this moving target, big volume can be considered as '*data whose size forces us to look beyond the tried-and-true methods that are prevalent at that time*' [137].

Furthermore 'big' volume is not only dependent on the available computing, but also on other characteristics and the application of the data. In a paper describing the vision and execution plan for their 'big data' research, researchers from MIT e.g. claim, that the handling of massive data sets for 'conventional SQL analytics' is well solved by data warehousing technology, while massive data is a bigger challenge for more complex analytics [2] [213].

It is also obvious that big volume problems are interdependent with velocity and variety. The volume of a data set might not be problematic, if it can be bulk-loaded and a processing time of one hour is fine. Handling the same volume might be a really hard problem, if it is arriving fast and needs to be processed within seconds. On the same time handling volume might get harder as the data set to be processed gets unstructured. This adds the necessity to conduct pre-processing steps to extract the information needed out of the unstructuredness and therefore leads to more complexity and a heavier workload for processing that data set. This exemplifies why volume or any other of 'big data's' characteristics should not be considered in isolation, but dependent on other data characteristics.

If we are looking at this interdependence, we can also try to explain the increase of data volume due to variety. After all, variety also means, that the number of sources organizations leverage, extract, integrate and analyse data from grows. Adding additional data sources to your pool also means increasing the volume of the total data you try to leverage. Both, the number of potential data

[2] e.g. machine learning workloads

sources as well as the amount of data they generate, are growing. Sensors in technological artifacts[3] or used for scientific experiments create a lot of data that needs to be handled. There is also a trend to 'datafy'[4] our lives. People e.g. increasingly use body sensors to guide their workout routines. Smartphones gather data while we use them or even just carry them with us. Alex Pentland describes some of the ways how location data from smartphones can be used to get valuable insights [177].

However, it is not only additional data sources, but it is also a change in mindset that leads to increased data volume. Or maybe expressed better: It is that change of mindset that also leads to the urge of even adding new data sources. Motivated by the figures and promises outlined in Chapter 1.1 and some industrial success stories, companies nowadays consider data an important asset and its leverage a possible competitive differentiator [147]. This leads, as mentioned above, to an urge to unlock new sources of data and to utilize them within the organization's analytics process. Examples are the analysis of clickstreams and logs for web page optimization and the integration of social media data and sentiment analysis into marketing efforts.

Clickstreams from web logs were for a long time only gathered for operational issues. Now new types of analytics allowed organisations to extract additional value from those data sets, that were already available to them. Another examples is Google Flutrends, where Google used already available data (stored search queries) and applied it to another problem (predicting the development of flu pandemics) [24, 107, 122]. In a more abstract way, this means that data can have additional value beyond the value or purpose it was first gathered and stored for. Sometimes available data can just be reused and sometimes it provides additional inside when integrated with new data sets [165, pp. 101-110]. As a result organisations start gathering as much data as they can and stop throwing unnecessary data away as they might need it in the future [141][11, p. 7].

Furthermore more data is simply considered to give better results, especially for more complex analytic tasks. Halevy et al. state, that for tasks that incorporate machine learning and statistical methods creating larger data sets is favourable to developing more sophisticated models or algorithms. They call this 'the unreasonable effectiveness of data' [127]. What they claim, is that for machine learning tasks large training sets of freely available, but noisy and not annotated web data typically yields a better result than smaller training sets of carefully cleaned and annotated data and the use of complicated models. They exemplify that with data-driven language translation services and state, that simple statistical models based on large memorized phrase tables extracted from prior translations do a better job than models based on elaborate syntactic and semantic rules. A similar line of argumentation is followed by Jeff Jones, chief scientist at IBM's Entity Analytics Group, and Anand Rajamaran, vice president at WalMart Global eCommerce and teacher of a web-scale data mining class at Stanford University [140, 183, 184, 185].

2.1.3 Data Velocity

Velocity refers to the speed of data. This can be twofold. First, it describes the rate of new data flowing in and existing data getting updated [83]. Agrawal et al. call this the 'acquisition rate challenge' [45]. Second, it corresponds to the time acceptable to analyse the data and act on it while it is flowing in, called 'timeliness challenge' [45]. These are essentially two different issues, that do not necessarily need to occur at the same time, but often they do.

The first of this problems - the acquisition rate challenge - is what Tim Kraska calls 'big throughput' [147]. Typically the workload is transactional[5] and the challenge is to receive, maybe filter, manage

[3] e.g. in airplanes or machines

[4] the term 'datafication' got coined by Viktor Mayer-Schönberger and Kenneth Cukier [165, pp. 73-97]

[5] OLTP-like

and store fast and continuously arriving data[6]. So, the task is to update a persistent state in some database and to do that very fast and very often. Stonebraker et al. also suggest, that traditional relational database management systems are not sufficient for this task, as they inherently process too much overhead in the sense of locking, logging, buffer pool management and latching for multi-threaded operation [213].

An example for this problem is the inflow of data from sensors or RFID systems, which typically create an ongoing stream and a large amount of data [83, 141]. If the measurements from several sensors need to be stored for later use, this is an OLTP-like problem involving thousands of write- or update operations per second. Another example are massively multiplayer games in the internet, where commands of millions of players need to be received and handled, while maintaining a consistent state for all players [213].

The challenge here lies in processing a huge amount of often rather small write operations, while maintaining a somehow consistent, persistent state. One way to handle the problem, is to filter the data, dismiss unnecessary and only store important data. This, however, requires an intelligent engine for filtering out data without missing important pieces. The filtering itself will also consume resources and time while processing the data stream. Furthermore it is also not always possible to filter data. Another necessity is to automatically extract and store metadata, together with the streaming data. This is necessary to track data lineage, which data got stored and how it got measured [45].

The second problem regards the timeliness of information extraction, analysis - that is identifying complex patterns in a stream of data [213]- and reaction to incoming data. This is often called stream analysis or stream mining [62]. McAfee and Brynjolfsson emphasize the importance to react to inflowing data and events in (near) real-time and state that this allows organisations to get more agile than the competition [166]. In many situations real-time analysis is indeed necessary to act before the information gets worthless [45]. As mentioned it is not sufficient to analyse the data and extract information in real-time, it is also necessary to react on it and apply the insight, e.g. to the ongoing business process. This cycle of gaining insight from data analysis and adjusting a process or the handling of the current case is sometimes called the feedback loop and the speed of the whole loop (not of parts of it) is the decisive issue [11].

Strong examples for this are often customer-facing processes [92]. One of them is fraud detection in online transactions. Fraud is often conducted not by manipulating one transaction, but within a certain order of transactions. Therefore it is not sufficient to analyse each transaction on itself, rather it is necessary to detect fraud patterns across transactions and within a user's history. Furthermore, it is necessary to detect fraud while the transactions are processed to deny the transactions or at least some of them [45]. Another example is electronic trading, where data flows get analysed to automatically make buy or sell decisions [213].

Mining streams is, however, not only about speed. As Babcock et al. [55] and Aggarwal [42] point out, processing data streams has certain differences to processing data at rest, both in the approach and the algorithms used. One important characteristic is, that the data from streams evolves over time. Aggarwal [42] calls this ‘'temporal locality’. This means that patterns found in stream change over time and are therefore dependent of the time interval or ‘sliding window’ [55] of the streaming data that is considered through analysis. As streams are typically unbounded, it is often infeasible to do analysis over the whole history, but historical processing is limited up to some point in time or for some interval. Changing that interval can have an effect on the result of the analysis. On the other hand, recognizing changing patterns can also be an analysis goal in itself, e.g. to timely react to a changing buying behaviour.

[6] ‘drink from the firehose’

Furthermore, to be feasible, streaming algorithms should ideally with one pass over the data, that is touching each data point just once while it flows in. Together with the above mentioned unboundedness, but also the unpredictability and variance of the data itself and rate at which it enters the system, this makes stream processing reliant on approximation and sketching techniques as well as on adaptive query processing [42, 55]. Considering these differences, it can be necessary to have distinct functionality for both, e.g. just storing the streaming data in some intermediate, transient staging layer and process it from there with periodical batch jobs might not be enough. This might be even more important, if the data stream is not to be stored in its entirety, but data points get filtered out, e.g. for volume reasons or because they are noise or otherwise not necessary. While the coverage of streaming algorithms is not part of this thesis, which focusses more on the architectural view of the ''big data' environment as a whole, I refer for an overview and a more detailed description to other literature [41, 79].

While the processing of streaming data often takes place in a distinct component, it is typically still necessary to access stored data and join it with the data stream. Most of the time it is, however, not feasible to do this join, all the processing and pattern recognition at run-time. It is often necessary to develop a model[7] in advance, which can be applied and get updated by the streaming-in data. The run-time processing gets thereby reduced to a more feasible amount of incremental processing. That also means, that it is necessary to apply and integrate analytic models, which were created by batch-processing data at rest, into a rule engine for stream processing [45, 231].

2.1.4 Data Variety

One driver of 'big data' is the potential to use more diverse data sources, data sources that were hard to leverage before and to combine and integrate data sources as a basis for analytics. There is a rapid increase of public available, text-focussed sources due to the rise of social media several years ago. This accompanies blog posts, community pages and messages and images from social networks, but there is also a rather new (at least in its dimension) source of data from sensors, mobile phones and GPS [46, 166]. Companies e.g. want to combine sentiment analysis from social media sources with their customer master data and transactional sales data to optimize marketing efforts. Variety hereby refers to a general diversity of data sources. This not only implies an increased amount of different data sources but obviously also structural differences between those sources.

On a higher level this creates the requirement to integrate structured data[8], semi-structured data[9] and unstructured data[10] [46, 83, 141]. On a lower level this means that, even if sources are structured or semi-structured, they can still be heterogeneous, the structure or schema of different data sources is not necessarily compatible, different data formats can be used and the semantics of data can be inconsistent [130, 152].

Managing and integrating this collection of multi-structured data from a wide variety of sources poses several challenges. One of them is the actual storage and management of this data in database-like systems. Relational database management systems (RDBMS) might not be the best fit for all types and formats of data. Stonebraker et al. state, that they are e.g. particularly ill-suited for array or graph data [213]. Array shaped data is often used for scientific problems, while graph data is important due to connections in social networks being typically shaped as graphs but also due to the Linked Open Data Project [2] use of RDF and therefore graph-shaped data.

[7] e.g. a machine learning model

[8] data with a fixed schema, e.g. from relational databases or HTML tables

[9] data with some structure, but a more flexible schema, e.g. XML or JSON data

[10] e.g. plain text

Another challenge lies in the face of semi- and unstructuredness of data. Before this kind of data can truly be integrated and analysed to mine source-crossing patterns, it is necessary to impose some structure onto it [45, 46]. There are technologies available to extract entitities, relationships and other information[11] out of textual data. These lie mainly in the fields of machine learning, information extraction, natural language processing and text mining. While there are techniques available for text mining, there is other unstructured data which is not text. Therefore, there is also a need to develop techniques for extracting information images, videos and the like [45]. Furthermore Agrawal et al. expect that text mining will typically not be conducted with just one general extractor, but several specialized extractors will be applied to the same text. Therefore they identify a need for techniques to manage and integrate different extraction results for a certain data source [46].

This is especially true, when several textual sources need to be integrated, all of them structured by using some extractors. In the context of integrating different data sources, different data - if its initially unstructured, semi-structured or structured - needs to be harmonized and transformed to adhere to some structure or schema that can be used to actually draw connections between different sources. This is a general challenge of data integration and techniques for it are available as there is an established, long-lasting research effort about data integration [46].

Broadly speaking there are two possibilities at which time information extraction and data harmonization can take place. One option is to conduct information extraction form unstructured sources and data harmonization as a pre-processing step and store the results as structured or semi-structured data, e.g. in a RDBMS or in a graph store. The second option is, to conduct information extraction and data harmonization at runtime of an analysis task. The first option obviously improves the runtime performance, while the second option is more flexible in the sense of using specialized extractors tailored for the analysis task at hand. It is also important to note, that in the process of transforming unstructured to structured data only that information is stored, that the information extractors were build for. The rest might be lost. Following the '*Never-throw-information-away*' principle mentioned in Chapter 2.1.2 it might therefore be valuable to additionally store original text data and use a combined solution. In that case information extraction runs as a pre-processing steps, extracted information gets stored as structured data, but the original text data keeps available and can be accessed at runtime, if the extracted information is not sufficient for a particular analysis task. The obvious drawback for this combined approach is a larger growth storage space needed.

Additionally, a challenge lies in creating metadata along the extraction and transformation process to track provenance of the data. Metadata should include which source data is from, how it got recorded there, what its semantics are, but also how it was processed during the whole analysis process, which information extractors where applied etc. This is necessary to give users an idea where data used for analysis came from and how reliable the results therefore are [45].

Data Sources

As mentioned with the growing ability to leverage semi- and unstructured data, the amount and variety of potential data sources is growing as well. This Section is intended to give an overview about this variety.

Soares classifies typical sources for ‘big data’ into 5 categories: Web and social media, machine-to-machine data, big transaction data, biometrics and human-generated data [202, pp. 10-12,143-209]:

[11] e.g. the underlying sentiment

Web Data & Social Media

The web is a rich, but also very diverse source of data for analytics. For one, there are web sources to directly extract content - knowledge or public opinion - from, which are initially intended for a human audience. These human-readable sources include crawling of web pages, online articles and blogs [83]. The main part of these sources is typically unstructured, including text, videos and images. However, most of these sources have some structure, they are e.g. related to each other through hyperlinks or provide categorization through tag clouds.

Next, there is web content and knowledge structured to provide machine-readability. It is intended to enable applications to access the data, understand the data due to semantics, allow them to integrate data from different sources, set them into context and infer new knowledge. Such sources are machine-readable metadata integrated into web pages[12], initiatives as the linked open data project [2] using data formats from the semantic web standard[13] [3], but also publicly available web services. This type of data is often graph-shaped and therefore semi-structured.

Other web sources deliver navigational data, that provide information how users interact with the web and how they navigate through it. This data encompasses logs and clickstreams gathered by web applications as well as search queries. Companies can e.g. use this information to get insight how users navigate through a web shop and optimize its design based on the buying behaviour. This data is typically semi-structured.

A last type of web data is data from social interactions. This can be communicational data, e.g. from instant messaging services, or status updates in social media sites. On the level of single messages this data is typically unstructured text or images, but one can impose semi-structure on a higher level, e.g. indicating who is communicating with whom. Furthermore social interaction also encompasses data describing a more structural notion of social connections, often called the social graph or the social network. An example for this kind are the ‘friendship’ relations on facebook. This data is typically semi-structured and graph-shaped. One thing to note is, that communicational data is exactly that. This means, the information people publish about themselves on social media is for the means of communication and presenting themselves. It is aiming at prestige and can therefore be biased, flawed or simply just lied. This is why Alex Pentland prefers to work with more behavioural data like locational data from phones, which he claims to tell ‘what you’ve chosen to do’ and not ‘what you would kike to tell’ [177]. A concrete example location check-ins people post on foursquare, as they often contain humorous locations that are you used to tell some opinion or make some statement [192]. Therefore one should be cautious how much trust to put into and which questions can be answered by this kind of data .

It is also worth to mention, that these different types of web data are not necessarily exclusive. There can be several overlaps. Social media posts can be both, human-readable publication of knowledge and communicational. The same goes for blog posts, which often include a comment function which can be used for discussion and is communicational. Another example is the Friend of a Friend (FOAF) project [1]. It is connected to the semantic web and linked open data initiatives and can be used to publish machine-readable data modelled as a RDF graph, but at the same time it falls into the category of structural social interactions.

Machine-to-machine data

Machine to machine communication describes systems communicating with technical devices that are connected via some network. The devices are used to measure a physical phenomenon like movement or temperature and to capture events within this phenomenon. Via the network the devices communicate with an application that makes sense of the measurements and captured events and

[12] e.g. through the HTML <metadata> tag or microformats

[13] RDF, RDFS and OWL

extracts information from them. One prominent example of machine to machine communication is the idea of the 'internet of things' [202, p. 11].

Devices used for measurements are typically sensors, RFID chips or GPS receiver. They are often embedded into some other system, e.g. sensors for technical diagnosis embedded into cars or smartmeters in the context of ambient intelligence in houses. The data created by these systems can be hard to handle. The BMW group e.g. predicts its Connected Drive cars to produce one petabyte per day in 2017 [168]. Another example are GPS receivers, often embedded into mobile phones but also other mobile devices. The later is an example of a device that creates locational data, also called spatial data [197]. Alex Pentland emphasizes the importance of this kind of data as he claims it to be close to peoples' actual behaviour [177]. Machine to machine data is typically semi-structured.

Big transaction data
Transactional data grew with the dimensions of the systems recording it and the massive amount of operations they conduct [83]. Transactions can e.g. be purchase items from large web shops, call detail records from telecommunication companies or payment transactions from credit card companies. These typically create structured or semi-structured data. Furthermore, big transactions can also refer to transactions that are accompanied or formed by human-generated, unstructured, mostly textual data. Examples here are call centre records accompanied with personal notes from the service agent, insurance claims accompanied with a description of the accident or health care transactions accompanied with diagnosis and treatment notes written by the doctor.

Biometrics
Biometrics data in general is data describing a biological organism and is often used to identify individuals (typically humans) by their distinctive anatomical and behavioural characteristics and traits. Examples for anatomical characteristics are fingerprints, DNA or retinal scans, while behavioural refers e.g. to handwriting or keystroke analysis [202]. One important example of using large amounts of biometric data are scientific applications for genomic analysis.

Human-generated data
According to Soares human-generated data refers to all data created by humans. He mentions emails, notes, voice recording, paper documents and surveys [202, p. 205]. This data is mostly unstructured. It is also apparent, that there is a strong overlap with two of the other categories, namely big transaction data and web data. Big transaction data that is categorized as such because it is accompanied by textual data, e.g. call centre agents' notes, have an obvious overlap. The same goes for some web content, e.g. blog entries and social media posts. This shows, that the categorization is not mutally exclusive, but data can be categorized in more than one category.

2.1.5 Data Veracity

According to a dictionary veracity means "conformity with truth or fact". In the context of 'big data' however, the term describes rather the absence of this characteristics. Put differently veracity refers to the trust into the data, which might be impaired by the data being uncertain or imprecise [231, pp. 14-15].

There are several reasons for uncertainty and untrustworthiness of data. For one, when incorporating different data sources it is likely, that the schema of the data is varying. The same attribute name or value might relate to different things or different attribute names or value might relate to the same thing. Jeff Jones, chief scientist at IBM's Entity Analytics Group, therefore claims that 'there is no such thing as a single version of the truth' [139]. In fact, in the case of unstructured data there is not even a schema and in the case of semi-structured data the schema of the data is not as exact and clearly defined as it uses to be in more traditional data warehousing approaches, where data is carefully cleaned, structured and adhering to a relational specification [130]. In the case of unstructured data, where information first needs to be extracted, this information is often extracted with some probability and therefore not completely certain. In that sense, variety directly works against veracity.

Furthermore, the data of an individual source might be fuzzy and untrustworthy as well. Boyd and Crawford state, that in the face of 'big data' duplication, incompleteness and unreliability need to be expected. This is especially true for web sources and human-generated content [73]. Humans are often not telling the truth or withholding information, sometimes intentionally, sometimes just because of mistakes and error. Agrawal et al. give several examples for such behaviour. Patients decide to hold back information about risky or embarrassing behaviour and habits or just forget about a drug they took before. Doctors might mistakenly provide a wrong diagnosis [45]. If there are humans in a process, there might always be some error or inconsistency.

There are several possibilities to handle imprecise, unreliable, ambiguous or uncertain data. The first approach is typically used in traditional data warehousing efforts and implies a thorough data cleansing and harmonisation effort during the ETL process, that is at the time of extracting the data from its sources and loading it into the analytic system. That way data quality [14] and trust is ensured up front and the data analysis itself is based on a trusted basis. In the face of 'big data' this is often not feasible, especially when hard velocity requirements are present, and sometimes simply infeasible, as (automatic) information extraction from unstructured data is always based on probability. Given the variety of data it is likely, that there still remains some incompleteness and errors in data, even after data cleaning and error correction [45].

Therefore it is always necessary to handle some errors and uncertainness during the actual data analysis task and manage 'big data' in context of noise, heterogeneity and uncertainness [45]. There are again essentially two options. The first option is, to do a data cleaning and harmonization step directly before or during the analysis task. In that case, the pre-processing can be done more specific to the analysis task at hand and therefore often be leaner. Not every analysis task needs to be based on completely consistent data and retrieve completely exact results. Sometimes trends and approximated results suffice [130].

The second option to handle uncertain data during the analysis task at hand is also based on the notion, that some business problems do not need exact results, but results 'good enough' - that is with a probability above some threshold - are 'good enough' [130]. So, uncertain data can be analysed without cleaning, but the results are presented with some probability or certainty value, which is also impacted by the trust into the underlying data sources and their data quality. This allows users to get an impression of how trustworthy the results are. For this option it is even more crucial thoroughly track data provenance and its processing history [45].

2.1.6 Data Value

While the other four characteristics were used to describe the underlying data itself, value refers to the processing of the data and the insights produced during analysis. Data is typically gathered with some immediate goal. Put differently, gathered data offers some immediate value due to the first time use the data was initially collected for. Of course, data value is not limited to a one-time use or to the initial analysis goal. The full value of data is determined by possible future analysis tasks, how they get realized and how the data is used over time. Data can be reused, extended and newly combined

[14] e.g. consistency

with another data set [165, pp. 102-110]. This is the reason why data is more and more seen as an asset for organisations and the trend is to collect potential data even if it is not needed immediately and to keep everything assuming that it might offer value in the future [141].

One reason why data sets in the context of 'big data' offer value, is simply that some of them are underused due to the difficulty of leveraging them due to their volume, velocity or lack of structure. They include information and knowledge which just was not practical to extract before. Another reason for value in 'big data' sources is their interconnectedness, as claimed by Boyd and Crawford. They emphasize that data sets are often valuable because they are relational to other data sets about a similar phenomenon or the same individuals and offer insights when combined, which both data sets do not provide if they are analysed on their own. In that sense, value can be provided when pieces of data about the same or a similar entity or group of entities are connected across different data sets. Boyd and Crawford call this 'fundamentally networked' [73].

According to the McKinsey Global Institute there are five different ways how this data creates value. It can create transparency, simply by being more widely available due to the new potential to leverage and present it. This makes it accessible to more people who can get insights and draw value out of it [163].

It enables organisations to set up experiments, e.g. for process changes, and create and analyse large amounts of data from these experiments to identify and understand possible performance improvements [163].

'Big data' sets can be used and analysed to create a more detailed segmentation of customers or other populations to customize actions and tailor specific services. Of course, some fields are already used to the idea of segmentation and clustering, e.g. market segmentation in marketing. They can gain additional value by conducting this segmentation and a more detailed micro-level or by doing it in real-time. For other industries this approach might be new and provide an additional value driver [163].

Furthermore the insights of 'big data' analysis can support human decision making by pointing to hidden correlations, potential effects to an action or some hidden risks. An example are risk or fraud analysis engines for insurance companies. In some cases low level decision making can even be automated to those engines [163].

Finally, according to the McKinsey Global Institute 'big data' can enable new business models, products and services or improve existing ones. Data about how products or services are used can be leveraged to develop and improve new versions of the product. Another example is the advent of real-time location data which lead to completely new services and even business models [163].

To create this value the focus of 'big data' gets focussed on more complex, 'deep' analysis [83]. Stonebraker et al. also claim, that conventional, SQL-driven analytics on massive data sets are available and well-solved by the data warehouse community, but that it is more complex analytics tasks on massive data sets, that needs attention in 'big data' research. They name predictive modelling of medical events or complex analysis tasks on very large graphs as examples [213]

In that sense, 'big data' is also connected with a shift to more sophisticated analysis methods compared to simple reports or OLAP exploration in traditional data warehouse approaches. This includes semantic exploration of semi- or unstructured data, machine learning and data mining methods, multivariate statistical analysis and multi-scenario analysis and simulation. It also includes visualization of the entire or parts of the data set and of the results and insights gained by the above mentioned advanced analysis methods [62].

2.2 Reference Architectures

2.2.1 Definition of the term 'Reference Architecture'

Before defining the term 'reference architecture', we must first establish an understanding of the term 'architecture'. Literature offers several definition to describe this later term. Some of the most widely adopted are the following:

> Garlan & Perry 1995: *The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.* [116]
>
> iEEE Standard 1471-2000: *Architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.* [4]
>
> Bass et al. 2012: *The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.* [60, p. 4]

All these definitions have in common, that they describe architecture to be about structure and that this structure is formed by components or elements and the relations or connectors between them. Indeed this is the common ground that is accepted in almost all publications [59, 123, 151, 193]. The term 'structure' points to the fact, that an architecture is an abstraction of the system described in a set of models. It typically describes the externally visible behaviour and properties of a system and its components [59], that is the general function of the components, the functional interaction between them by the mean of interfaces and between the system and its environment as well as the non-functional properties of the elements and the resulting system [15][193]. In other words, the architecture abstracts away from the internal behaviour of its components and only shows the public properties and behaviour visible due to interfaces.

However, an architecture typically has not only one, but several 'structures'. Most more current definitions support this pluralism [59, 60, 123]. Different structures represent different views onto the system. These describe the system along different levels of abstraction and component aggregation, describe different aspects of the system or decompose the system and focus on subsystems add citation. A view is materialized in one or several models.

As mentioned, an architecture is abstract in terms of the system it describes, but it is concrete in the sense of it describing a concrete system. It is designed for a specific problem context and describes system components, their interaction, functionality and properties with concrete business goals and stakeholder requirements in mind. A reference architecture abstracts away from a concrete system, describes a class of systems and can be used to design concrete architectures within this class. Put differently a reference architecture is an 'abstraction of concrete software architectures in a certain domain' and shows the essence of system architectures within this domain [52, 114, 172, 173].

A reference architecture shows which functionality is generally needed in a certain domain or the solve a certain class of problems, how this functionality is divided and how information flows between the pieces (called the reference model). It then maps this functionality onto software elements and the data flows between them [59, pp. 24-26][222, pp. 231-239]. Within this approach reference architectures incorporate knowledge about a certain domain, requirements, necessary functionalities and their interaction for that domain together with architectural knowledge how to design software systems,

[15] e.g. security and scalability

their structures, components and internal as well as external interactions for this domain which fulfil the requirements and provide the functionalities (see Figure 2.1) [52, 173][222, pp. 231-239].

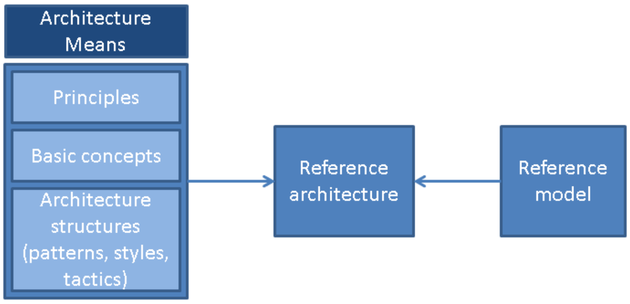


Figure 2.1: Elements of a Reference Architecture [222, p. 232]

The goal of bundling this kind of knowledge into a reference architecture is to facilitate and guide future design of concrete system architectures in the respective domain. As a reference architecture is abstract and designed with generality in mind, it is applicable in different contexts, where the concrete requirements of each context guide the adoption into a concrete architecture [52, 85, 172]. The level of abstraction can however differ between reference architectures and with it the concreteness of guidance a reference architecture can offer [114].

2.2.2 Reference Architecture Methodology

While developing a reference architecture, it is important to keep some of the points mentioned in Section 2.2.1 in mind. The result should be relevant to a specific domain, that is incorporate domain knowledge and fulfil domain requirements, while still being general enough to be applicable in different contexts. This means that the level of abstraction of the reference architecture and its concreteness of guidance need to be carefully balanced. Following a design method for reference architectures helps accomplishing that and the basis for the reference architecture to be well-grounded and valid as well as to provide rigour and relevance.

However, the research about reference architectures and respective methodology is significantly more rare than that about concrete architectures. The choice of design methods in that space is therefore rather limited and the one proposed by Galster and Avgeriou [114] is to the best of my knowledge the most extensive and best grounded of those. Therefore I decided to loosely follow the proposed development process, which consists of the following 6 steps, which are distributed across this thesis.

Step 1: Decide on the reference architecture type
Deciding about a particular type of reference architecture helps to fix its purpose and the context to place it in. The characterisation of the reference architecture and its type will be described in Section 2.2.3. This guides the design and some overarching design decisions as described in the same Section.

Step 2: Select the design strategy
The second step is to decide, if the reference architecture will be designed from scratch (research-driven) or designed based on existing architecture artifacts within the domain (practice-driven). As Galster and Avgeriou [114] point out, the design strategy should be synchronized with the reference architecture type chosen in step 1. Therefore, the selection of the design decision will be made at the end of Section 2.2.3.

Step 3: Empirical acquisition of data
The third step is about identifying and collecting data and information from several sources. It is generally proposed to gather data from people (customers, stakeholders, architects of concrete architectures), systems (documentations) and literature [114]. As the scope of this thesis does not allow to use comprehensive interviews or questionnaires, the reference architecture will mainly be based on the later two. It will involve document study and content analysis of literature about 'big data' including industrial case studies, white papers, existing architecture descriptions and academic research papers. A first result of the literature study is the establishment of requirements the resulting reference architecture will be based on. These requirements will be presented in Chapter 3.

Step 4: Construction of the reference architecture
After the data acquisition, the next step is to construct the reference architecture, which will be described in Chapter 4. As pointed out in Section 2.2.1, an architecture consists of a set of models. Constructing the reference architecture therefore means to develop these models. To structure the set of models Galster and Avgeriou [114] agree with the general recommendation within the software architecture literature to use the notion of views [60, pp. 9-18,331-344][193, pp. 27-37][222, pp. 76-92]. According to the respective iEEE and ISO standards for the design of software architectures[4, 7], a view consists of one or several models that represent one or more aspects of the system particular set of stakeholder concerns. In that sense a view targets a specific group of stakeholders and allows them to understand and analyse the system from their perspective filtering out elements of the architecture which are of no concern for that specific group. This enhances comprehensibility by providing a set concise, focussed and manageable models instead of putting every aspect of the system into one big, complex model which would be hard or impossible to understand. All views together describe the system in its entity, the different views are related and should of course not be inconsistent.

Step 5: Enabling reference architecture with variability
I will omit this step and I will not add specific annotations, variability models or variability views. I consider the variability to be inherit in the abstractness of the reference architecture. I aim for completeness regarding the functional components, so variability can be implemented by choosing the functionality required for a concrete architecture based on its requirements, while leaving unwanted functionality out. Furthermore the last step, the mapping to technology and software platforms will not be a fixed 1:1 mapping, but more loosely discuss several options and choices of technology and software to implement a functional component. It will also not be fixed towards specific industrial products. This provides the freedom to make this choice based on the concrete situation. This freedom is also necessary, considering the whole 'big data' space is not completely mature yet, under steady development and new technologies will still arise during the next couple of years.

Step 6: Evaluation of the reference architecture
Unfortunately it will not be possible to evaluate the reference architecture within a concrete project situation due to the scope of this work, but also due to the lack of access to such a project situation. The evaluation and verification will therefore rely on mapping the reference architecture to concrete 'big data' architectures described in research papers and industrial whitepapers and reports. This will be done in Chapter 5 and allows to evaluate the completeness, correctness, compliance and validity of the reference architecture [114].

2.2.3 Classification of the Reference Architecture and general Design Strategy

As mentioned in Section 2.2.1, reference architectures can have different levels of abstraction. However, this is not the only major characteristic they can differ in. To design a reference architecture it is important to first decide its type, mainly driven by the purpose of the reference architecture. Galster and Avgeriou [114] mention the decision on the type of the reference architecture as the first step in its design and propose a classification method by Angelov et al. [51].

I will follow this proposition but will use a more recent publication of the same authors, in which they extend their initial work [52] to determine the type of a reference architecture. They base their framework on the three dimensions context, goals and design and describe complex interrelations between these dimensions (see Figure 2.2). The architecture goals limit the possible context of the architecture definition and impact its design. The other way round, architecture design and context dictate if the goals can be achieved. Furthermore design choices are made within a certain context and therefore influenced by it. A design choice might also imply a certain context, while it would not be valid in another.

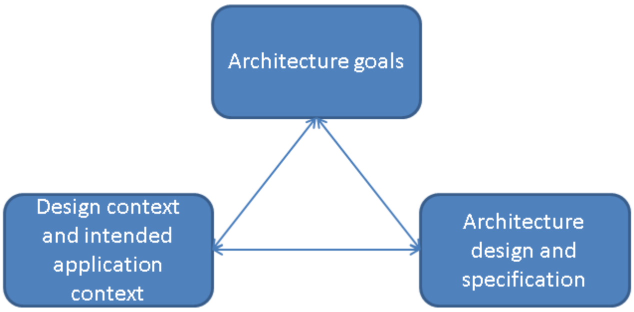

Architecture goals
Design context and intended application context
Architecture design and specification

Figure 2.2: Interrelation between architecture goals, context and design [52]

All these dimensions have a couple of sub-dimensions. However, as hinted on above, not every combination of these dimensions is valid. Angelov et al. call a reference architecture 'congruent', if the goals fit into the context and both are adequately represented within the design. Reference architecture types are then valid, specific value combinations within this dimensional space [52].

Reference Architecture Goals
This dimension classifies the **general goal of a reference architecture** and typically drives decisions about the other two dimensions. While goals in practice are quite diverse and could be classified in more detail, Angelov et al. postulate, that a coarse-granular classification between reference architectures aimed at standardization of concrete architectures and those aimed at facilitation of concrete architectures is sufficient to describe the interplay with the context and design dimension [52].

Reference Architecture Context
The context of a reference architecture classifies the situation in which it gets designed and possible situations in which it can get applied. First, it classifies the **scope of its design and application**, that is if it is designed and intended to be used within a single organization[16] or in multiple organizations [52].

Second, it classifies the **stakeholders that participate in either requirements definition or design** of the reference architecture. These can be software organizations that intend to develop

[16] In this case a reference architecture is sometimes also called a standard architecture for the respective organization

software based on the reference architecture, user organizations that apply software based on the reference architecture or independent organizations[17] [52].

Third, the context defines also **the time a reference architecture gets developed** compared to the existence of relevant concrete systems and architectures. It is necessary to decide if the reference architecture gets developed before any systems implemented the architecture and their entire functionality in practice (preliminary) or as a accumulation of experience from existing systems (classical). Typically reference architectures were based on knowledge from existing systems and their architecture and therefore on concepts proven in practice [85, 171]. That is, they are often classical reference architectures. However, a reference architecture can also be developed before respective technology or software systems actually exist or they might enhance and innovate beyond the existing, concrete architectures in the domain. In that case, they are preliminary [52].

Reference Architecture Design
The design of a reference architecture can be faced with a lot of design decisions and they way it is designed can therefore differ in multiple ways. This dimensions helps to classify some of the general design decisions. First, it can be classified by the **element types** it defines. As stated in most of the definitions of the term 'software architecture' in Chapter 2.2.1, an architecture typically incorporates components, connectors between components and the interfaces used for communication. Another mentioned element type are policies and guidelines. Furthermore a reference architecture can possibly also include descriptions of protocols and algorithms used [52].

Second, there needs to be a decision on which **level of detail** the reference architecture should be designed. Angelov et al. propose a broad classification into detailed, semi-detailed and aggregated elements and the classification can be done individually for each element type mentioned above [52]. The level of detail refers to the number of different elements. While in a more detailed reference architecture different sub-systems are modelled as individual element, in a more aggregated reference architecture sub-systems are not explicitly modelled. It is however difficult to provide a formal measure to distinguish between detailed, semi-detailed and aggregated reference architectures based on the number of elements. In a complex domain an aggregated reference architecture can still contain a lot of elements. The classification is therefore more a imprecise guideline, but Angelov et al. consider this sufficient for the purpose of their framework. It should also be noted, that reference architectures can comprise different aggregation levels to support different phases of the design or for communication with different stakeholders.

Third, the **level of abstraction** of the reference architecture can be classified. It is important to distinguish between abstraction and aggregation as described in the previous sub-dimension. While aggregation refers to how detailed sub-elements are modelled, abstraction refers to how concrete decisions about functionality and used technology are. The sub-dimension differentiates between abstract, semi-concrete and concrete reference architectures. While an abstract reference architecture specifies the nature of the elements in a very general way, e.g. general functionality, a concrete architecture describes very specific choices for each element, e.g. a concrete vendor and software product. A semi-concrete reference architecture lies in between and couples elements to a class of products or technology [52].

Fourth, reference architectures are classified according to the **level of formalization** of the specification. Informal reference architectures are specified in natural language or some graphical ad-hoc notation. Semi-formal specifications use an established modelling language with clearly defined semantics, but one that has no formal or mathematical foundation, e.g. UML. A formal specification uses a formal architecture language, e.g. C2 or Rapide, that has a thorough mathematical foundation and strictly defined semantics [52].

[17] Independent organizations can e.g. be research, standardization, non-profit or governmental organizations

Dimension	Classification
G1: Goal	Facilitation
C1: Scope	Multiple Organizations
C2: Timing	Classical
C3: Stakeholders	Independent Organization (Design) Software Organizations (Requirements) User Organizations (Requirements)
D1: Element Types	Components Interfaces Policies / Guidelines
D2: Level of Detail	Semi-detailed components and policies / guidelines Aggregated or semi-detailed interfaces
D3: Level of Abstraction	Abstract or semi-concrete elements
D4: Level of Formalization	Semi-formal element specifications

Table 2.1: Characteristics of Reference Architectures Type 3 [52]

Application of the Reference Architecture Framework

The framework described above can be applied to guide the design of reference architectures. It can do so, by providing five architecture types placed in the classification space, that the authors claim to be valid and congruent. Reference architectures that cannot be mapped to one of these types are considered incongruent. When designing a new reference architecture these predefined types can be used as guidance for the general design decisions. The application of the framework starts with assessing the general goal and the contextual scope and timing of the planned reference architecture. The result of these decisions can then be mapped against the framework to determine the fitting reference architecture type. If no type fits to the respective choices for goals and context it is a strong indication, that these choices should be revised. Otherwise, the next step after mapping the type, is to ensure, that input from the stakeholders specified in the chosen type is available. If this is not possible, the goals and context should again be revised to fit to the available stakeholder input or the design effort should be stopped. If a match is found, the general design decisions can be taken as guidelines from the identified type [52].

As described in the problem statement (Chapter 1.2), this thesis aims to give an overview of existing technology within the ‘big data’ space, put it into context and help architects designing concrete system architectures. Therefore, the general goal according to the framework is clearly **facilitation**. This rules out the choice of classical reference architectures aimed at standardization for both, multiple organizations (type 1) and within a single organization (type 2). The scope of the reference architecture will not be focussed onto one organization, but it is intended to be general enough to be applicable in **multiple organizations**, making a classical, facilitation architecture to be used within a single organization (type 4) a poor choice. Furthermore, there already exist multiple system in the ‘big data’ space and much of the underlying technology is available and proven. According to the timing sub-dimensions the reference architecture can thus be classified as **classical**. Mapped against the framework, a classical, facilitation reference architecture to be used in multiple organizations (type 3) is therefore the fitting choice (see Table 2.1) and not a preliminary, facilitation architecture (type 5), which aims to guide and move the design of future systems forward.

A type 3 reference architecture is a ‘classical, facilitation architecture designed for multiple organizations by an independent organization’ [52]. This kind of reference architecture is **developed by an independent organization**, typically a research center, based on existing knowledge and experience about respective architectures in research and industry. One critical point and possible weakness of the resulting reference architecture is, that it is not possible, due to the scope and timeline of the work, to actively involve user and software organizations. The requirements elicitation is therefore only based on a literature study. This can lead to overlooking requirements important for practice or over-emphasizing requirements that are less important to practice. However, the reference architecture will be verified by mapping existing architecture from industry onto it. This might help to reduce this weakness.

From the design dimension we can now derive general design decisions. As this reference architecture intends to give an overview of necessary functionality, existing technologies, how the functionality can be distributed onto existing software packages and how different packages interact within the architecture, the use of components, connectors and interfaces makes intuitively sense for the design. Policies and guidelines will be used to give decision rules in cases, where multiple options, e.g. multiple types of database systems, exist to implement a certain functionality. The design of the reference architecture will be conducted in multiple phases, starting with a aggregated and abstract view to then stepwise refine the models and add more detail and concreteness. According to the framework, the design will not be too detailed and definitely not to concrete, e.g. specifying an element to be a graph-based database systems but not explicitly to be Neo4J, to allow for a broader applicability of the reference architecture. Views according to the different levels of details and abstraction will be included. Referring to the framework's recommendation to use a semi-formal way to specify the design elements, I will use UML and some of its diagram types.

In Chapter 2.2.3 I picked the timely context to refer to a classical reference architecture. From that point of view it is only logical to base the reference architecture on existing architectural artifacts and technology and therefore decide for a **practice-driven** design strategy.

Note however, that this classification is to some extent arguable. It is true, that involved technologies are available. The Google File System [119], MapReduce [94] as well as their open-source implementations within Hadoop [12] are just examples. However, while there are Apache projects aiming at solving this, supporting tools e.g. for administration and metadata management are still not completely mature. Some software companies relying on Hadoop, e.g. Cloudera [23], try to fill these gaps with their own solutions, some of them open-source, other proprietary. Furthermore, most published architectures focus on parts of the 'big data' space, while there is, to the best of my knowledge, no published, concrete or reference architecture that aims at the space as a whole. Therefore the design strategy is to some extent hybrid, based on existing architectural artifacts and technologies where possible and suggesting possible solutions where not.

2.3 Related Work

2.3.1 Traditional BI and DWH architecture

Business intelligence is a widely but rather ambiguously used term. It typically describes all technologies, software applications and tools used to create business insights and understanding and to support business decisions. That includes the whole data lifecycle from data acquisition to data analysis and the back-flow of analysis results to adjust and improve business processes. However, the term is often not only used to describe software tools, but a holistic, enterprise-wide approach for decision support. This broader definition additionally incorporates analysis and decisions processes , organizational standards, e.g. standardized key performance indicators, practices and strategies, e.g. knowledge management[61, pp. 13-14].

Considering this, it is obvious that ‘big data’ falls into the area of business intelligence, at least with its analytical part, which is the scope of this thesis. Traditionally, the data warehouse is software tool within this area, that is responsible for integrating data, structuring and preparing it and storing it in a way that supports and is optimized for analytical applications. In that, data warehousing has a big overlap in tasks with ‘big data’ solutions or put differently, it is most influenced by the advent of ‘big data’ and its characteristics. Therefore it makes sense to give an overview of data warehousing principles and typical architectures.

A data warehouse is an organization-wide, central repository for historical, physically integrated data from several sources, applications and databases. The data is prepared, structured and stored with the aim of facilitating access to this data for analysis and decision support purposes[61, 82, 100, 188]. Additionally, in its initial definition by Inmon [135], a data warehouse is subject-oriented (the data is structured to represent real-world objects, e.g. products and customers), time-variant (data is loaded in time intervals and stored with appropriate timestamps to allow comparison and analysis over time) and non-volatile (data is stable and is not changed or deleted once it was loaded). Yet, in practice the characteristics of Inmon’s definition were often criticized for being both too strict and not significant enough, e.g. by Bauer and Günzel [61, pp. 7-8] and Jiang [138], and the focus of a data warehouse is on the integrated view onto data optimized for analysis purposes. Data warehousing then describes the whole technical process of extracting data from its sources, integrating, preparing and storing. There is however an ambiguous use of the term data warehouse of either being an information system supporting the whole data warehousing process including data transformation routines or only being the central repository to store the data.

Based on this definition Figure 2.3 shows an early, general data warehousing architecture described by Chaudhuri and Dayal [82].

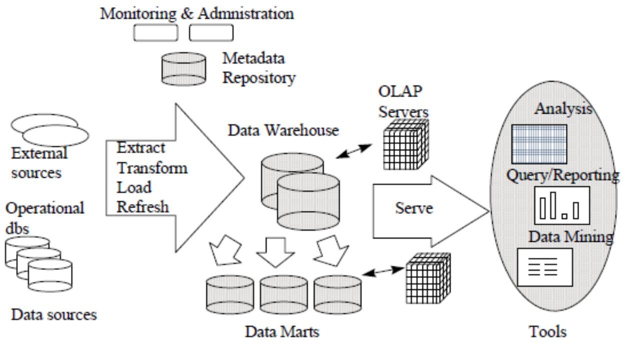


Figure 2.3: Traditional Data Warehousing Architecture [82]

According to this a traditional data warehousing architecture encompasses the following components [82]:

- data sources as external systems and tools for extracting data from these sources
- tools for transforming, that is cleaning and integrating, the data
- tools for loading the data into the data warehouse
- the data warehouse as central, integrated data store
- data marts as extracted data subsets from the data warehouse oriented to specific business lines, departments or analytical applications
- a metadata repository for storing and managing metadata

- tools to monitor and administer the data warehouse and the extraction, transformation and loading process
- an OLAP (online analytical processing) engine on top of the data warehouse and data marts to present and serve multi-dimensional views of the data to analytical tools
- tools that use data from the data warehouse for analytical applications and for presenting it to end-users

This architecture exemplifies the basic idea of physically extracting and integrating mostly transactional data from different sources, storing it in a central repository while providing access to the data in a multi-dimensional structure optimized for analytical applications[53, 100, 188]. However, the architecture is rather old and, while this basic idea is still intact, it is rather vague and imprecise about several facts.

First, most modern data warehousing architectures use a staging or acquisition area between the data sources and the actual data warehouse[53, 100, 228][61, pp. 55-56]. This staging area is part of the extract, transform and load process (ETL process). It temporarily stores extracted data and allows transformations to be done within the staging area, so source systems are directly decoupled and not longer strained.

Second, the interplay between data warehouse and data marts in the storage area are not completely clear. Actually, in practice this is one of the biggest discourses about data warehousing architecture with two architectural approaches proposed by Bill Inmon and Ralph Kimball [74]. Inmon places his data warehousing architecture in a holistic modeling approach of all operational and analytical databases and information in an organization, the Corporate Information Factorty (CIF). What he calls the atomic data warehouse is a centralized repository with a normalized, still transactional and fine-granular data model containing cleaned and integrated data from several operational sources[135]. Subsets of the data from the centralized atomic data warehouse can then be loaded into departmental data marts where they are optimized and stored oriented at analysis purposes, typically online analytical processing (OLAP).

Storing data OLAP oriented means, that they are transferred into a logical, multi-dimensional model, often called data cube, where data is structured according to several dimensions, which represent real-world concepts, e.g. products, customers or time. In a relational database this multi-dimensional model is typically implemented either via the star- or the snowflake schema. Both consist of a central fact table which contains different key figures as attributes and refers to surrounding dimension tables via foreign key relations. The dimension tables describe the real-life concepts, that are used to structure the fact data, and their attributes. They are typically hierarchical, e.g. a time dimension containing attributes hour, day, month and year. In the star schema the dimension tables are flat, while they are normalized in the snowflake schema. See Figure 2.4 for examples of both. Additionally, optimization for analysis purposes also includes calculation of application specific key figures, aggregation and view materialization, that is the pre-calculation and physical storage of data views that are often used by analytical applications and during analysis. An OLAP engine then provides access to this multi-dimensional data in the data marts, presents views of the data to front-end tools, e.g. for reporting, dashboarding or ad-hoc OLAP querying, and allows navigation through the multi-dimensional model by translating OLAP queries into actual SQL queries that can be processed in the database[18].

[18] This is only a brief summarization of these concepts, to get an idea of the building blocks in the overall data warehouse architecture. A complete description is out of scope and would be too extensive. For a definition of OLAP and its principles see the original paper by Codd et al. [86]. For a short, general introduction into all mentioned concepts see Chaudhuri and Dayal [82]. For an in-depth description of OLAP functionality, the multi-dimensional model and the star- and snowflake schema see e.g. Bauer and Günzel [61, pp. 114-130,201-338] or Kimball et al. [145, pp. 137-314].

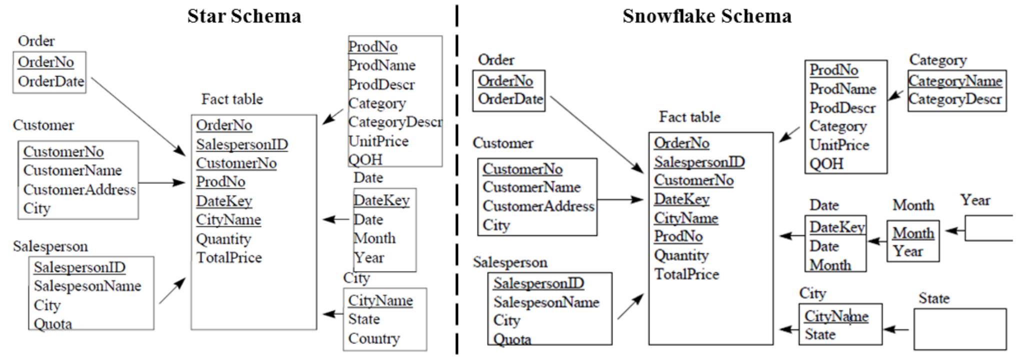


Figure 2.4: Comparison of the star schema (left) and the snowflake schema (right) [82]

Inmon's approach, also called enterprise data warehouse architecture by Ariyachandra and Watson [53], is often considered a top-down approach, as it starts with building the centralized, integrated, enterprise-wide repository and then deriving data marts from it to deliver for departmental analysis requirements. It is however possible, to build the integrated repository and the derived data marts incrementally and in an iterative fashion. Kimball on the other hand proposes a bottom-up approach which starts with process and application requirements[142, 145]. With this approach, first the data marts are designed based on the organization's business processes, where each data mart represents data concerning a specific process. The data marts are constructed and filled directly from the staging area while the transformation takes places between staging area and data marts. The data marts are analysis-oriented and multi-dimensional as described above. The data warehouse is then just the combination of all data marts, where the single data marts are connected and integrated with each other via the data bus and so-called conformed dimensions, that is data marts use the same, standardized or 'conformed' dimension tables. If two data marts use the same dimension, they are connected and can be queried together via that identical dimension table. The data bus is then a net of data marts, which are connected via conformed dimensions. This architecture (also called data mart bus architecture with linked dimensional data marts by Ariyachandra and Watson [53]) therefore forgoes a normalized, enterprise-wide data model and repository.

Based on some of these ideas Bauer and Günzel [61, pp. 37-86] describe a more detailed reference architecture for data warehousing, see Figure 2.5. Note, that they use the term data warehouse in a rather extensive meaning, where it encompasses the entire system including extraction, transformation and load processes and procedures and not just the centralized repository. What gets apparent in their reference architecture is the idea of data getting processed and transformed in multiple stages, comparable to a pipeline, from the raw source data to data prepared and optimized for analysis. The pipeline processing gets triggered by the monitor, which tracks changes in the data sources. From there, the data first gets extracted into a temporary staging area, then cleaned and integrated in a first transformation step before it gets loaded into the basis database, which is a normalized, integrated and central repository comparable to Inmon's atomic data warehouse. This represents the first part of the data warehousing pipeline, the integration area. Afterwards a second transformation step in the analysis area transforms the data into the multi-dimensional, analysis-oriented model and loads it into a central, derived database. Views from this central, multi-dimensional data store can then be loaded into analysis databases, that is data marts specific for departmental use or certain applications. Analysis tools are applied on top of these analysis databases / data marts. Compared to a typical Inmon architecture, Bauer and Günzel [61] therefore add the derived database as central repository of multi-dimensional data, before the data gets distributed into the data marts. In a sense, this central derived database can be seen as a version of Kimball's data bus and it leads to the fact, that dimensions across data marts are standardized, which is not necessarily the case for Inmon architectures. The whole data warehousing pipeline gets controlled by a single data warehouse manager. It e.g. triggers the processing steps and rolls back or restarts failed tasks. Additionally, a metadata manager mediates the access to a central metadata repository, provides metadata to the processing steps where needed and extracts metadata along the processing pipeline.

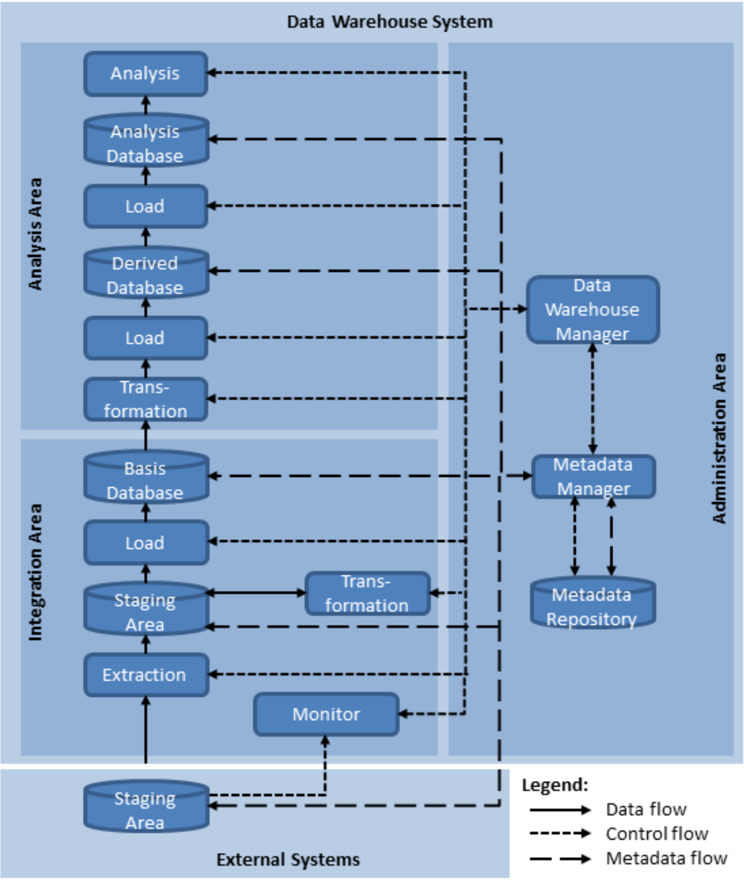


Figure 2.5: Data Warehouse Reference Architecture as adapted and translated from Bauer and Günzel [61]

2.3.2 Big Data architectures

To my best knowledge, currently there is no extensive and overarching reference architecture for analytical 'big data' systems available or proposed in literature. One can find however several concrete, smaller-scale architectures. Some of them are industrial architecture and product-oriented, that is they reduce the scope to the products from a certain company or from a group of companies. Some of them are merely technology-oriented or on a lower lever. These typically omit a functional view and mappings of technology to functions. None of them really fits into the space of an extensive, functional reference architecture. To a large extent that is by definition, as these are typically concrete architectures

One of those product-oriented architectures is the 'HP Reference Architecture for MapR M5'[25]. MapR is a company selling services around their Hadoop distribution. The reference architecture described in this white paper is then more an overview of the modules incorporated in MapR's Hadoop distribution and of the deployment of this distribution on HP hardware. One could consider it a product-oriented deployment view, but it is definitely far from a functional reference architecture.

Oracle also published several white papers onto ‘big data’ architecture. In their first white paper [217], they described a very high-level ‘big data’ reference architecture along a processing pipeline with the steps ‘acquire’, ‘organize’, ‘analyze’ and ‘decide’. They keep it very close to traditional information architecture based on a data warehouse supplemented with unstructured data sources, distributed file systems or key value stores for data staging, MapReduce for the organization and integration of the data and additional sandboxes for experimentation. Their reference architecture is however, just a mapping of technology categories to the high-level processing steps. They provide little information about interdependencies and interaction between modules. However, they provide three architectural patterns. These can be useful, even if they are kind of trivial and mainly linked to Oracle products. These patterns are (1) mounting data from the Hadoop Distributed File System via virtual table definitions and mappings into a database management system so they can be directly queried with SQL tools, (2) using a key value stores to stage low-latency data and provide it to a streaming engine, while using Hadoop to calculate rule models and provide them to the streaming engine for processing the real-time data and raising alerts if necessary, (3) using the Hadoop file system and key value stores as staging areas from which data is processed via MapReduce either for advanced analytics applications (e.g. text analytics, data mining) or for loading the results into a data warehouse, where they can be further analyzed using in-database analytics or be accessed by further business intelligence applications (e.g. dashboards). The key principles and best practices that are focussed on in the paper are first, to integrate structured and unstructured data, traditional data warehouse systems and ‘big data’ solutions, that is to use e.g. MapReduce as a pre- and post-processor for traditional, relational sources and link the results back. The second key principle they mentions, is to plan for and facilitate experimentation in a sandbox environment. In a second, more current white paper [101], they refresh the idea of using distributed file systems and NoSQL databases, especially key value stores, for data acquisition and staging and MapReduce for data organization and integration, while results are written back to a relational data warehouse for in-database analytics and as a structured source for other analytical applications. They do this however in a very product-oriented way, mainly mapping Oracle products onto the different stages.

A third paper from Oracle [77], takes that idea of incorporating ‘big data’ technologies and knowledge discovery through data mining into a traditional information architecture environment and covers in more detail two process approaches to organizationally arrive at an integrated architecture, starting from a traditional enterprise data warehouse system. Both of them add a knowledge discovery layer to the data warehouse architecture, which contains an ‘analytical discovery sandbox’. These sandboxes are then used to experiment how ‘big data’ can be used to derive new knowledge. The derived knowledge as well as the used ‘big data’ capabilities are then incorporated into the enterprise data warehouse architecture, either into the ETL process or as an pool for unstructured data into the foundational layer linked to the basis data warehouse. Derived knowledge in the sense of calculated rule models can also be incorporated into a complex event processing engine.

Another reference architecture is proposed by Soares [202, pp. 237-260], which he describes as part of his ‘big data’ governance framework (see Figure 2.6). Again, the proposed reference architecture is kind of high level and while it provides a good overview of software modules or products applicable for ‘big data’ settings, it no little information about interdependencies and interaction between these modules. Furthermore, the semantics are not clear. There is e.g. no explanation, what the three arrows mean. It is also not clear, what the layers mean, e.g. if there is an chronological interdependency or if the layer got ordered depending on usage relations. They are also on different levels and there are some overlaps between layers. Data warehouses and data marts e.g. are implemented using databases. That is, they are technically on different levels. A usage relation would be applicable, but this does not work for data warehouses and big data sources, as those are different systems and both an the same, functional level. An overlap exists e.g. between Hadoop Distributions and the Open Source

Foundational Componens (HDFS, MapReduce, Hadoop Common, HBase). Therefore, the reference architecture can give some ideas, what functionality and software to take into account, but it is far from a functional reference architecture, which is the objective of this thesis.

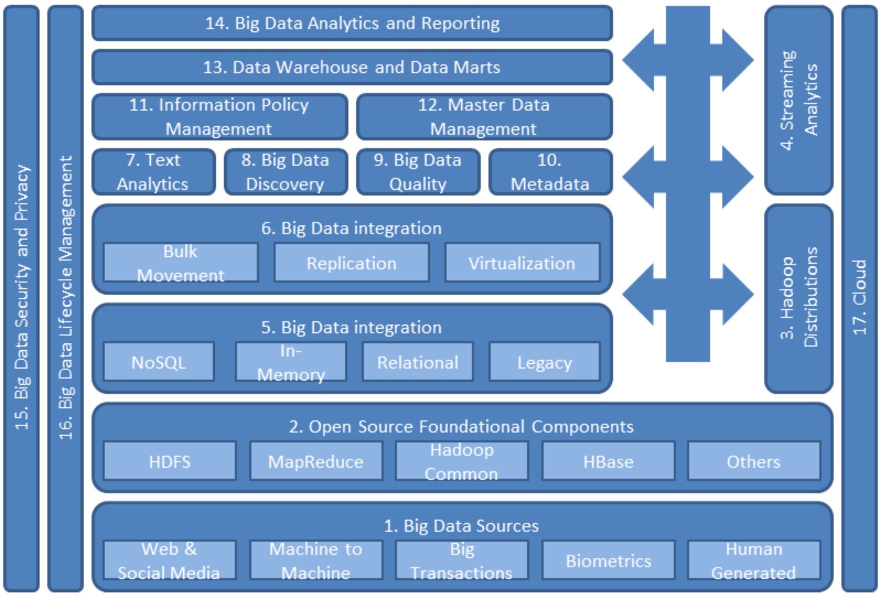


Figure 2.6: A reference architecture for big data taken from Soares [202, p. 239]

In academia there is, to the best of my knowledge, also no proposal for an overarching reference architecture. Most research effort and architectural work is on a lower level, targeting specific platforms, concrete systems or software. One example for this is the ASTERIX project [47, 64, 71, 72]. It describes a concrete architecture consisting of a data management system to perform query in a self-developed query language, AsterixQL, an algebraic abstraction layer which also serves as a virtual machine to ensure compatibility to other query languages, e.g. HiveQL, and data parallelization platform called Hyracks as the foundation. The project aims at developing a parallel processing framework over different levels as an alternative to Hadoop as the authors claim different flaws in the architecture of the Hadoop stack.

Furthermore, there are several publications to describe best practices and patterns for 'big data' systems, e.g. by Kimball [143, 144]. Marz and Warren [164] also describes an architecture which he calls Lamba Architecture. What he describes is however more a general pattern how to structure an architecture with based on the principles of immutability and human fault-tolerance. The architecture is merely based on technical consideration and characteristics and also does not incorporate a functional view along the data processing pipeline. It provides however a very useful pattern for separating high-latency batch processing from processing data for low-latency requirements and a corresponding distribution of data storage with the aim to isolate and reduce complexity. These best practices and patterns will be incorporated into my 'big data' reference architecture in Chapter 4.

3 Requirements framework

To build the reference architecture on solid ground and to make evidence-based instead of arbitrary design decisions it is necessary to base it on a set of requirements. Extracting, formulating and specifying requirements helps understanding a problem in detail and is therefore a pre-requisite to design a solution for that problem. This specification is part of this Chapter. In Section 3.1 I will first define the term requirement and then describe how requirements can be described and how they can be structured. I will also develop a specification template. Afterwards, in Section 3.2, I will use this template to describe requirements the reference architecture needs to tackle. These will later be used in Chapter 4 to base design decisions on them.

3.1 Requirements Methodology

Motivation for specifying requirements
Understanding the problem and its requirements is important for designing a solution, but also for users afterwards to understand the solution, which problem it solves and how to apply it. To make matters more concrete, in this case the problem is handling different types of 'big data' and the solution is a reference architecture. At design time of this reference architecture, requirements help to understand ''big data', what challenges it creates, to focus on the problem and to reason about design decisions. Of course, a reference architecture is an abstract construct and needs to be applicable in different situations, where it gets realized in the form of a concrete architecture for that particular situation. This part of realizing a concrete architecture is the application. When applying it in a concrete context, the requirements description helps to identify if the reference architecture is applicable and feasible for that context, but they can also serve as an inspiration or input for the concrete requirements for the respective situation. While a reference architecture is typically broad to be applicable in a variety of situations, a concrete architecture project normally selects necessary parts and only implements those. Therefore, a subset of these abstract requirement set can be chosen and the individual requirements be concretised by filling placeholders to match the concrete project situation. Placeholders will be referred to in the requirements description by <p1>, <p2> etc.

Definition of the term 'requirement'
Requirements specify what a system should do, how it should behave, which qualities it should show along the way and within which borders or constraints this behaviour should take place to provide value. In this sense requirements are often categorised into functional requirements[1], non-functional

[1] actions the system performs

requirements[2] and constraints on the development process or the design of the system [146, pp. 6-7][227, pp. 7-12][189, pp. 1-11].

Scope of the requirements specification
One thing to note is, that requirements are about what to implement and not how to implement it. The later is about architecture and design decisions. Therefore the need for scalability e.g. creates a non-functional requirement, while the usage of parallel processing to ensure scalability is not a requirement but an architectural decision.

Another point worth mentioning is, that functional requirements are often at the application level, that is they are about what functionality a system should deliver to the end-user. A functional requirement might e.g. be to calculate a sentiment score for a certain product based on tweets. As the reference architecture should not be application-specific, but focus on the infrastructure and support different application scenarios on top, functional requirements on the application level will not be part of this specification. However, there will be functional requirements on the infrastructure level, such as to manage metadata throughout the data processing steps.

Furthermore requirements should typically be very specific and measurable. This is however not feasible in the context of a reference architecture. The exact measure or fit criterion to judge fulfilment of a requirement is very situation dependent. While it is important for the reference architecture to e.g. specify a requirement to ensure latency, it is very application dependent if the latency needs to be within microseconds or if even 5 seconds are sufficient. Therefore, exact measures and fit criteria also need to be omitted.

Structure of the requirements specification
As explained, requirements for a reference architecture are typically more abstract. Therefore it is not feasible to exactly adopt a template for requirements specification[3]. I used the Volere template as an inspiration, but adjusted the attributes used to describe individual requirements. The specification will be structured hierarchically for clarity reasons. First, the requirements will be organized below five high-level goals. These goals are directly derived from the characteristics of ''big data' described in Chapter 2.1 and are therefore:

VOL Handle data volume

VEL Handle data velocity

VAR Handle data variety

VER Handle data veracity

VAL Create data value

All requirements will be grouped by those high-level goals and will be identified accordingly, e.g. as 'VOL1'. Of course, sometimes a requirement might support several of those goals. In that case, all related goals will be listed, but the requirement identifier will follow the one, where the relation is strongest. Furthermore, requirements themselves can be hierarchical, that is a requirement can be supported by several sub-requirements. The identifier of those sub-requirements will be grouped accordingly. A sub-requirement of 'VOL1' might e.g. have the identifier 'VOL1.1'.

Beyond this structure requirements can have three more relationship structures. One is a link to supporting literature, that is a reference to literature which mentions the requirement. The second relationship structure are dependencies on other requirements. Those other requirements do not need to be within the same hierarchy. If a requirement is dependent on another this means that they

[2] qualities or properties of the system
[3] e.g. the Volere Template [189, pp. 393-472]

are either complementary or even strongly dependent, that is one cannot be implemented without implementing the other. The third relationship structure are conflicts with other requirements. If a requirement creates a conflict to another requirement, implementing one of them makes it harder or even impossible to implement the other. These relationships help deepening the understanding for the problem on hand and how different aspects play together. Once a concrete architecture has been developed, they also help to guide requirements evolution. If a certain requirement changes they point to related requirements that might also need to be adjusted or at least thought about.

This leads to the following attributes for individual requirements:

Req. ID:	Identifier of the requirement	**Req. Type:**	Functional / Non-functional
Parent Req.:	Parent requirement in the requirements hierarchy (if any)	**Goals:**	High-level goals the requirement is supporting
Description:	A one-sentence specification of the requirement, including placeholder values for concretization		
Rationale:	Reasoning or justification for the requirement		
Dependencies:	Dependencies to other requirements and their type (complementary / strongly dependent)		
Conflicts:	Conflicts to other requirements and their type (competing / impossible)		
Literature:	References to literature that supports this requirement		

3.2 Requirements Description

This section will now instantiate the requirements template with concrete requirements and is structured after the overarching goals to handle data volume, data velocity, data variety and data veracity as well as to create value as described in Chapter 3.1. Each section starts with an overview diagram of the requirements categorized under the respective goal, and their dependencies and conflicts with each other, but also with requirements corresponding to other goals. The relationship between requirements get displayed as plus, equal or minus signs. A plus sign indicates complementary requirements, that is implementing one of the requirements will make it easier or support to fulfil the other one. An equal sign shows that two requirements are strongly dependent on each other and variables specified in both should be synchronized. It e.g. makes no sense to built a system, that can extract data of a certain format, but cannot process or store it. Finally, a minus sign refers to competing requirements, that is implementing one of them will make it harder to implement the other one. For such requirements there is typically the need to establish some trade-off. Requirements are furthermore coloured to depict their categorization. See Figure 3.1 for a legend of those colours. The left side of the figure shows the colours of the overarching goals, while the right side shows the colours of related requirements. Afterwards the different requirements will be listed and described in more detail.

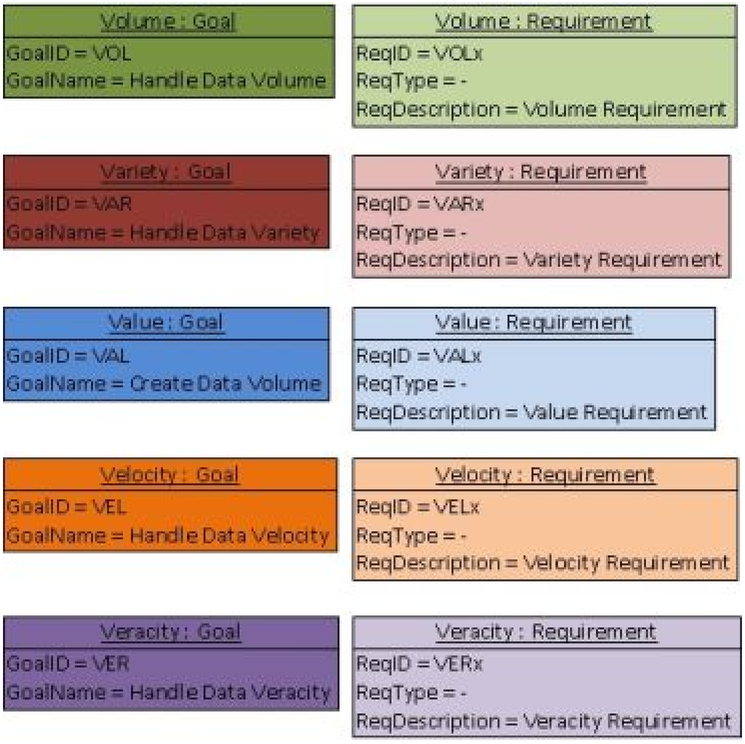


Figure 3.1: Requirements Visualization: Legend

3.2.1 Requirements aimed at Handling Data Dolume

Volume Requirements Overview

Handling a growing volume of data obviously adheres to the ability to store that data[4] and to process it with the aim of getting valuable results. While requirements specifying the concrete processing functionality are mainly part of Chapter 3.2.5 and described in the context of creating value, some requirements need to be fulfilled to enable processing in the advent of data value. First, the system should provide the necessary query performance[5], that is responding to queries within a certain time frame or with a certain latency, given a static data volume. Second, the system needs to be scalable[6], in the sense that it allows to add additional resources, to keep that query performance stable if the data volume grows [45, 132, 141].

Storage, performance and scalability all influence each other. If the system stores and uses a lot of data, this volume makes it harder to provide a reasonable query performance, while scalability supports meeting the performance requirement in the case of larger data volume stored in the system. This also means, that a large data volume and a high-volume storage requirement put pressure onto the scalability requirement, as the system needs to be more scalable to process that amount of stored data.

[4] see requirement VOL1
[5] see requirement VOL3
[6] see requirement VOL2

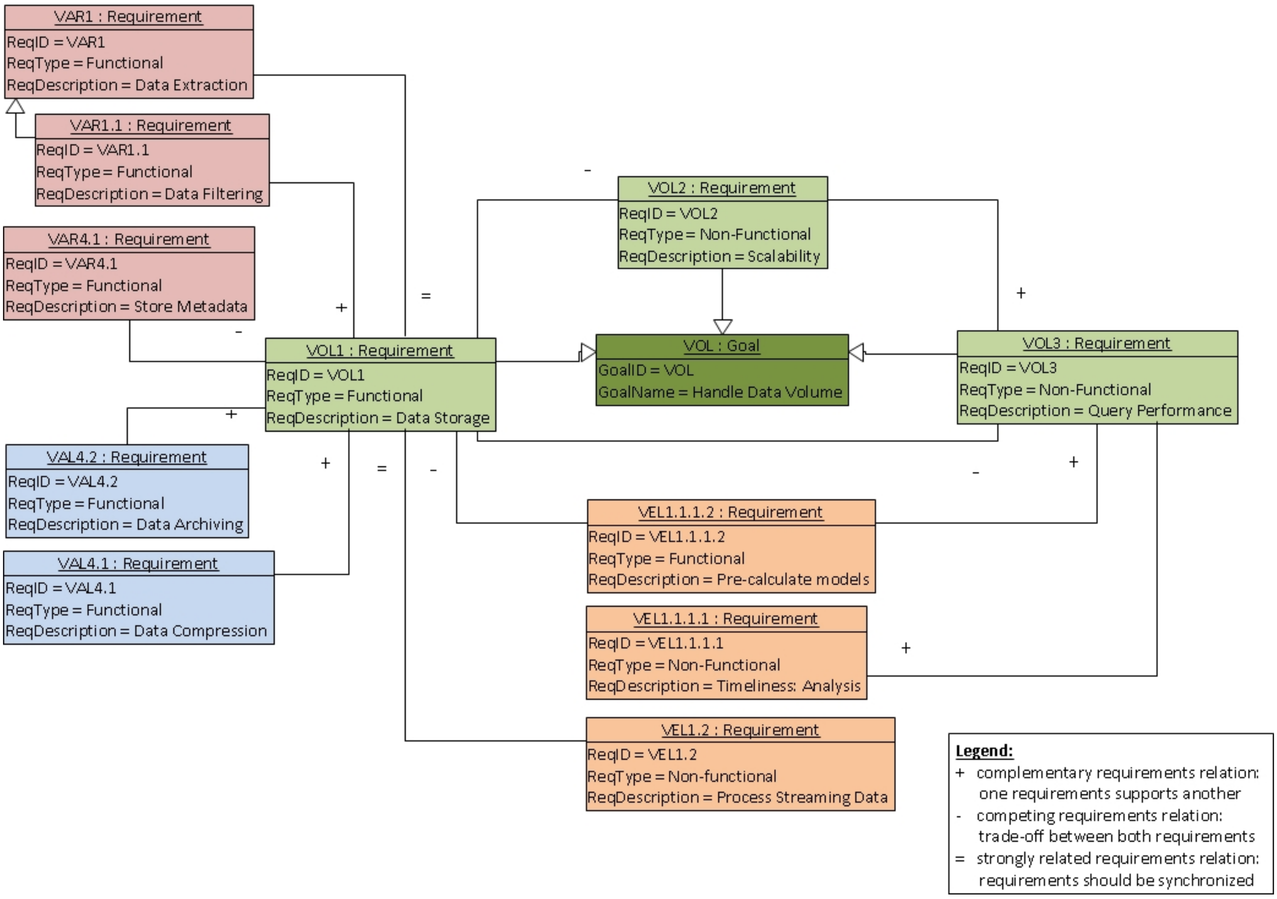


Figure 3.2: Requirements Visualization: Data Volume View

Furthermore, there is typically a trade-off between the required amount of storage and administrative or system management requirements. The later often require to store additional metadata and therefore increase the storage need. Another factor is the pre-calculation of intermediate results, rules and models, which can again help to increase the query performance, but also to handle velocity requirements. The later is also true for an improvement of performance in itself.

On the other hand, it is also possible to conduct measures and formulate respective requirements to help managing and decreasing the necessary storage. This is typically part of the notion of data lifecycle management, mainly data archiving and data compression. Another option is to filter out unnecessary data during the extraction process and only store what is needed. In general, one should note, that the requirements for data extraction and data storage should be synchronized. This is especially true because data storage is labelled as a volume requirement, but also has a variety component. In most cases it does not make sense to require the storage of particular data formats if data sources of this format are not to be extracted. There might be exceptions, e.g. data that gets extracted and is directly processed for an analytics task without this data be persisted in the meantime. However, these parameters should be synchronized and if there is a gap between both, there should be specific and documented reason for it.

Volume Requirements Specification

Table 3.1: Requirement VOL1 - Storing the Data

Req. ID:	VOL1	**Req. Type:**	Functional: Data Storage
Parent Req.:	-	**Goals:**	VOL,VAR
Description:	The system shall store data up to a volume of <p1: specify data volume> in the following formats <p2: specify required data formats>.		
Rationale:	This requirement is directly related to data volume and data variety as described in Chapters 2.1.2 and 2.1.4. Obviously, a system aiming at analysing large amounts of data also needs to store this data and the results. If data storage involves a lot of different formats, it makes sense to use this as a general requirement and create a sub-requirement for each data format that needs to be stored.		
Dependencies:	VAR1: Extracted data obviously needs to get stored. The formats specified in both requirements should be consistent. VAR1.1: Filtering out data decreases the storage need. VAL4.1: Compressing data decreases the data volume and therefore the amount of storage needed. VAL4.2: Archiving data decreases the data volume and therefore the amount of storage needed. VEL1.2: Similar to the extraction of data-at-rest, streaming data acquired for later used needs to get stored. If streaming data needs to get acquired, requirements should be synchronized and the respective data format regarded for storage.		
Conflicts:	VOL2,VOL3: The more data needs to be stored, the more scalable the system needs to be to handle the data and the harder it gets to provide performance. VAR4.1: Storing additional metadata increases the storage need. VEL1.1.1.2: Pre-calculated intermediate results, models or rules need to be stored and therefore increase the storage volume required.		
Literature:	-		

Table 3.2: Requirement VOL2 - Scaling with Growing Data Volume and Workload

Req. ID:	VOL2	**Req. Type:**	Non-functional: Scalability
Parent Req.:	-	**Goals:**	VOL, VEL
Description:	The system shall be scalable, in the sense that the processed data volume per time unit can be improved by adding hardware resources while making use of the additional resources in a linear manner with a factor of at least <p1: define scaling factor>.		
Rationale:	Fulfilling this requirement ensures, that the system can be enhanced with hardware resources to keep the response time constant on the level specified in VEL1, while the data volume is increasing over time. As described in Chapter 2.1.2, data volume is expected to be growing and it is necessary to efficiently (see the scaling factor) scale the system with that data volume.		
Dependencies:	VOL3: Scalability allows to keep performance while growing the data volume		
Conflicts:	VOL1: The more data needs to be stored, the more scalable the system needs to be to handle the data.		
Literature:	[45, 132, 141]		

Table 3.3: Requirement VOL3 - Providing Sufficient Performance when Answering Queries

Req. ID:	VOL3	**Req. Type:**	Non-functional: Query Performance
Parent Req.:	-	**Goals:**	VOL, VEL
Description:	The system shall respond to a query that involves <p1: define amount of data> within a response time of <p2: define response time>, while the system runs on <p3: define base hardware configuration>.		
Rationale:	This requirement also supports timeliness mentioned in Chapter 2.1.3, but it refers to query performance in general. While timeliness refers to analysing and getting results directly when data flows in, query performance is also applicable for batch processing of data and processing stored data at later point in time. Performance is of course, one of the supporting factors to achieve timeliness, but reasonable performance is also necessary for batch and ad-hoc processing to ensure user satisfaction. If necessary VOL3 can be decomposed into sub-requirements, which specify necessary query performance dependent to the analysis tasks.		
Dependencies:	VOL2: Ensuring a constant response time with growing data volume requires scalability VEL1.1.1.1: Query performance supports the timely analysis of inflowing data. VEL1.1.1.2: Pre-calculated intermediate results, models or rules can, if they are applicable, improve the performance of queries in general. VAL2.3: The abstraction away from implementation details in declarative query languages teypically allows query translation and execution in those languages to be highly optimized, both logically and physically. Therefore it frees programmers from doing this optimization in lower level code and prevents the usage of unoptimized code.		
Conflicts:	VOL1: The more data needs to be stored, the harder it gets to provide performance.		
Literature:	-		

3.2.2 Requirements aimed at Handling Data Velocity

Velocity Requirements Overview

Requirements aiming at velocity mainly tackle the idea of stream processing [62], that is to process data directly while it flows in. The challenge here lies in the speed of the incoming data and is therefore best reflected by a couple of non-functional requirements, which pose constraints on the rate of processing of incoming data. As described in Section 3.2.2, there can be distinguished between two parts of that challenge.

One is to create insights from the inflowing data and react on it in time [166, 213]. That is the timeliness challenge [7] [45]. This can be broken down into two phases of the feedback loop, analysing inflowing data[8] and reacting according to this insights [9] [11, 231]. As the reference architecture presented in this thesis focusses on the analytical site of 'big data', the reaction itself will be out of scope and part of an operational system. It is however required to communicate with that system and to trigger the reaction. Handling the timeliness challenge typically does not allow to do a deep analysis

[7] see requirement VEL1.1 and sub-requirements

[8] see requirements VEL1.1.1 and sub-requirements

[9] see requirement VEL1.1.2 and sub-requirement

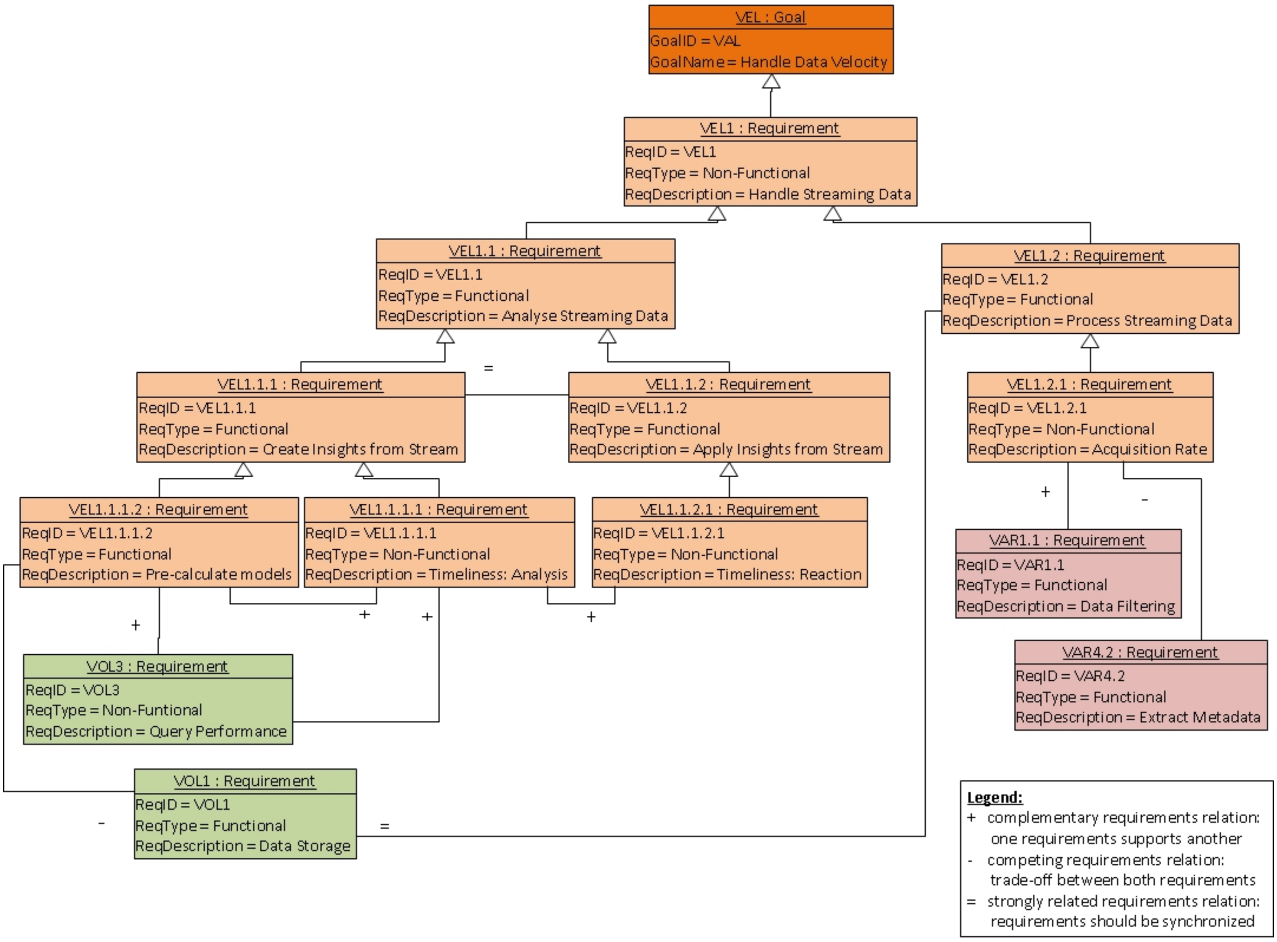


Figure 3.3: Requirements Visualization: Data Velocity View

and compare inflowing data against a large amount if historical data, but requires the pre-calculation of models or rules the streaming data can be matched against[10].

The other challenge is handling the acquisition rate[11]. This refers to acquiring data from data streams and storing it in the system [45, 147, 213] and typically to processing a large amount of rather small transactions while maintaining a persistent state.

There is a natural interaction between velocity and volume requirements. It is intuitively clear, that the data volume gets bigger, the faster data flows in. The overarching goals of volume and velocity are therefore already interwoven. On the requirements level this leads to an obvious conflict between the acquisition rate and data storage. The higher the acquisition rate, the more data will be stored over time and the harder it will get to store all of them. The pre-calculation of models also slightly stretches the storage requirements, as those models need to be stored. On the other hand, these models can improve the general query performance if they are applicable during the query processing. Furthermore, the acquisition rate challenge can be made easier, by filtering out data and decreasing the effective inflow rate, that is the rate of inflowing data that actually needs to be stored. On the other hand, requiring the extraction of metadata during the inflow takes time, slows down the process and makes it harder to confirm with the required acquisition rate.

[10] see requirement VEL1.1.1.2

[11] see requirement VEL1.2 and sub-requirement

Velocity Requirements Specification

Table 3.4: Requirement VEL1 - Handling Streaming Data while it is flowing in

Req. ID:	VEL1	**Req. Type:**	Non-Functional: Handle Streaming Data
Parent Req.:	-	**Goals:**	VEL
Description:	The system shall handle data while it is flowing in with a rate of up to <p1: specify inflow rate>.		
Rationale:	This requirement is directly related to data stream processing and handling velocity as described in Chapter 2.1.3. It is a parent requirement to contain timeliness and acquisition rate as the main challenges of stream processing.		
Dependencies:	-		
Conflicts:	-		
Literature:	[11, 45, 62, 83, 92, 141, 147, 166, 213, 231]		

Table 3.5: Requirement VEL1.1 - Reacting on-time to Streaming Data

Req. ID:	VEL1.1	**Req. Type:**	Functional: Analyse Streaming Data
Parent Req.:	VEL1	**Goals:**	VEL, VOL
Description:	The system shall conduct analysis tasks on streaming data, create insights as specified in VEL1.1.1 and react to them as specified in VEL1.1.2 while data is flowing in with a rate as specified in VEL1.		
Rationale:	This requirement is directly related to the timeliness problem and the feedback loop as described in Chapter 2.1.3. It is an overarching requirement, which contains the analysis of streaming data and the reaction to it and represents the feedback loop as a whole. It allows to directly react to in-flowing business transactions and increase business agility.		
Dependencies:	-		
Conflicts:	-		
Literature:	[11, 45, 62, 92, 166, 213, 231]		

Table 3.6: Requirement VEL1.1.1 - Creating Insights from Streaming Data

Req. ID:	VEL1.1.1	**Req. Type:**	Functional: Create Insights from Data Streams
Parent Req.:	VEL1.1	**Goals:**	VEL, VOL
Description:	The system shall conduct analysis tasks on streaming data and create insights as specified in <p1: specify functional requirements that describe the actual data analysis steps> while data is flowing in.		
Rationale:	This requirement is directly related to the timeliness problem as described in Chapter 2.1.3. It addresses the first part of the feedback loop, that is to timely create insights on inflowing data and is a parent requirement to functional requirements <p1>, which specify the single steps and methods for analysing streaming data.		
Dependencies:	VEL1.1.2: Creating insights from streaming data is just one part of the feedback loop. These insights need to be applied and reacted on. <p1>: Add dependencies to the functional requirements that specify the analysis tasks		
Conflicts:	-		
Literature:	[11, 45, 62, 92, 166, 213, 231]		

Table 3.7: Requirement VEL1.1.1.1 - Creating Insights from Streaming Data - Timeliness

Req. ID:	VEL1.1.1.1	**Req. Type:**	Non-Functional: Timeliness Analysis
Parent Req.:	VEL1.1.1	**Goals:**	VEL, VOL
Description:	The system shall create insights from streaming data as specified in VEL1.1.1 within a time frame of <p1: specify time acceptable for generating insights> while data is flowing in with a rate as specified in VEL1.		
Rationale:	This requirement is directly related to the timeliness problem as described in Chapter 2.1.3. It addresses the first part of the feedback loop, that is to timely create insights on inflowing data and specifies the time horizon that is acceptable for creating those insights.		
Dependencies:	VOL3: Timely analysis of streaming data gets supported by query performance in cases where it involves querying existing data within the system VEL1.1.1.2: Pre-calculated models or rules can directly be applied to inflowing data and rapidly improve the analysis time compared to analysing it against all stored data. VEL1.1.2.1: The timely analysis and insight creation from streaming data is a necessary prerequisite for reacting on streaming data in-time, the faster insights are created they faster they can be applied.		
Conflicts:	-		
Literature:	[11, 45, 62, 92, 166, 213, 231]		

Table 3.8: Requirement VEL1.1.1.2 - Creating Insights from Streaming Data - Pre-computation

Req. ID:	VEL1.1.1.2	**Req. Type:**	Functional: Pre-calculate models
Parent Req.:	VEL1.1.1	**Goals:**	VEL, VOL
Description:	The system shall pre-compute the following rules and models <p1: specify models to pre-compute> from the persistent data available in the system and apply these models to process streaming-in data.		
Rationale:	The analysis of inflowing data often involves comparing the new data to a large amount of similar, historical data. It is not feasible to do this in real-time. Therefore it is necessary to create rules or models that are applicable to the streaming data to create the required insights, but also to create suitable index structures to quickly find relevant historical data for comparison. This combines analysis at-rest to process large amounts of available data to compute applicable models with real-time analysis to apply these models to inflowing data.		
Dependencies:	VEL1.1.1.1: Pre-calculated models or rules can directly be applied to inflowing data and rapidly improve the analysis time compared to analysing it against all stored data. VOL3: Pre-calculated intermediate results, models or rules can, if they are applicable, improve the performance of queries in general.		
Conflicts:	VOL1: Pre-calculated intermediate results, models or rules need to be stored and therefore increase the storage volume required.		
Literature:	[45][231, pp. 9-14]		

Table 3.9: Requirement VEL1.1.2 - Using Insights from Streaming Data

Req. ID:	VEL1.1.2	**Req. Type:**	Functional: Apply Insights from Data Streams
Parent Req.:	VEL1.1	**Goals:**	VEL, VOL, VAL
Description:	The system shall communicate the insights from VEL1.1.1 as specified in <p1: specify functional requirements that describe to which systems or recipients and in which format insights should be communicated>.		
Rationale:	This requirement is directly related to the timeliness problem as described in Chapter 2.1.3. It addresses the second part of the feedback loop, that is to react to inflowing data in (near) real-time. Note, that the actual reaction to inflowing data will not happen within the analytical system, which is in scope of this reference architecture, but the insights will be sent back to operational systems or as alerts to employees to be handled. The actual reaction is out of scope of this reference architecture. This requirement is a parent requirement to functional requirements <p1>, which specify formats, steps and partners of the communication.		
Dependencies:	VEL1.1.1: To communicate, apply and react on insights from streaming data, these insight obviously need to be created first. <p1>: Add dependencies to the functional requirements that specify how, in which formats and to which systems / recipients to communicate the insights		
Conflicts:	-		
Literature:	[11, 45, 62, 92, 166, 213, 231]		

Table 3.10: Requirement VEL1.1.2.1 - Using Insights from Streaming Data - Timeliness

Req. ID:	VEL1.1.2.1	**Req. Type:**	Non-Functional: Timeliness Reaction
Parent Req.:	VEL1.1.2	**Goals:**	VEL, VOL, VAL
Description:	The system shall communicate the insights from VEL1.1.1 as specified in VEL1.1.2 within a time frame of <p1: specify time acceptable for communicating insights> while data is flowing in with a rate as specified in VEL1.		
Rationale:	This requirement is directly related to the timeliness problem as described in Chapter 2.1.3. It addresses the second part of the feedback loop, that is to timely react to insights on inflowing data and specifies the time horizon that is acceptable for communicating those insights to operational systems.		
Dependencies:	VEL1.1.1.1: The timely analysis and insight creation from streaming data is a necessary prerequisite for reacting on streaming data in-time, the faster insights are created they faster they can be applied.		
Conflicts:	-		
Literature:	[11, 45, 62, 92, 166, 213, 231]		

Table 3.11: Requirement VEL1.2 - Acquiring and Processing Streaming Data

Req. ID:	VEL1.2	**Req. Type:**	Functional: Process Streaming Data
Parent Req.:	VEL1	**Goals:**	VEL, VOL
Description:	The system shall process and acquire data as specified in <p1: specify functional requirements that describe how and which parts of the streaming data should be acquired and pre-processed> and store it.		
Rationale:	This requirement is directly related to the acquisition rate challenge as described in Chapter 2.1.3. It allows to acquire very fast inflowing data for later use and to create a basis for large OLTP-like workload. This requirement is a parent requirement to functional requirements <p1>, which specify acquisition and pre-processing of streaming data before it gets stored. Note that pre-processing only refers to steps that are necessary before storing the data for later analytical use and not to operational processing of streaming data, which is out of scope of this reference architecture.		
Dependencies:	VOL1: Similar to the extraction of data-at-rest, streaming data acquired for later used needs to get stored. If streaming data needs to get acquired, requirements should be synchronized and the respective data format should be regarded for storage.		
Conflicts:	-		
Literature:	[45, 83, 141, 147, 213]		

Table 3.12: Requirement VEL1.2.1 - Acquiring and Processing Streaming Data - Acquisition Rate

Req. ID:	VEL1.2.1	**Req. Type:**	Non-Functional: Acquisition Rate
Parent Req.:	VEL1.2	**Goals:**	VEL, VOL
Description:	The system shall acquire, pre-process and store streaming data as specified in VEL1.2 with an acquisition rate of <p1: specify acquisition rate> while data is flowing in with a rate as specified in VEL1.		
Rationale:	This requirement is directly related to the acquisition rate challenge as described in Chapter 2.1.3 and could also be referred to as ‘write performance’. It specifies the necessary acquisition rate. The acquisition rate should typically be equal to the rate of in-flowing data or smaller in cases where not all in-flowing data needs to be acquired.		
Dependencies:	VAR1.1: Filtering out data decreases the amount of data to store and therefore decreases the necessary acquisition rate and write performance		
Conflicts:	VAR4.2: Extracting additional metadata during the data acquisition process decreases prolongs the acquisition process and makes it harder to achieve the acquisition rate.		
Literature:	[45, 83, 141, 147, 213]		

3.2.3 Requirements aimed at handling data variety

To not mess up the requirements visualization and keep it somehow simple, it is split up into two parts. The first part describes the requirements that aim at extracting information of varied sources and integrating it[12]. The second part describes managing metadata to describe the data and sources and to keep them manageable and understandable[13].

Variety Requirements Overview - Part 1

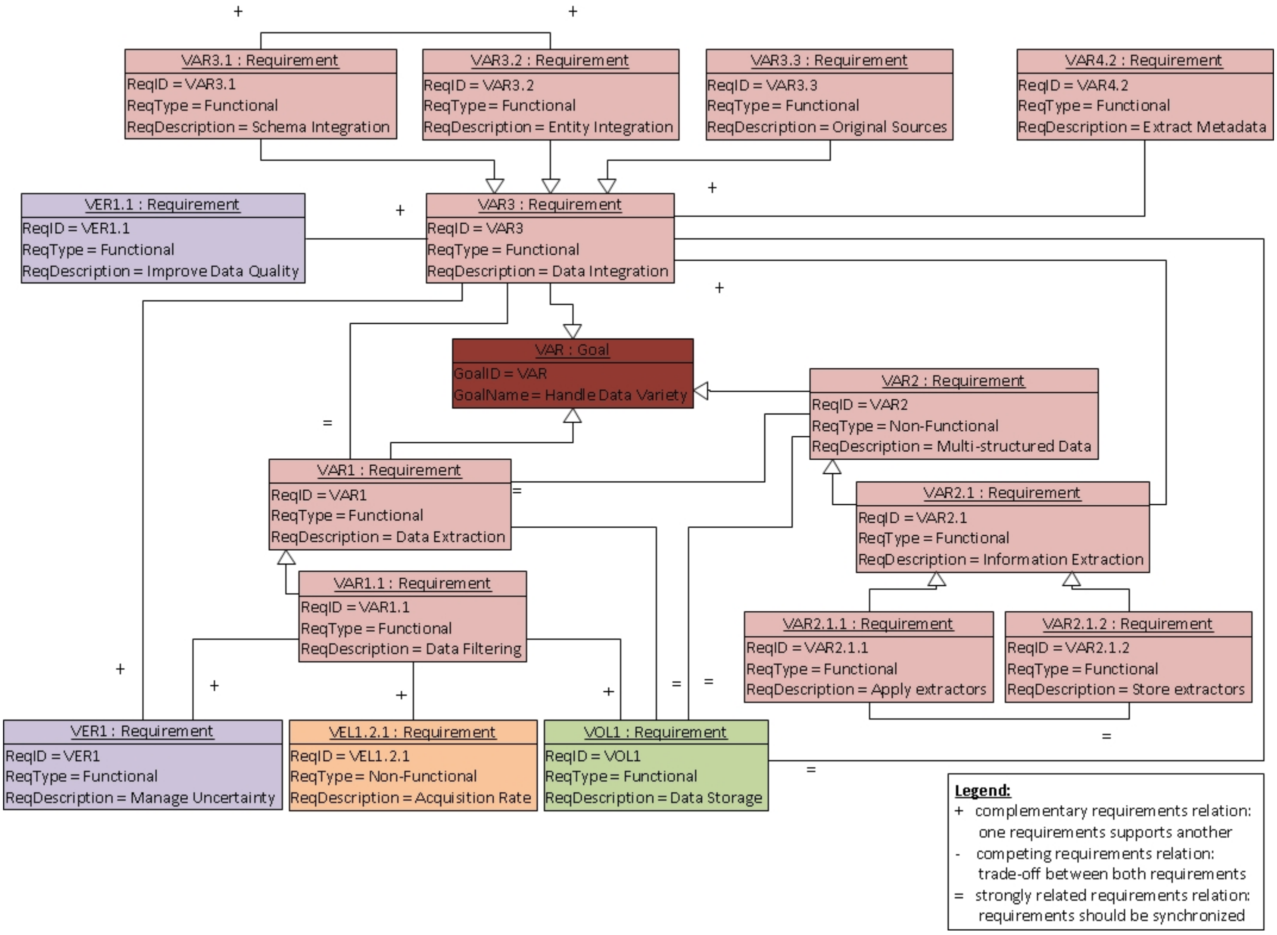


Figure 3.4: Requirements Visualization: Data Variety View 1 - Information Extraction and Integration

As described in Chapter 2.1.4, variety refers to deriving information from several, heterogeneous data sources[45, 83, 166][202, pp. 10-12,143-209]. Doing this can be regarded as a 3-step process. First, data needs to get extracted from these sources and loaded into the 'big data' system[14]. One important measure to consider for data extraction, is data filtering[15][45]. This can help to satisfy several other requirements, namely to decrease the storage needs, to decrease the acquisition rate needs and to decrease uncertainty.

[12] see Figure 3.4

[13] see Figure 3.5

[14] see requirement VAR1

[15] see requirement VAR1.1

As a second step, in the case of unstructured or sometimes semi-structured data[16], there is a need to impose some structure, that can later be used for analysing the data[17]. Information extraction is the process of extracting this structure, e.g. in the form of entities mentioned in a text or a sentiment expressed by the author [45, 46, 104]. Typically there is a need to manage and apply several extractors to extract different kinds of information[18]. Identifying the sentiment of a text can be quite different from resolving entitities [45, 46].

The third step is to integrate data from different sources, so they can be queried and analysed together to create overarching insights, that are not available from analysing different sources in isolation. On a higher level this means integrating multi-structured data[83, 141], after the necessary information extraction has been done as a pre-step. On a lower level, this means the general integration of heterogeneous data based on schema- and entity integration[19]. Schema integration refers to defining a general, overarching schema and mapping the different data sources to it. This includes mappings and transformations on the field level, e.g. splitting a ‘’name’ field into two fields ‘first name’ and ‘surname’ or calculation a field ‘’salary before taxes’ to ‘salary after taxes’. There is also a great dependency between schema integration and metadata management. Schema integration requires high quality metadata and information about the schemas of the different data sources to finally map those schema onto an integrated one. Entity integration refers to identifying unique entities on the row level and collapsing information that are spread over several rows and data sources, but are actually about the same entity or object. Examples are collapsing information about two people ‘M. Maier’ and ‘’Markus Maier’ or identifying that two keys refer to the same entity based on a mapping table.

As described in Chapter 2.1.4, it is however important to note, that it can be necessary to allow data analysts to access the raw source data[20]. If this is the case, raw data needs to be stored in the system in the same form as it gets extracted from the sources. The decision is then, either to build a virtual, integrated schema, which can be queried and transparently transforms the query into sub-queries onto the different raw data sources based on a set of mapping and transformation rules, or to run information extraction and integration tasks as a regular batch job and store a persistent, integrated schema additionally to the raw data. This is obviously a trade-off between a decreased storage need (first option) and a performance gain for all analysis tasks, that can directly run on the integrated schema (second option).

Variety Requirements Specification - Part 1

Table 3.13: Requirement VAR1 - Extracting Data from Varied Sources

Req. ID:	VAR1	**Req. Type:**	Functional: Data Extraction
Parent Req.:	-	**Goals:**	VAR
Description:	The system shall extract data in the following formats <p1: specify required data formats> from the following sources <p2: specify required data sources>.		

[16] see requirement VAR2
[17] see requirement VAR2.1
[18] see requirements VAR2.1.1 and VAR2.1.2
[19] see requirements VAR3.1 and VAR3.2
[20] see requirement VAR3.3

Rationale:	This requirement is directly related to data variety as described in Chapter 2.1.4. The requirement specifies the sources the system needs to acquire data from and the formats of this data. Acquiring data is the obvious pre-step of every data analysis. If data extraction involves a lot of different formats and sources, it makes sense to use this as a general requirement and create a sub-requirement for each source from which data needs to be acquired.		
Dependencies:	VOL1: Data that gets extracted should obviously be able to be stored. The formats specified in both requirements should be consistent.		
Conflicts:	-		
Literature:	[45, 83, 166][202, pp. 10-12,143-209]		

Table 3.14: Requirement VAR1.1 - Filtering Data during the Extraction

Req. ID:	VAR1.1	**Req. Type:**	Functional: Data Filtering
Parent Req.:	VAR1	**Goals:**	VOL,VEL,VAR,VER
Description:	The system shall filter out data extracted from sources based on the following rules <p1: specify filter rules>		
Rationale:	It is often the case, that only parts of the data from a source are needed or valuable for the analysis. In this cases it is reasonable to filter out unnecessary parts of the data. This can help to decrease the storage need and handle the acquisition rate challenge. It can also be reasonable to filter out very untrustworthy data.		
Dependencies:	VOL1: Filtering out data decreases the storage need VEL1.2.1: Filtering out data decreases the amount of data to store and therefore decreases the necessary acquisition rate and write performance VER1: Filtering out untrustworthy data can help to decrease the total uncertainty of the data within the system		
Conflicts:	-		
Literature:	[45][203, pp. 138-139]		

Table 3.15: Requirement VAR2 - Handling Data in Multiple Structures and Formats

Req. ID:	VAR2	**Req. Type:**	Non-Functional: Multi-structured data
Parent Req.:	-	**Goals:**	VAR
Description:	The system shall handle multistructured data in the following formats <p1: specify required data formats> and from the following source <p2: specify required data sources>, where handling means to store them, manage them, extract the necessary information and apply analysis tasks as specified in the other requirements.		
Rationale:	This requirement is directly related to data variety as described in Chapter 2.1.4 and adresses the diversity of structures. It is a parent requirement for several other requirements that tackle the challenge of data formats with different levels of structure. If a lot of different formats are involved, it makes sense to use this as a general requirement and create a sub-requirement for each source from which data needs to be acquired. <p1> can refer to structured, semi-structured (e.g. XML files possibly connected to a schema definition, but also concrete HTML pages) and unstrutured (text, video and audio) data sources.		

Dependencies:	VOL1: Handling multi-structured data is the more general requirement, while the storage of the data is one of the necessary requirements to accomplish that. The formats specified in both requirements should be consistent. VAR1: It obviously makes only sense to extract data if it can also be handled. The formats specified in both requirements should be consistent.
Conflicts:	-
Literature:	[45, 46, 83, 130, 141, 166]

Table 3.16: Requirement VAR2.1 - Extracting Information from Unstructured Data

Req. ID:	VAR2.1	**Req. Type:**	Functional: Information Extraction
Parent Req.:	VAR2	**Goals:**	VAR, VAL
Description:	The system shall extract (additional) machine-readable information from the following formats <p1: specify data formats> and from the following source <p2: specify required data sources> and therefore impose structure onto this data.		
Rationale:	This requirement is directly related to data variety as described in Chapter 2.1.4 and is a sub-requriement for VAR2. To generally handle multi-structured data it is first necessary, to extract machine-processable information out of the data and set it in relation to impose some structure.		
Dependencies:	VAR2: Information extraction is a sub-requirement of generally handling multi-structured data. <p1> should be a subset of the data formats specified in VAR2, as there might be sources that do not require information extraction to be processed (e.g. relational, but also semi-structured data like XML files)		
Conflicts:	-		
Literature:	[45, 46, 104]		

Table 3.17: Requirement VAR2.1.1 - Applying Multiple Extractors for Information Extraction

Req. ID:	VAR2.1.1	**Req. Type:**	Functional: Apply extractors
Parent Req.:	VAR2.1	**Goals:**	VAR, VAL
Description:	The system shall host and manage several extractors using different information extraction techniques and algorithms and apply a configurable subset of them to data from different sources.		
Rationale:	As Agrawal et al. point out [46], it is to be expected that a variety of extractors will be used to extract information from one and the same source. This can be valuable to tune the information extraction to a particular data source or the analysis task at hand. It is also the case, that different information extractors aim at different types of information (e.g. entities, relationships or sentiment). It can make sense to create additional sub-requirements that list the necessary extractors and map them to data sources. This also depends on how closely data sources and information extractors are coupled and how flexible the linking should be.		
Dependencies:	VAR2.1: Information Extraction is one necessary step to handle multi-structured data.) VAR2.1.2: The information extracted by independent extractors needs to stored and accessible in an integrated manner.		
Conflicts:	-		
Literature:	[46]		

Table 3.18: Requirement VAR2.1.2 - Storing and Managing Multiple Extractors for Information Extraction

Req. ID:	VAR2.1.2	**Req. Type:**	Functional: Store extractors
Parent Req.:	VAR2.1	**Goals:**	VAR, VAL
Description:	The system shall store the information extracted by different extractors according to requirement VAR2.1.1 in a structured form in one integrated data model and related to the source data.		
Rationale:	Information extracted from semi- and unstructured source data, needs to be stored in an integrated way, so it is easier to keep an overview over the available information as well as how and from which source data it got created. This information also needs to be available for further analysis tasks and for end-users to search through.		
Dependencies:	VAR2.1: Information Extraction is one necessary step to handle multi-structured data.) VAR2.1.1: The information extracted by independent extractors needs to stored and accessible in an integrated manner.		
Conflicts:	-		
Literature:	[45, 46]		

Table 3.19: Requirement VAR3 - Integrating Data from Varied Sources

Req. ID:	VAR3	**Req. Type:**	Functional: Data Integration
Parent Req.:	-	**Goals:**	VAR, VER, VAL
Description:	The system shall integrate heterogeneous data from the following sources <p1: specify data sources> and with the following formats <p2: specify data formats> into an unified view.		
Rationale:	As described in Chapter 2.1.4 analysis tasks in the 'big data' environment often span data from several sources involving several data formats and schemas, to do this in a meaningful way it is necessary to impose a global schema on top of the different sources and to harmonize data semantics.		
Dependencies:	VAR1, VOL1: The data that gets integrated needs first to extracted and stored. The data sources and formats specified in VAR3 should therefore be a subset of those specified in VAR1 and VOL1, as it might be the case that not all extracted sources need to be integrated, e.g. if data from one source is only needed for a separated analysis task.) VAR2.1: Information extraction is a necessary pre-step for integrating multi-structured data. Unstructured sources and formats that are listed in VAR 3, should also be listed in VAR2.1. VER1: Data integration can resolve some of the issues of unertainty, e.g. bei harmonizing entities and adopting values from other data sources. VER1.1: An improved data quality by completing and correcting values can make it easier to do the data integration step.		
Conflicts:	-		
Literature:	[45, 67, 83, 128, 130, 169, 212]		

Table 3.20: Requirement VAR3.1 - Integrating the Schema of Data from Varied Sources

Req. ID:	VAR3.1	**Req. Type:**	Functional: Schema Integration
Parent Req.:	VAR3	**Goals:**	VAR, VER, VAL
Description:	The system shall maintain a global schema and a semantic mapping of the schemas of the different data sources specified in VAR3 onto that global schema. The global schema should be <p1: virtual / persisted>.		
Rationale:	As described in Chapter 2.1.4 and Chapter 2.1.6 data needs to be combined from different sources to provide overarching insights and value. However, even in case of structured or semi-structured sources, their structure or schema typically differs. A global schema is an unique schema, visible to the users, which is mapped to the schemas of the source data. This schema can be either virtual, that is queries to the global schema are translated according to the mapping and passed to the original data sources on runtime, or persistent, that is the original source data is transformed according to the mapping rules and stored in the global schema before runtime. In a concrete architecture, it can make sense to add requirements to fix the (most important parts of the) global schema and the (most important) mapping rules.		
Dependencies:	VAR2.1, VAR2.1.2: Extracting information and impose structure onto unstructured data is a necessary pre-step to integrate that data and map it to a global schema.) VAR3.2: If both, schema integration and entity integration, are to be persistent, they can use the same intermediate storage layer for their persistence.		
Literature:	[45, 73, 83, 128, 130, 169, 212]		

Table 3.21: Requirement VAR3.2 - Integrating Entities from Data from Varied Sources

Req. ID:	VAR3.2	**Req. Type:**	Functional: Entity Integration
Parent Req.:	VAR3	**Goals:**	VAR, VER, VAL
Description:	The system shall be able maintain rules to resolve and match similar entities, that is to identify tuples and field values that refer to the same entity and collapse them. This should be done <p1: virtually / persistent>		
Rationale:	As described in Chapter 2.1.4 data needs to be combined from different sources to provide overarching insights. However, it is very probable that different sources refer to the same entity or object with different names or keys. There can even be duplicates of objects which are referred to by different names or keys within a single source. The system should be able to recognize these and integrate tuples which refer to the same object into one tuple to provide an integrated and more consistent view onto this object. This can not only happen for identifiers of tuples, but also for attribute values within a tuple. It is also typical to happen for information extracted from unstructured data, as those data sources typically do not use a unique key to refer to an entity. The resolution can either be virtual, that is the rules are applied at runtime, or persistent, that is entities are collapsed before runtime and stored as a single tuple.		
Dependencies:	VAR2.1, VAR2.1.2: Extracting information and entities from unstructured data is a necessary pre-step to integrate these entities with objects from other sources.) VAR3.1: If both, schema integration and entity integration, are to be persistent, they can use the same intermediate storage layer for their persistence.		

Conflicts:	-
Literature:	[45, 67, 83, 110, 130, 212, 225]

Table 3.22: Requirement VAR3.3 - Providing Access to Original Source Data

Req. ID:	VAR3.3	**Req. Type:**	Functional: Original Sources
Parent Req.:	VAR3	**Goals:**	VAR, VAL
Description:	The system shall enable users and analysis tasks to directly access the original source data without any pre-processing or integration.		
Rationale:	In some cases, inconsistencies and noise in the data sources are part of the analysis task or can provide additional insights. In cases, where schema and entity integration are not persistent, it can sometimes also be necessary to avoid the virtual integration step for performance reasons. In those cases, the necessary amount of integration and handling connection between different data sources needs to be handled within the analysis task.		
Dependencies:	VAR3.1,VAR3.2: Accessing original sources means bypassing the functionality specified in schema and entity integration.)		
Conflicts:	-		
Literature:	[45, 83, 110, 130, 225]		

Variety Requirements Overview - Part 2

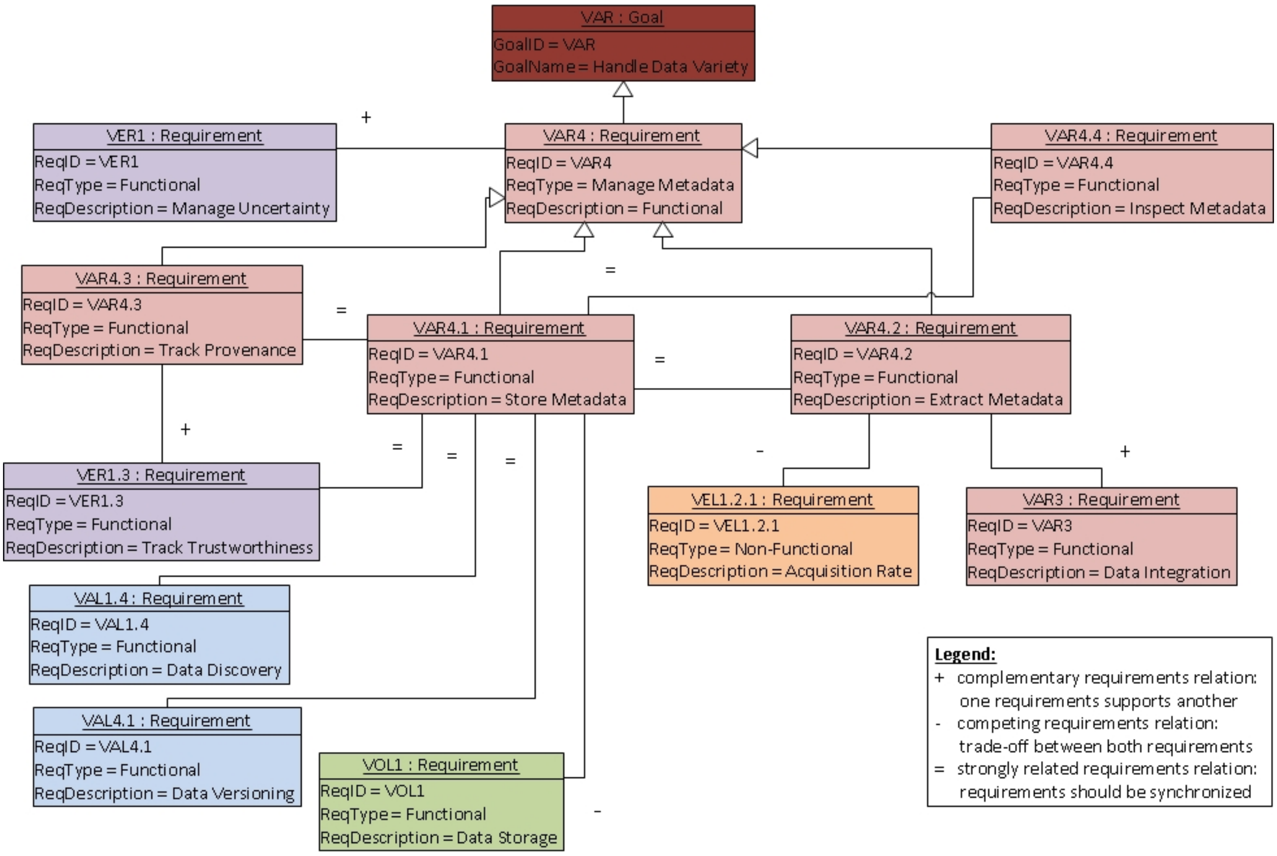

VAR : Goal
GoalID = VAR
GoalName = Handle Data Variety
VER1 : Requirement
ReqID = VER1
ReqType = Functional
ReqDescription = Manage Uncertainty
VAR4 : Requirement
ReqID = VAR4
ReqType = Manage Metadata
ReqDescription = Functional
VAR4.4 : Requirement
ReqID = VAR4.4
ReqType = Functional
ReqDescription = Inspect Metadata
VAR4.3 : Requirement
ReqID = VAR4.3
ReqType = Functional
ReqDescription = Track Provenance
VAR4.1 : Requirement
ReqID = VAR4.1
ReqType = Functional
ReqDescription = Store Metadata
VAR4.2 : Requirement
ReqID = VAR4.2
ReqType = Functional
ReqDescription = Extract Metadata
VER1.3 : Requirement
ReqID = VER1.3
ReqType = Functional
ReqDescription = Track Trustworthiness
VEL1.2.1 : Requirement
ReqID = VEL1.2.1
ReqType = Non-Functional
ReqDescription = Acquisition Rate
VAR3 : Requirement
ReqID = VAR3
ReqType = Functional
ReqDescription = Data Integration
VAL1.4 : Requirement
ReqID = VAL1.4
ReqType = Functional
ReqDescription = Data Discovery
VAL4.1 : Requirement
ReqID = VAL4.1
ReqType = Functional
ReqDescription = Data Versioning
VOL1 : Requirement
ReqID = VOL1
ReqType = Functional
ReqDescription = Data Storage
+
=
=
=
+
=
=
=
-
+
-
Legend:
+ complementary requirements relation: one requirements supports another
- competing requirements relation: trade-off between both requirements
= strongly related requirements relation: requirements should be synchronized

Figure 3.5: Requirements Visualization: Data Variety View 2 - Metadata Management

The management of metadata[21] is necessary to allow users to get an overview of available data in the system, to understand that data and its semantics, to work with it, but also to administrate the system as a whole [45, 96, 97]. It also helps to users to understand the certainty and quality of the data, to determine which questions they can ask and to estimate how much they can rely on a certain analysis result. Therefore metadata management supports the handling of uncertainty in the data. Furthermore, as mentioned above, it also supports the integration of data from different sources.

Metadata management mainly consists of four parts. Metadata obviously needs to get created during the normal data extraction process[22] and it needs to get stored in relation to the actual data [23]. Therefore, metadata also poses some drawbacks. It can slightly slow down the extraction process and conflicts with the acquisition rate challenge. It also increases the storage need. On the other hand, several information necessary to fulfil other requirements namely the tracking of trustworthiness, also get stored as metadata. Furthermore, metadata is necessary to enable users with a effective, ad-hoc discovery of data and data sources. However, metadata does not only need to be extracted and stored, it also needs to be collected along the whole data processing. It should be possible for users to track the data provenance and to track analysis results all the way back to identify from which data sources the involved data was extracted and which data integration, information extraction and other steps have been conducted on it[24] [45]. It should also be possible for administrators and users to inspect the stored metadata and modify it, if needed [25].

Variety Requirements Specification - Part 2

Table 3.23: Requirement VAR4 - Managing Metadata of Data from Varied Sources and during Processing it

Req. ID:	VAR4	**Req. Type:**	Functional: Manage Metadata
Parent Req.:	-	**Goals:**	VAR,VER,VAL
Description:	The system shall gather and store metadata to describe: • the data source structure, schema and recording method • the data structures used within the system • analysis techniques and processing steps within the system • for data items from which source they are from and which processing steps have already been conducted (data provenance) • operational information about the analysis processes		
Rationale:	As described in Chapter 2.1.4, the management of metadata is important to keep an overview of the data available to and within the system and its structure, it is important for users to put analysis results into context and to judge the reliability of those results, it is important for integrating data with different structure and schema, it is important to query developers to determine which data they can use and which semantics this data has and it is important for operators to administrate the system.		
Dependencies:	VER1: Managing metadata helps to handle uncertainty, as metadata describes the available data, its structure and how it got recorded and as it allows users to set the data into context and judge its trustworthiness.		

[21] see requirement VAR4 and sub-requirements
[22] see requirement VAR4.2
[23] see requirement VAR4.1
[24] see requirement VAR4.3
[25] see requirement VAR4.4

	VAL1.3.1: The data discovery and navigation through the available data and data sources relies on the stored metadata about this data and this data sources. The availability of metadata is a necessary prerequisite.		
Conflicts:	-		
Literature:	[45, 169, 202]		

Table 3.24: Requirement VAR4.1 - Storing Metadata Related to the Data it Describes

Req. ID:	VAR4.1	**Req. Type:**	Functional: Store Metadata
Parent Req.:	VAR4	**Goals:**	VAR,VER,VAL
Description:	The system shall store metadata using the following data structures <p1: describe format / schema of the structures> and directly related to the data or structures it describes.		
Rationale:	Metadata of different data and structures needs to get stored in an integrated way, so it is easier to administrate, navigate through and use the metadata. It should be directly linked to the data or structures it describes as typically both data and its metadata get used together.		
Dependencies:	VER1.3: When computing the metric to track trustworthiness it needs to be stored as metadata within the metadata structures. VAR4.2,4.3: Metadata automatically extracted from sources and during data provenance need to be stored within the metadata storage structures. VAR4.4: To inspect the metadata it is necessary to access the metadata storage structures. When metadata gets manipulated during the inspection, the changes need to be written back. VAL1.4: The data discovery and navigation through the available data and data sources relies on the stored metadata about this data and this data sources.		
Conflicts:	VOL1: Storing metadata additionally to the data itself increases the storage need.		
Literature:	[45, 169]		

Table 3.25: Requirement VAR4.2 - Extracting Metadata during Data Extraction from Varied Sources

Req. ID:	VAR4.2	**Req. Type:**	Functional: Extract Metadata
Parent Req.:	VAR4	**Goals:**	VAR,VER,VAL
Description:	The system shall additionally extract the following metadata in the following formats <p1: specify metadata formats, e.g. microformats> when extracting data from: • Data source • Structure of the data source • Recording method of the data in the data source		

Rationale:	To work with data it is important to know where it comes from, how it was recorded or measured there and in which structure it was store before it got transformed to the data structures used in the system. It allows users to put analysis results into context and to judge the reliability of those results, it allows query developers to determine what semantics and structure their input data has, it gives an overview about which data sources are already leveraged within the system and it provides information for intgegrating data from different sources.
Dependencies:	VAR3.1,VAR3.2,VAR3.3: Extracting metadata from the sources provides necessary information to process schema and entity integration and it gives information about the structure for directly accessing the original source data. VAR4.1: Metadata automatically extracted from data sources needs to be stored within the metadata storage structures.
Conflicts:	VEL1.2.1: Extracting additional metadata during the data extraction process decreases the performance and makes it harder to achieve the acquisition rate.
Literature:	[45, 169]

Table 3.26: Requirement VAR4.3 - Tracking Provenance and Processing History for Data

Req. ID:	VAR4.3	**Req. Type:**	Functional: Track Provenance
Parent Req.:	VAR4	**Goals:**	VAR,VER,VAL
Description:	The system shall collect metadata during the processing of data to track the provenance for data sets within the system, that is their source and which processing steps where conducted to them.		
Rationale:	According to [45] it is important to track the provenance of data during the analysis process to resolve dependencies to other steps in the case of error in one processing step and to allow users to put analysis results into context and to judge the reliability of those results.		
Dependencies:	VER1.3: The computation of a metric to track trustworthiness of data relies on information about the provenance of this data. VAR4.1: Metadata about data provenance automatically extracted during data processing needs to be stored within the metadata storage structures.		
Conflicts:	-		
Literature:	[45, 169]		

Table 3.27: Requirement VAR4.4 - Enabling Manual Inspection and Adjustment of Metadata

Req. ID:	VAR4.4	**Req. Type:**	Functional: Inspect Metadata
Parent Req.:	VAR4	**Goals:**	VAR,VER,VAL
Description:	The system shall allow administrators and users to inspect the available metadata for each data source, data item, data structure etc. and to adjust metadata they consider to be incorrect.		
Rationale:	To provide an overview of available data and data sources it is necessary for users and administrators to inspect the metadata about them. In some cases the extraction of metadata for a source can be faulty or incomplete. In this cases it makes sense for administrators to adjust this metadata.		
Dependencies:	VAR4.4: To inspect the metadata it is necessary to access the metadata storage structures. When metadata gets manipulated during the inspection, the changes need to be written back.		
Conflicts:	-		
Literature:	[45]		

3.2.4 Requirements aimed at handling data veracity

Veracity Requirements Overview

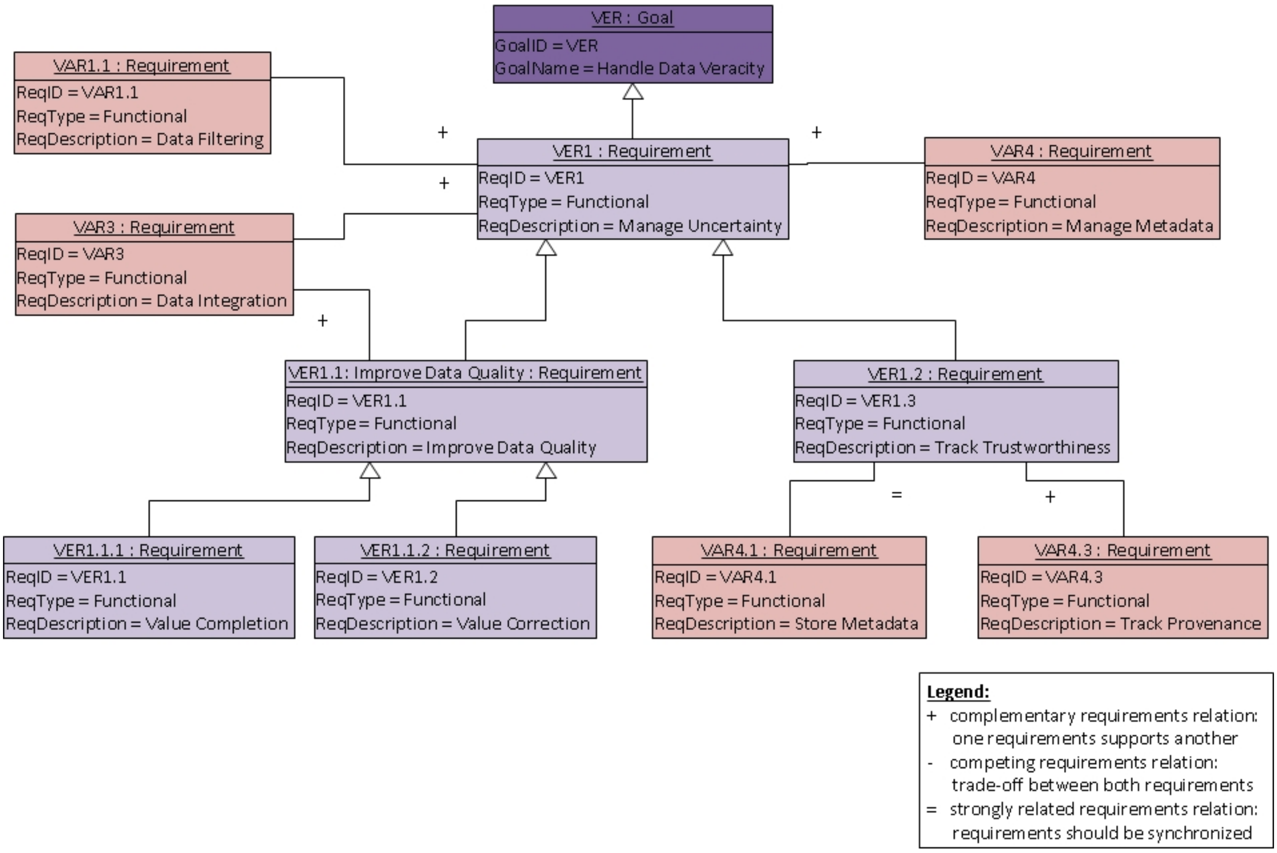

VER : Goal
GoalID = VER
GoalName = Handle Data Veracity
VAR1.1 : Requirement
ReqID = VAR1.1
ReqType = Functional
ReqDescription = Data Filtering
VER1 : Requirement
ReqID = VER1
ReqType = Functional
ReqDescription = Manage Uncertainty
VAR4 : Requirement
ReqID = VAR4
ReqType = Functional
ReqDescription = Manage Metadata
VAR3 : Requirement
ReqID = VAR3
ReqType = Functional
ReqDescription = Data Integration
+
+
+
+
VER1.1: Improve Data Quality : Requirement
ReqID = VER1.1
ReqType = Functional
ReqDescription = Improve Data Quality
VER1.2 : Requirement
ReqID = VER1.3
ReqType = Functional
ReqDescription = Track Trustworthiness
=
+
VER1.1.1 : Requirement
ReqID = VER1.1
ReqType = Functional
ReqDescription = Value Completion
VER1.1.2 : Requirement
ReqID = VER1.2
ReqType = Functional
ReqDescription = Value Correction
VAR4.1 : Requirement
ReqID = VAR4.1
ReqType = Functional
ReqDescription = Store Metadata
VAR4.3 : Requirement
ReqID = VAR4.3
ReqType = Functional
ReqDescription = Track Provenance
Legend:
+ complementary requirements relation: one requirements supports another
- competing requirements relation: trade-off between both requirements
= strongly related requirements relation: requirements should be synchronized

Figure 3.6: Requirements Visualization: Data Veracity View

Veracity requirements aim at managing uncertainty [26]. As described in Chapter 2.1.5, uncertainty is inherit in most 'big data' sources [45, 130, 139, 231]. There are generally two strategies to manage it. One is, to reduce uncertainty by improving the quality of the underlying data[27]. Improving the data quality also has the effect, that it typically supports the data integration. Improving the data quality comprehends two basic tasks, correcting incorrect values[28] and completing missing ones[29]. There are several techniques available to do this, e.g. machine learning techniques to identify incorrect values and derive values for empty fields, simple mapping rules based on other fields or populating values from another data sources during or after the data integration step.

The second strategy is to allow for and expect quality problems in the data. Typically, that is even the case after data cleansing and quality improvements. In that case, it is necessary to give users the necessary information to estimate the trustworthiness of the data. It is even more important to provide with information about data provenance. It can help to define a metric for trustworthiness and calculate it based on the source, the underlying originates from, and on the trust into the different preprocessing steps[30]. After all, some of the preprocessing steps, e.g. information extractions, are based on statistical or machine learning techniques and therefore on probability or estimation [45, 130]. Obviously, tracking trustworthiness of data has a strong dependency to metadata management and especially the tracking of data provenance, as these provide the necessary input for the calculation.

Veracity Requirements Specification

Table 3.28: Requirement VER1 - Managing Uncertain Data

Req. ID:	VER1	**Req. Type:**	Functional: Manage Uncertainty
Parent Req.:	-	**Goals:**	VER, VAR
Description:	The system shall handle uncertain data, that is give meaningful results in the face of uncertain data and put those results into context.		
Rationale:	This requirement directly concerns veracity and data uncertainty as discussed in Chapter 2.1.5. It is a parent requirement to group together sub-requirements that tackle uncertainty. Data of a source itself might be fuzzy and untrustworthy, but in the case of multiple sources it is probable that schemas and semantics differ and they are not always resolvable in an integration step, or that they even contain inconsistent and conflicting data.		
Dependencies:	VAR1.1: Filtering out untrustworthy data can help to decrease the total uncertainty of the data within the system. VAR3: Data integration can resolve some of the issues of unertainty, e.g. bei harmonizing entities and adopting values from other data sources. VAR4: Managing metadata helps to handle uncertainty, as metadata describes the available data, its structure and how it got recorded and as it allows users to set the data into context and judge its trustworthiness.		
Conflicts:	-		
Literature:	[45, 130, 139, 231]		

[26] see requirement VER1
[27] see requirement VER1.1
[28] see requirement VER1.1.2
[29] see requirement VER1.1.1
[30] see requirement VER1.2

Table 3.29: Requirement VER1.1 - Improving Data Quality to Decrease Uncertainty

Req. ID:	VER1.1	**Req. Type:**	Functional: Improve Data Quality
Parent Req.:	VER1.1	**Goals:**	VER, VAR
Description:	The system shall improve data quality by cleaning data from the following sources <p1: specify data sources> and in the following formats <p2: specify data formats>.		
Rationale:	As pointed out in Section 2.1.5, it is often the case that data of a data source is uncertain because of poor data quality. Cleaning imprecise, erroneous or incomplete data before analysing it can improve the data quality, decrease data uncertainty and improve quality of analysis results.		
Dependencies:	VAR3: An improved data quality by completing and correcting values can make it easier to do the data integration step. VER1.1.1,VER1.1.2: Cleaning data consists of value completion and value correction		
Conflicts:	-		
Literature:	[45, 212]		

Table 3.30: Requirement VER1.1.1 - Improving Data Quality - Complete Empty Values

Req. ID:	VER1.1.1	**Req. Type:**	Functional: Value Completion
Parent Req.:	VER1.1	**Goals:**	VER, VAR
Description:	The system shall use the following techniques <p1: specify techniques> to fill empty fields with estimated values under the following conditions <p2: specify conditions>.		
Rationale:	It is often the case, that fields important for data analysis are not propagated. In that case it can be reasonable to fill these fields with some technique (e.g. a machine learning technique, a simple transformation from other fields or adoption of the value from another data source). However, not every field is important and there are situations where an empty field is meaningful. Therefore it should be specified under which conditions (e.g. just specific fields) values should be derived.		
Dependencies:	VAR3.1,VAR3.2: If the schema and entitities among data sources are integrated, empty fields in one data set can be derived from another data set. VER1.1: Value Completion is a child requirement of Improve Data Quality.		
Conflicts:	-		
Literature:	[45]		

Table 3.31: Requirement VER1.1.2 - Improving Data Quality - Correct Wrong Values

Req. ID:	VER1.1.2	**Req. Type:**	Functional: Value Correction
Parent Req.:	VER1	**Goals:**	VER, VAR
Description:	The system shall use the following conditions <p1: specify conditions> to identify untrustworthy or incorrect data and apply the following techniques <p2: techniques> to resolve the untrustworthiness.		
Rationale:	Conditions like error models or conflicting data between different sources can often be used to identify erroneous data values and these errors can be resolved with some technique (e.g a machine learning technique, a simple transformation from other fields or adoption of the value from another data source).		

Dependencies:	VAR3.1,VAR3.2: If the schema and entitities among data sources are integrated, it is possible to identify conflicting information and resolve these conflicts by comparing different data sets. VER1: Value Correction is a child requirement of Manage Uncertainty.
Conflicts:	-
Literature:	[45]

Table 3.32: Requirement VER1.2 - Informing Users about Trustworthiness of Data

Req. ID:	VER1.2	**Req. Type:**	Functional: Track Trustworthiness
Parent Req.:	VER1	**Goals:**	VER, VAR
Description:	The system shall track the trustworthiness of data on the level of <p1: specify level to track trustworthiness> and calculate a trustworthiness metric in the following way <p2: specify method to calculate trustworthiness>.		
Rationale:	As described in Chapter 2.1.5 even after integrating data, completing and correcting it as good as possible, it is still likely that some errors remain. This is even more true in cases, where data correction and integration is based on probability (e.g. when using some machine learning approaches). This uncertainty needs to be visible to the user, so he can evaluate the analysis results in context. Providing a metric of the trustworthiness of the underlying data can help in this process. This metric can be calculated on several levels (e.g. per attribute, entity or data source) and in many ways (e.g. giving a trust value to the data source and manipulating that trust value for all processing steps conducted to the data).		
Dependencies:	VAR1,VAR2.1,VAR3.1,VAR3.2,VER1.1,VER1.2: The sources data is extracted from, the way to extract information from this data, the techniques for data integration, data completion and data correction can all influence the trustworthiness metric for that data. VAR4.1: When computing the metric to track trustworthiness of data it is necessary to access and use the stored metadata. Once the metric is computed, it needs to be stored within the metadata structures.		
Conflicts:	-		
Literature:	[45, 130]		

3.2.5 Requirements aimed at creating value

To not mess up the requirements visualization for value and keep to it somehow simple, it is split up into two parts. The first part describes the requirements that aim at the typical analysis tasks and the final interaction of the user within the system[31]. The second part describes additional services or tasks the system must conduct to maintain or increase the value of the data[32].

Value Requirements Overview - Part 1

Value created from data mainly originates from the analysis tasks conducted over the data[33]. The data analysis can broadly be classified into three categories. The first category comprises classical

[31] see Figure 3.7

[32] see Figure 3.8

[33] see requirement VAL1

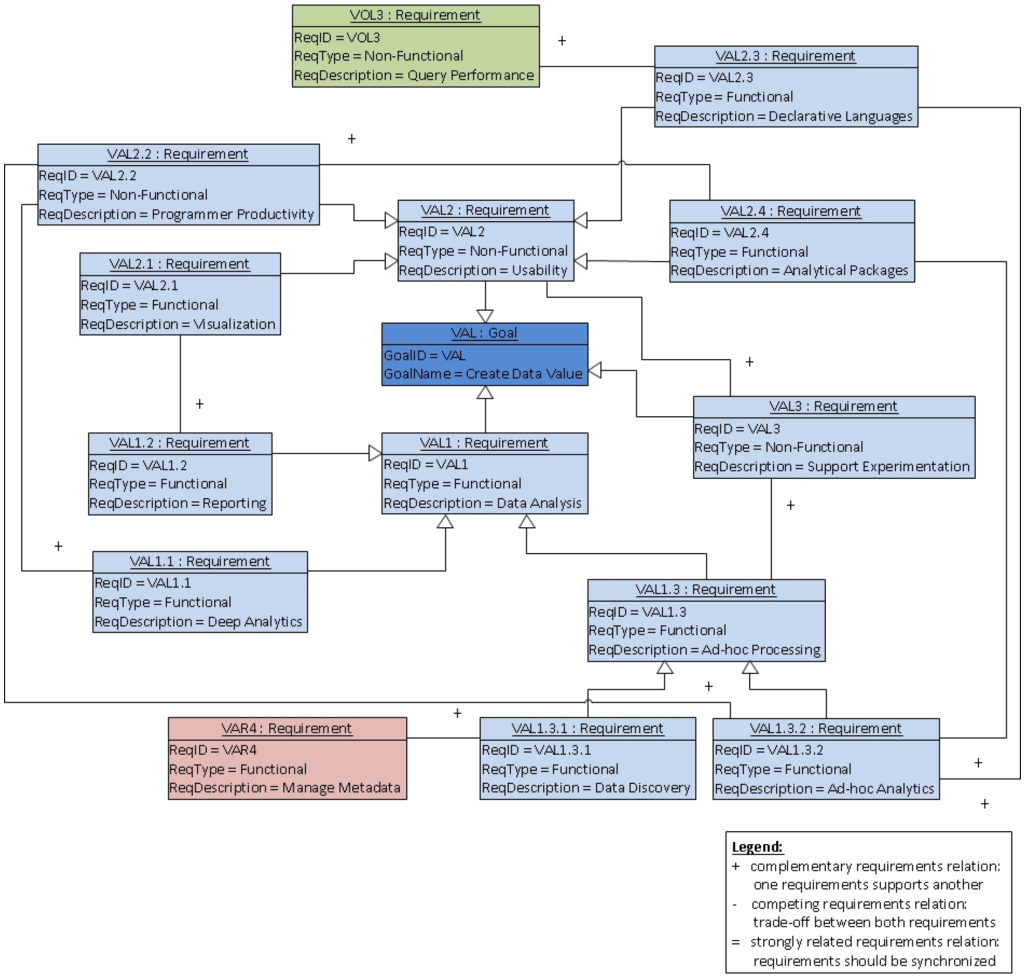


Figure 3.7: Requirements Visualization: Data Value View 1 - User Interaction

business reporting functionality. This is very similar to traditional data warehousing front-ends and refers to predefined and -calculated, often static reports and dashboards as well as classical OLAP navigation functionality[34]. The second category refers to ad-hoc or interactive processing for power users to dig into the available data and conduct deeper analysis [45]. It consists of mere discovery and navigation through the available data and metadata[35] and of ad-hoc analytics or querying based on a declarative language, scripts or on some analytics package[36], e.g. R or SAS. This analysis is conducted ad-hoc and interactive, therefore the response time is an issue. Finally, the third category involves deep analytics tasks, that is complex analysis tasks over large amounts or all of the available data[37]. These tasks typically take a lot of time and are processed during a batch job.

While the data analysis requirements mentioned above mainly describe direct functionality, several usability requirements can be considered, that allow users to interact with the system more effectively[38]. First, the use of visualization techniques make it easier for users to interpret results[39] and are typically prominent in dashboard and reports [45, 132]. Therefore they directly support business reporting, but they can also be applied to ad-hoc analytics results, e.g. visualization functionality embedded in

[34] see requirement VAL1.2
[35] see requirement VAL1.3.1
[36] see requirement VAL1.3.2
[37] see requirement VAL1.1
[38] see requirement VAL2 and sub-requirements
[39] see requirement VAL2.1

analytics packages, and even for deep analytics results. Another requirement is to provide appropriate programming models and framework to enhance the productivity of programming analysis tasks within the system[40] [132]. This directly supports deep analytics tasks, as those are typically developed on demand and need to be changed when analysis requirements change. It can, however, also support ad-hoc analytics and queries, in the case that they are formulated with scripting languages. Another, probably the most wide-spread, possibility is the use of declarative languages[41] to formulate ad-hoc queries. The obvious example is SQL, but there are also new languages, which got developed within the 'big data' ecosystem, e.g. Pig or HiveQL. Declarative languages are in most cases also highly optimized and provide a level of optimization that is only hard and with a lot of effort to achieve in non-declarative languages. Therefore they can have a positive effect onto query performance. Finally, it can be considered to integrate and use analytics package or libraries, which provide often used functionality[42], e.g. machine learning algorithms. An example is Apache Mahout. This supports programmers productivity and these packages can often also be used to quickly formulate some ad-hoc analysis.

Another requirement, which is tightly connected with ad-hoc analytics, is the support of experimentation[43]. Working with data and identifying or optimizing appropriate analysis techniques and parametrization of those techniques require experimentation by the data analysts. Furthermore, data from new sources might need to be experimented with, before a decision is made if they are to be acquired on a regular basis. This requires a sandbox approach, which allows analysts and programmers to 'play' with data in an isolated area. Experimentation gets supported by all usability requirements and by effective ad-hoc analytics possibilities.

Value Requirements Specification - Part 1

Table 3.33: Requirement VAL1 - Analysing the Data

Req. ID:	VAL1	**Req. Type:**	Functional: Data Analysis
Parent Req.:	-	**Goals:**	VAL
Description:	The system shall conduct analysis tasks as specified in sub-requirements VAL1.1, VAL1.2 and VAL1.3 over data from the following sources <p1: specify sources>.		
Rationale:	The whole system is built with the reason to create insights by analysing data. Therefore, this requirement is obvious and the most important one. This is at least true, considering the scope of the reference architecture to exclude operational and transactional processing of 'big data'. The analysis of data can be classified into 3 categories, which are represented by the sub-requirements. Also note, that <p1> needs not necessarily refer to original data sources, but can refer to data sets either persisted or virtual resulting from pre-processing steps, e.g. an data set integrated from several original sources.		
Dependencies:	VAL1.1,VAL1.2,VAL1.3: Data Analysis can be classified into Deep Analytics, Reporting and Ad-hoc Analytics represented by these sub-requirements.		
Conflicts:	-		
Literature:	[45, 62, 63, 83, 169, 195, 213]		

[40] see requirement VAL2.2

[41] see requirement VAL2.3

[42] see requirement 2.4

[43] see requirement VAL3

Table 3.34: Requirement VAL1.1 - Batch-Processing the Data for Deep Analytics Tasks

Req. ID:	VAL1.1	**Req. Type:**	Functional: Deep Analytics
Parent Req.:	VAL1	**Goals:**	VAL
Description:	The system shall conduct the following deep analytics tasks <p1: specify functional requirements that describe the necessary deep analytics tasks> as batch-jobs over the following data sources <p2: specify data sources>		
Rationale:	As described in Section 2.1.6, ‘big data’ is also connected to a shift to more complex analysis methods including data mining, machine learning and simulation over large amounts of data. These tasks are typically to complex and take to much time to be done in an interactive fashion, but must be conducted during regular batch-jobs.		
Dependencies:	VAL2.2: Deep Analytics tasks can typically not be formulated using declarative methods, but need to be programmed in a lower-level language. Methods to enhance programmer productivity, e.g. abstraction from lower-level concepts or from parallelisation, can therefore simplify and accelerate development of deep analytics tasks. Note that the data sources mentioned in <p2> should be a subset of the data sources specified in VAL1.		
Conflicts:	-		
Literature:	[45, 62, 63, 83, 169, 213]		

Table 3.35: Requirement VAL1.2 - Creating Standard Reports from Data

Req. ID:	VAL1.2	**Req. Type:**	Functional: Reporting
Parent Req.:	VAL1	**Goals:**	VAL
Description:	The system shall provide users with pre-calculated standard reports as specified in <p1: specify functional requirements to describe standard reports> with traditional OLAP-like navigation functionality using data from the following sources <p2: specify data sources>.		
Rationale:	Not all end-users and decision makers that use results and insights from ‘big data’ analysis have the necessary skill or the time to formulate ad-hoc queries and directly interpret deep analytics results. Therefore, the system needs to create easy-to-understand standard reports and dashboards for frequently used data and analysis results. Note that the data sources mentioned in <p2> should be a subset of the data sources specified in VAL1.		
Dependencies:	VAL2.1: Standard reports and dashboards are typically targeted at business end-users and often use visualization techniques to present the data.		
Conflicts:	-		
Literature:	[103]		

Table 3.36: Requirement VAL1.3 - Enabling Users to formulate Ad-hoc Processing Tasks

Req. ID:	VAL1.3	**Req. Type:**	Functional: Ad-hoc Processing
Parent Req.:	VAL1	**Goals:**	VAL
Description:	The system shall enable users to interactively work with the available data and to query and analyse it in an ad-hoc manner.		

Rationale:	Power users often work with ‘big data’ in a way of iteratively running queries, analysing the results, drawing conclusions, adjusting and re-running the queries to check assumptions, test hypotheses and to experiment with the data. This needs to be done in an ad-hoc manner and response times as found in batch processing are typically not acceptable.
Dependencies:	VAL3: The ability for users to quickly discover available data and to quickly write and process ad-hoc queries for basic analysis tasks supports the experimentation with data.
Conflicts:	-
Literature:	[45, 63, 111, 195]

Table 3.37: Requirement VAL1.3.1 - Enabling Users to Interactively Discover Data

Req. ID:	VAL1.3.1	**Req. Type:**	Functional: Data Discovery
Parent Req.:	VAL1.3	**Goals:**	VAL,VAR
Description:	The system shall provide users an overview of and enable them to navigate through all available data sources, data available in the system and already computed results for analysis tasks and single processing steps all together with the related metadata.		
Rationale:	To work with data it is obviously necessary for users to be able to discover the available data. They need to be able to navigate through the original raw data, through data along the different pre-processing steps and through the corresponding metadata. This allows them to identify data available to them, view example data and look up the format of this data.		
Dependencies:	VAR4: The data discovery and navigation through the available data and data sources relies on the stored metadata about this data and this data sources. The availability of metadata is a necessary prerequisite.		
Conflicts:	-		
Literature:	[]		

Table 3.38: Requirement VAL1.3.2 - Enabling Users to Perform Ad-hoc Analysis of Data

Req. ID:	VAL1.3.2	**Req. Type:**	Functional: Ad-hoc Analytics
Parent Req.:	VAL1.3	**Goals:**	VAL
Description:	The system shall provide users with an end-point to formulate and process ad-hoc queries and processing tasks over data from the following sources <p1: specify data sources> using the following methods <p2: specify methods to formulate queries and processing tasks, e.g. query languages>.		
Rationale:	As mentioned in the parent requirement VAL1.3, querying and analysing data in an ad-hoc manner is important for power users to interactively work with data. Ad-hoc analysis can either be free or guided. Free methods specified in <p2> can e.g. involve declarative query languages, usage of analytical packages and tools such as SAS, R and Weka Explorer, but also scripting languages, e.g. Python. Guided ad-hoc analysis largely refers to multi-dimensional navigation through a data cube as represented by traditional OLAP tools. Note that the data sources mentioned in <p1> should be a subset of the data sources specified in VAL1.		

Dependencies:	VAL2.3, VAL2.4: Easy-to-use declarative languages and analytical packages enable users to quickly and easily write ad-hoc queries and to conduct ad-hoc analysis tasks. VAL2.2: Programmer productivity measures can support ad-hoc analysis in those cases in which scripting languages are used, e.g. by the inclusion of libraries for that scripting language.		
Conflicts:	-		
Literature:	[45, 63, 102, 111, 195]		

Table 3.39: Requirement VAL2 - Providing Functionalities in an Easy-to-Use Manner

Req. ID:	VAL2	**Req. Type:**	Non-Functional: Usability
Parent Req.:	-	**Goals:**	VAL
Description:	The system shall make it easy for users to explore available data, analysis results as well as results of intermediate steps and should support users in understanding and interpreting those results.		
Rationale:	This is a high-level requirement to comprise and structure the different usability requirements.		
Dependencies:	VAL3: A system that easily allows users to quickly write scripts or prototypes, to formulate queries directly in some declarative language and to use analytical packages and tools for their tasks supports the experimentation with data and analysis techniques.		
Conflicts:	-		
Literature:	[45, 132]		

Table 3.40: Requirement VAL2.1 - Visualizing Data and Analysis Results

Req. ID:	VAL2.1	**Req. Type:**	Functional: Visualization
Parent Req.:	2	**Goals:**	VAL
Description:	The system shall visualize the following data and analysis results <p1: specify data to be visualized> as specified in <p2: specify functional requirements that describe the necessary visualization tasks>.		
Rationale:	Visualization allows users to grasp data and analysis results faster, more intuitively and supports them with interpreting these results.		
Dependencies:	VAL1.2: Standard reports and dashboards are typically targeted at business end-users and often use visualization techniques to present the data. <p2>: Add dependencies to the functional requirements that specify the visualization tasks		
Conflicts:	-		
Literature:	[45, 169, 213][108, pp. 65-68]		

Table 3.41: Requirement VAL2.2 - Supporting Programmer Productivity

Req. ID:	VAL2.2	**Req. Type:**	Non-Functional: Programmer Productivity
Parent Req.:	VAL2	**Goals:**	VAL
Description:	The system shall support programmer productivity by allowing programmers to focus on the application logic while abstracting low-level implementation and infrastructure details away.		

Rationale:	Programming 'big data' systems can involve a lot of low-level implementation and infrastructure details, such as handling parallelization. These tasks, if done manually, typically involve a lot of effort and require experienced programmers with a very specific skill set. This can make the development of functions for such a system rather expensive. Using a framework which takes over responsibility for and abstracts away from infrastructure functionality, e.g. a Map Reduce framework for handling distribution, lets programmers focus on the actual application logic, increases productivity and cuts down development cost. Increasing programming productivity also involves the inclusion and usage of software libraries in the system which implement lower level algorithms, e.g. Mahout or Weka.
Dependencies:	VAL1.1: Deep Analytics tasks can typically not be formulated using declarative methods, but need to be programmed in a lower-level language. Methods to enhance programmer productivity, e.g. abstraction from lower-level concepts or from parallelisation, can therefore simplify and accelerate development of deep analytics tasks. VAL1.3.1: Programmer productivity measures can support ad-hoc analysis in those cases in which scripting languages are used, e.g. by the inclusion of libraries for that scripting language. VAL2.4: Analytical packages often allow to access included functions programmatically via APIs and can be used similar to libraries to increase programmer productivity, e.g. the distinction between the Weka library and the Weka Explorer UI on top.
Conflicts:	-
Literature:	[132]

Table 3.42: Requirement VAL2.3 - Enabling Users to Formulate Queries and Analysis Tasks Using Declarative Query Languages

Req. ID:	VAL2.3	**Req. Type:**	Non-Functional: Declarative Query Languages
Parent Req.:	VAL2	**Goals:**	VAL
Description:	The system shall provide an end-point that allows users to formulate queries and analysis tasks in the following declarative query language(s) <p1: specify declarative query languages> and process those queries over the following data sources <p2: specify data sources>.		
Rationale:	Declarative are a powerful tool for doing analysis in an ad-hoc manner as they abstract away from implementation details. They are typically easer to learn and use for non-technical users compared to a lower level programming language. It is typically also faster to formulate a query in a declarative language than programming it in a lower level language. Note, that not all data sources might be feasible to be queried in a declarative manner, this might e.g. be difficult for unstructured text data. Therefore, the data sources that can be queried in this way are to be specified in <p2>.		
Dependencies:	VAL1.3.2: Easy-to-use declarative languages enable users to quickly and easily write ad-hoc queries for analysis tasks and are the main building block for ad-hoc analysis.		

	VOL3: The abstraction away from implementation details in declarative query languages typically allows query translation and execution in those languages to be highly optimized, both logically and physically. Therefore it frees programmers from doing this optimization in lower level code and prevents the usage of unoptimized code.
Conflicts:	-
Literature:	[45]

Table 3.43: Requirement VAL2.4 - Providing Analytical Packages to Formulate Processing Tasks

Req. ID:	VAL2.4	**Req. Type:**	Functional: Analytical Packages
Parent Req.:	VAL2	**Goals:**	VAL
Description:	The system shall incorporate the following analytical packages <p1: specify analytical packages> to be used by users to analyse data from the following sources <p2: specify data sources>.		
Rationale:	Analytical tools, such as SAS or SPSS, are powerful tools for users to explore data and analyse it using some more advanced methods, e.g. statistical analysis or predictive modeling. Note, that some analytical packages are very specialized and optimized for a certain type of task. Therefore, it can make sense to provide users with several of those tools to use the one best suited for the task at hand.		
Dependencies:	VAL1.3.2: Easy-to-use analytical packages enable users to quickly and easily conduct ad-hoc analysis tasks.		
Conflicts:	-		
Literature:	[83]		

Table 3.44: Requirement VAL3 - Supporting Users in Experimenting with Data and Analysis Methods

Req. ID:	VAL3	**Req. Type:**	Non-Functional: Support experimentation
Parent Req.:	-	**Goals:**	VAL
Description:	The system shall enable and support users to experiment with data.		
Rationale:	As already mentioned in requirement VAL1.3, power users work interactively with data, making hypotheses, testing these by experimenting with the data, drawing some conclusions and refine their hypotheses. They also need to experiment with new analysis methods, data mining and machine learning techniques, parametrisation of those etc. This kind of experimentation needs to be supported by the system, e.g. by making it easy to extract samples out of data and to work with it in a sandbox.		
Dependencies:	VAL2: A system that easily allows users to quickly write scripts or prototypes, to formulate queries directly in some declarative language and to use analytical packages and tools for their tasks supports the experimentation with data and analysis techniques.		
Conflicts:	-		
Literature:	[95, 186, 191]		

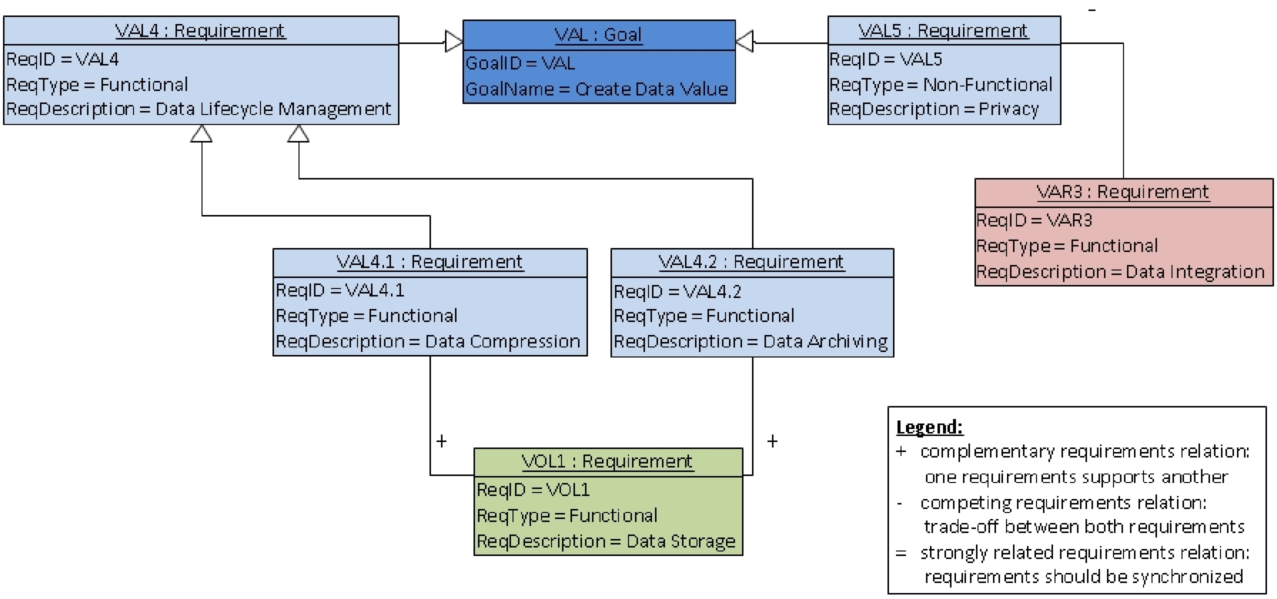

VAL4 : Requirement
ReqID = VAL4
ReqType = Functional
ReqDescription = Data Lifecycle Management
VAL : Goal
GoalD = VAL
GoalName = Create Data Value
VAL5 : Requirement
ReqID = VAL5
ReqType = Non-Functional
ReqDescription = Privacy
-
VAR3 : Requirement
ReqID = VAR3
ReqType = Functional
ReqDescription = Data Integration
VAL4.1 : Requirement
ReqID = VAL4.1
ReqType = Functional
ReqDescription = Data Compression
VAL4.2 : Requirement
ReqID = VAL4.2
ReqType = Functional
ReqDescription = Data Archiving
+
+
VOL1 : Requirement
ReqID = VOL1
ReqType = Functional
ReqDescription = Data Storage
Legend:
+ complementary requirements relation: one requirements supports another
- competing requirements relation: trade-off between both requirements
= strongly related requirements relation: requirements should be synchronized

Figure 3.8: Requirements Visualization: Data Value View 2 - Additional Services

Value Requirements Overview - Part 2

Finally, there are some requirements that need to be satisfied to maintain and govern the value creation of data. The first of those is the application of data lifecycle management[44] [202]. Data lifecycle management helps to keep the data volume within the system in line by archiving outdated data that is not needed on a regular basis anymore[45]. Furthermore, it reduces the amount of storage needed by compressing the stored data[46]. Compressing data can also create performance improvements in some cases. Another important issue is to ensure data privacy[47]. The system needs to conform to governmental regulations and policies and need to be in line with the internal privacy standards. This is even more important in the case of 'big data', where the large scale integration of data from different sources can lead to privacy breaches, which is anonymized in each single source but where personal information can be identified and re-personalized when those data sources are integrated and put together [160].

Value Requirements Specification - Part 2

Table 3.45: Requirement VAL4 - Managing Data along its Lifecycle

Req. ID:	VAL4	**Req. Type:**	Functional: Data Lifecycle Management
Parent Req.:	-	**Goals:**	VAL
Description:	The system shall manage the lifecycle of the stored data, that is track data records and documents from their creation and, determine based on the following rules <p1: specify functional requirements that describe data lifecycle management rules> if data is still to retain or became stale and to archive or delete data.		

[44] see requirement VAL4
[45] see requirement VAL4.2
[46] see requirement VAL4.1
[47] see requirement VAL5

Rationale:	Data is typically stored because it is considered valuable. Not every data item has the same value, though. With a restricted IT budget and growing data volumes, it makes sense to prioritize data, focussing effort, e.g. for data cleaning and integration, onto higher priority data and to save up storage by discarding stale data from the system, either through archiving or deleting it. These actions, especially the deletion of stale data, however need to comply to governmental laws and regulations, e.g. the Sarbanes-Oxley Act which regulates which records need to be stored how long. Having a system that can conduct data lifecycle management tasks automatically and rule-based guarantees the compliance to such regulations while reducing effort needed for the IT department.
Dependencies:	<p1>: Add dependencies to the functional requirements that specify the data lifecycle rules
Conflicts:	-
Literature:	[202, pp. 133-140]

Table 3.46: Requirement VAL4.1 - Compressing Data to Save Storage Space

Req. ID:	VAL4.1	**Req. Type:**	Functional: Data Compression
Parent Req.:	VAL4	**Goals:**	VAL,VOL
Description:	The system shall compress data where possible using the following compression technique <p1: specify compression technique>.		
Rationale:	With growing data volumes it makes sense to compress data where possible, to decrease the required storage space. There are also cases where compression leads to better performance.		
Dependencies:	VOL1: Compressing data decreases the data volume and therefore the amount of storage needed.		
Conflicts:	-		
Literature:	[202, pp. 137-138]		

Table 3.47: Requirement VAL4.2 - Archiving Data when it is no longer Needed

Req. ID:	VAL4.2	**Req. Type:**	Functional: Data Archiving
Parent Req.:	VAL4	**Goals:**	VAL,VOL
Description:	The system shall archive or delete data based on the rules specified in requirement VAL4 using the following archiving method <p1: specify archiving method and archiving system used>.		
Rationale:	With growing data volumes it makes sense to archive stale data where possible, to decrease the required storage space.		
Dependencies:	VOL1: Archiving data decreases the data volume and therefore the amount of storage needed.		
Conflicts:	-		
Literature:	[202, pp. 137-138]		

Table 3.48: Requirement VAL5 - Ensuring Privacy and Security of Data

Req. ID:	VAL5	**Req. Type:**	Non-Functional: Privacy
Parent Req.:	-	**Goals:**	VAL
Description:	The system shall ensure privacy of data, that is data should not be able to be accessed by people not authorized for it. The system shall therefore conduct the following measures <p1: specify functional sub-requirements that describe measures to ensure privacy>.		
Rationale:	People whose information is stored and processed in an information system, e.g. users, customers or employees, have particular expectation about how their data is stored and used and how its privacy is protected. Furthermore, there are governmental regulations that define how data can be used and what privacy measures need to be taken. Privacy breaches can have a huge impact on a company's business. As described above, this is even more important in the case of integrating many different sources, because cross-references between those sources can be used to de-anonymize data which is anonymized in the single sources. Privacy measures specified in <p1> can include authentication and authorization, tracking data access by users and anonymization techniques.		
Dependencies:	<p1>: Add dependencies to the functional sub-requirements that specify privacy measures.		
Conflicts:	VAR3: The integration of data across sources can make it possible to identify and assign personal information which is not identifiable in each single source.		
Literature:	[81, 160]		

4 Reference Architecture

In this chapter I will develop and describe the 'big data' reference architecture. This will be based on the groundwork from the previous chapters, namely the definition of 'big data' in Section 2.1, the definition of reference architectures as well as the characterization of the reference architecture on hand in Section 2.2 and finally the requirements specified in Chapter 3. The reference architecture is intended to give an overview of the 'big data' space, involved functionality and technology and to guide and reason about design decision for a respective concrete architecture.

First, in Section 4.1, I will describe and reason about the methodology I use to develop the reference architecture. Afterwards, I will step-wise develop and refine the functional reference architecture in Section 4.2 and finally, I will map existing technology to the functional components of the reference architecture in Section 4.3.

4.1 Architectural Methodology

As discussed in Section 2.2.2 under 'Step 4: Construction of the reference architecture', developing a reference architecture means building a set of models structured according to defined views. To guide on the decision how to separate the concern of system architecture design into different views, there are several models or frameworks available. One of the earliest was Kruchten's 4+1 view model for architecture [150]. Rozanski and Woods [193, pp. 211-360] developed a viewpoint catalogue based on Kruchten's 4+1 model. Here, a viewpoint describes guidelines and principles for a particular view and identifies the respective stakeholders and their concerns. As this reference architecture aims at describing the general functionality needs within the 'big data' space and how this functionality can be mapped onto available technologies and software modules, I will pick the functional and the development viewpoint out of this catalogue. The other viewpoints are less applicable due to the abstraction of the reference architecture and its placement on the system level. The information viewpoint is not applicable as the reference architecture does not describe concrete data models of a respective domain. The concurrency viewpoint is not directly applicable at the system level of the reference architecture, but is addressed on a lower level within its subsystems and modules (e.g. within a massively parallel processing framework). The deployment view is not applicable, as the reference architecture focusses on software, not on its deployment and hardware.

As a guideline and to structure the design process itself, I will use the process of stepwise refinement as described and applied by Grefen et al. [125]. According to this approach and in line with Galster and Avgeriou [114] the design of an architecture is based on a set of requirements. Building on the requirements identified in Section 3.2, I will build a high-level model of functionality in Section 4.2.1 and then step-wise refine that functional model. In a first step I will break down the clusters of functionality to a more detailed level. The result can be seen as the functional view of the architecture. Afterwards I will map the functionality to general software components and add development related information, e.g. general data stores. This represents a general development view of the reference architecture. This last refinement step is described in Section 4.2.2. In a final step, I will add concreteness by mapping these general software components to concrete technology and software platforms, e.g. to more concrete database technology. This can also mean, that I will identify and mark gaps, where no or no mature software modules are available, yet. See Section 4.3 for this last step.

4.2 Reference Architecture Design

In this Section, I will develop the proposed reference architecture. As described in Section 4.1, this will be a step-wise process. Section 4.2.1 will describe the functional, more high-level view of the reference architecture, while I will concretize in Section 4.2.2 to a more implementation oriented view of the reference architecture.

4.2.1 Reference Architecture Functional View

The high-level reference architecture shown in Figure 4.1 aims at showing the main functionality needed in the system. This can be seen as a process of steps or functions that get conducted on the data, from preparing the data, analysing it and presenting the results. In that, the ''big data' process is not un-similar to the traditional data warehousing, where data gets sequentially processed in different layers or phases of the extract-transform-load process before it gets analysed, e.g. within an OLAP engine, and presented to the end-user[1].

To acknowledge this process characteristic und to provide further structure, I am using and incorporating the 'Big Data Analysis Pipeline' as proposed by Agrawal et al. [45]. Agrawal et al. split the analysis of 'big data' into distinct phases of a sequential processing pipeline: Acquisition / Recording, Extraction / Cleaning / Annotation, Integration / Aggregation / Representation, Analysis / Modeling and Interpretation. I organize the functionality the system needs to provide along the different steps of this pipeline. In other words, the different functions are assigned to different phases of the analysis pipeline and therefore to different phases of a 'big data' analysis process. The system functions are depicted as components of an UML component diagram, while the pipeline is shown as packages. Keep in mind however, that these are general functions and not identical to software components. They can be mapped, but one function might be distributed across several software components or a software component might implement several functions. For the high level diagram, functional components and the pipeline steps proposed by Agrawal et al. [45] are quite similar and have a large overlap. I also omitted interfaces to external systems, e.g. data sources at this point. This will change and get more complicated, the further down the architecture process and the more detailed the functional components will be broken down.

The functional components are very closely tight to functional requirements as specified in Chapter 3.2 and focus on depicting the higher level requirements. In that sense it is very coarse-grained, but it gives a first overview, serves as a starting point of the architecture process and guides the further

[1] see Section 2.3.1 for an overview of traditional data warehouse architecture

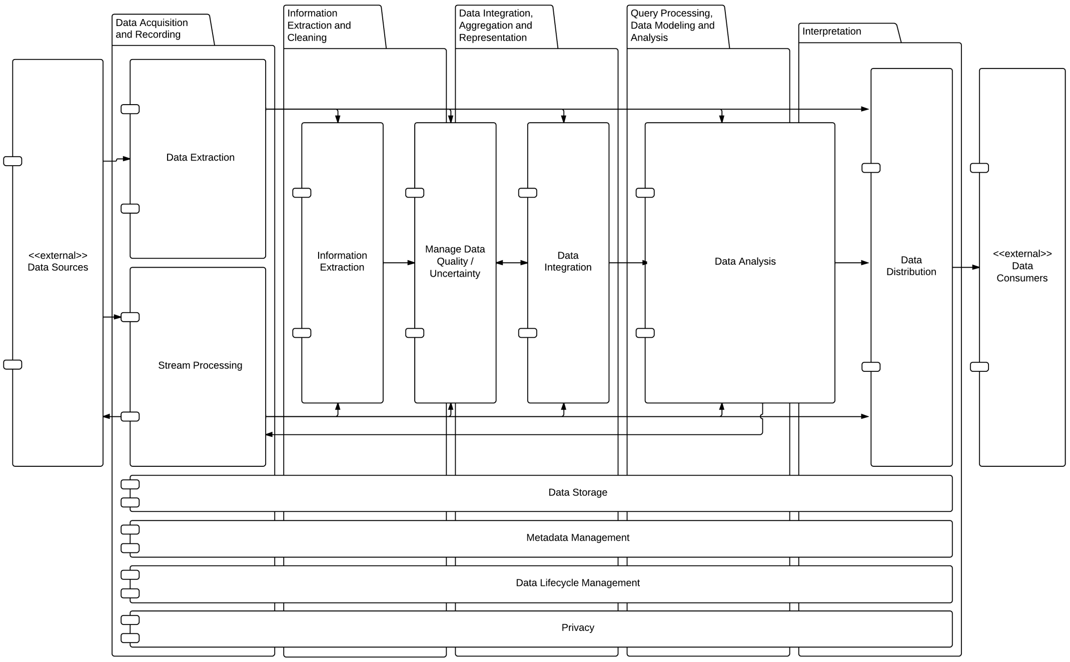

Data Acquisition and Recording
Information Extraction and Cleaning
Data Integration, Aggregation and Representation
Query Processing, Data Modeling and Analysis
Interpretation
<<external>> Data Sources
Data Extraction
Stream Processing
Information Extraction
Manage Data Quality / Uncertainty
Data Integration
Data Analysis
Data Distribution
<<external>> Data Consumers
Data Storage
Metadata Management
Data Lifecycle Management
Privacy

Figure 4.1: Reference Architecture: High-level functional view

design. The mapping of the requirements onto functional components is quite straightforward and can be seen in Table 4.1.

Looking at the reference architecture, it gets apparent, that there are functional components which are placed and create value along the analysis pipeline. These can be seen as steps within a data analysis process which in the end lead to a result presented to the user.

However, these steps are not necessarily mandatory. This is depicted by the structure of the connectors (arrows). It can e.g. make sense to go from the data extraction directly to data cleaning or even data integration if the source data is in a structured or well-understood semi-structured format. Another example are applications of data discovery where end-users can directly navigate through the source data. In the case of heavy timeliness needs it can also be necessary to leave out data cleansing and to accept more approximate results in exchange for latency. This can be especially important for stream processing.

Furthermore, some of these steps are not necessarily sequential, but can be iterative. This is the case for data cleaning and data integration, which can have a rather complex interrelation. Doing the cleaning step first, as Agrawal et al. [45] propose, means cleaning data from each source in isolation. Identifying incorrect data or filling in empty attributes for a data set can make it easier to integrate it with another data set. On the other hand, integrating two or more data sets can lead to quality issues in the resulting set, e.g. new duplicates or inconsistencies between both sources, which need again to be cleaned. Rahm and Do [182] e.g. point out, that data cleansing needs increase significantly when several data sources are merged. Furthermore, data integration can also data cleansing activities which were not possible before. As an example it is possible to conduct data validation across data sets to detect inconsistencies, incorrect data and to fill empty attributes from another data set.

Functional Component	Requirements
Data Extraction	VAR1 (Data Extraction) and sub-requirements
Stream Processing	VEL1 (Handle Data Streams) and sub-requirements
Information Extraction	VAR2.1 (Information Extraction) and sub-requirements
Manage Data Quality / Uncertainty	VER1 (Manage Uncertainty) and sub-requirements
Data Integration	VAR3 (Data Integration) and sub-requirements
Data Analysis	VAL1 (Data Analysis) and sub-requirements
Data Distribution	VAL2 (Usability) and sub-requirements
Data Storage	VOL1 (Data Storage)
Metadata Management	VAR4 (Manage Metadata) and sub-requirements
Data Lifecycle Management	VAL4 (Data Lifecycle Management) and sub-requirements
Privacy	VAL5 (Privacy) and sub-requirements

Table 4.1: Reference Architecture: High-level, functional view

Data Extraction

Data that is to be analysed naturally needs to get into the system. This is the task of data extraction (derived from requirement VAR1) and to some extent stream processing (see next component). The difference between data extraction and stream processing is broadly speaking, that data extraction aims at acquiring data at rest, while stream processing aims at acquiring data in motion. The data extraction component is therefore more batch related and includes connectors to the source systems acquire the data, but also management components, e.g. to monitor changes in the data source and handle delta loads. The extraction can e.g. be realized using standard or proprietary database interfaces (e.g. ODBC), API's provided by the source application, but also file-based interfaces. Data extraction can also refer to crawling data from web pages. Referring to requirement VAR2 (Multi-structured data) data can be extracted from **structured, semi-structured and unstructured data sources** [45, 46, 141]. The distinction between different types of data sources is derived from and just depicts the variety of in-flowing data as described in Section 2.1.4. The type of data can have some influence on the processing pipeline. Unstructured data e.g. necessitates an information extraction step, while structured data does not. Data extraction can also involve **data filtering** [45]. This sub-function is directly derived from requirement VAR1.1 (Data Filtering). Some times data can or needs to be filtered if it is not necessary for further analysis or if it is not suitable for further analysis due to privacy concerns. The filtering can be conducted rule- or attribute-based or based on some machine-learning model, e.g. to classify relevant data items.

Stream Processing

The stream processing component is directly derived from requirement VEL1 and its sub-requirements. As mentioned above it aims at acquiring and analysing data from streams near real-time.

Stream Processing - Data Stream Acquisition: The data stream-acquisition sub-function is directly derived from requirement VEL1.2 (Process Streaming Data) and its sub-requirement. It refers to the acquisition rate challenge as described in Section 2.1.3 and aims at acquiring data from in-flowing data streams and store relevant data (after filtering) [213]. The acquisition includes capabilities to 'subscribe' to a data stream or to simulate the stream via quickly repeating API calls. Storing means to put the streaming data into some temporary data store from where it gets incorporated into the regular processing pipeline, so it can be analysed together with data at rest. It can additionally be loaded into temporary views, so it can be analysed by end-users more timely, if the time waiting for the data running through processing pipeline is prohibitive, mainly because the time interval between the regular transformation and loading process is too long. In that case the data can be deleted from the temporary views, once it is loaded into the regular data stores. Again, data stream acquisition can also include **data filtering**, similar to the data extraction component.

Stream Processing - Data Stream Analysis: The functional sub-component for Data Stream Analysis is directly requirement derived from VEL1.1 (Analyse Streaming Data) and its sub-requirements. It refers to the timeliness challenge and the task is to automatically analyse streaming data while it is flowing and accordingly react to the analysis results as described in Section 2.1.3. This is different compared to analysis in a temporary view as described under Data Stream Acquisition. Loading streaming data into temporary view can only supplement manual analysis tasks (e.g. reporting or ad-hoc analytics), but due to its manual nature cannot ensure a timely reaction to the in-flowing data. This needs to be done via an automatic analysis, which sends a response back to the source application, so the source application can adjust the process or an alert into a report or dashboard.

In its nature stream analysis is however fundamentally different compared to analysing data at-rest. The data is always incomplete in the sense that there is not a fixed data set as input, but the data set is growing during the analysis and ‘unknown before execution’ [214]. Stream analysis is ongoing incremental and requires ‘one-go’ analysis. Techniques like sketching and approximation are therefore fundamental [126], while they play no or only a minor role in analysis at-rest. An extensive description of these techniques and special stream mining algorithms is out of scope of this thesis as the focus is on the general architecture for ‘big data’ and an overview of infrastructure software to support it. For an introduction and deeper discussion of these concepts I refer to according literature [41, 42, 43, 55, 69, 79, 214].

The Data Stream Analysis can however use partial results, models and rules pre-calculated during a deep analytics task at-rest to accelerate the analysis [45][231, pp. 9-14], therefore the backflow from the data analysis component in the diagram in Figure 4.1. Data Stream Analysis also includes the triggering of actions based on analysis results, the communication of results back to the transactional application or alerts to users.

Information Extraction

Information extraction is derived from requirement VAR2.1 and its sub-requirements. It aims at all functionality to extract information from semi-structured and unstructured content and thereby impose structure on it, that is storing the extracted information in structured form [45, 46, 57]. This can include simple parsing and **structure extraction** from semi-structured sources e.g. XML or HTML pages [200]. It also refers to using more complex techniques like text analytics and natural language processing for **classification**, **entity recognition** and **relation extraction** from unstructured data [57]. Using ontologies, e.g. expressed in OWL, can also be helpful, e.g. for identifying and classifying entities, while the other way round information extraction can be used to populate ontologies.

Information Extraction - Classification: Classification refers to all functions that aim at classifying unstructured data in different dimensions, e.g. extracting a topic or sentiment analysis.

Information Extraction - Entity Recognition: Entity recognition is used to identify unique entities in unstructured data. This also includes entity matching to merge identical entities and therefore needs to consider synonyms, that is different names referring to the same entity e.g. due to term evolution, and homonyms, that is different entities sharing the same name. An entity recognition component furthermore needs to categorize the identified e.g. as persons or products. This is easier, if domain knowledge can be used and the categorization is based on a set of known and named entity types. It is harder, if general entities need to be extraction, whose possible types are not known in advance [57].

Information Extraction - Relation Extraction: Building on entity recognition it is also possible to extract facts or relations about entities, e.g. in the form of triples comparable to RDF. These facts can either be attributes and attributes values, e.g. the age of a person, or general relationships between entities, e.g. a person 'is born' in a city, where both the person and the city are extracted entities. Relation extraction is often limited onto a context, where possible entity types are named and known in advance, as the extraction of attributes necessitates knowledge about which attributes can be present. For the task of extracting general relationships it is furthermore especially useful to use ontologies that specify possible relationships between entity types. Furthermore, there is an interdependence between extracting attributes for an entity and the entity matching task during entity recognition, as identical or similar attribute values are a strong indication for entities to be identical as well [57].

Information Extraction - Structure Extraction: This functions aims to extract information out of semi-structured data by using the structure that is given. This can include parsers to parse and extract metadata from binary objects, splitting an object of comma separated values into single attributes are parsing and extracting relevant information from documents specified using some markup language, e.g. translating XML or JSON files into the relational schema or extracting relevant information from the HTML code of crawled web pages [200].

Manage Data Quality / Uncertainty:

This component is derived from VER1 and its sub-requirements. It partially refers to data cleaning, that is correcting errors in the data, completing empty attributes and optionally identifying and eliminating noise and outliers [182]. It therefore refers to handling data quality problems that originate from a single data set, contrary to data integration, which refers to harmonizing different data sources and handling quality issues and inconsistencies that occur when combining data from different sources. Data cleaning techniques can generally be classified as data scrubbing, that is using domain knowledge and rules to identify and correct errors (e.g. defining a threshold for an attribute value), and data auditing, that is to use statistical, data mining and machine learning techniques to do the same. A simple example for the later is the use of the average of an attribute value and to identify tuples that exceed a tolerance interval of that average as outliers. One can also distinguish between several sub-functions: **value completion**, **outlier detection & smoothing**, **duplicate filtering** and **inconsistency correction** [61, pp. 101-105]. Note, however, that data can only be cleaned to a certain extent and the resulting data quality is still largely dependent on the quality of the source data.

Another part of managing data quality refers to giving users an assumption of the credibility of the data and estimate the probability of the results. This is related to metadata management (see below), mainly provenance tracking, presenting users with the necessary information and calculating a trustworthiness score. Though, this cannot happen as a separate function within the data processing pipeline. At this point in time, the processing is just not complete and information from processing steps further down the pipeline also need to be incorporated into the score. Therefore, this needs to happen integrated into other function, namely metadata management (provenance tracking) and data analysis as well as the user interface (calculating and presenting the trustworthiness score).

Manage Data Quality - Value Completion: Value Completion is directly derived from requirement VER1.1.1 (Value Completion). It includes all functionality to fill incomplete and empty attribute values. This can be based on simple constant, derivations of values from other attributes or statistical and machine learning techniques. A simple example for the later would again be to use the average of all attribute values

Manage Data Quality - Outlier Detection & Smoothing: Outlier detection is derived from requirement VER1.1.2 (Value Correction). The goal is to identify outliers and errors and to correct or smooth them. The identification can be based on rules, thresholds or on statistical or machine learning models. As a reaction a tuple which contains incorrect attribute values can either be simply flagged, discarded, passed to a separate error handling area where it can be manually inspected and corrected or it can be automatically corrected or smoothed. Correcting an attribute value is similar to value completion and can share functionality. Generally an empty attribute value can be seen as a special version of an outlier.

Manage Data Quality - Duplicate Filtering: The function of this component is obvious - filtering out multiple data items that actually refer to one identical object or entity. This is easy if one tuple is just duplicated using the same key or at least identical values of the other attributes. It gets more complicated if attribute values differ between duplicated and involves record matching techniques in these occasions [110]. Additionally this also means, that two possible attribute values need to be reduced to one valid. Using an error handling area for a manual decision can again be a good option if the issues does not occur to often and is feasible to be corrected manually. Note also, that there is some overlap between Duplicate Filtering and Entity Matching during Data Integration, as both involve identifying and resolving identical entities.

Manage Data Quality - Inconsistency Correction: There are often constraints on single attributes or on a combination of attributes, e.g. a ‘late delivery’ flag needs to be set if the interval of ‘order data’ and ‘delivery data’ is too large. There is also the issue of referential integrity between relations. If these constraints are violated, e.g. because they are not checked and imposed in the data source, they need to be identified and corrected if possible. Note again, that there is some overlap to error detection. Note further, that especially cases of referential integrity, e.g. if a tuple of a relation references a tuple of another relation which does not exist, are often impossible to correct automatically. In those cases the only options are either to discard the reference or the whole tuple, to create a dummy tuple in the referenced relation or to have the correction been done manually.

Data Integration

Data Integration is derived from requirement VAR3 and its sub-requirements. It aims at all measures to persistently or virtually harmonize data from different sources and transform them into one overarching schema, so they can be queried together. This mainly includes **schema integration** and **entity resolution** [106, 128, 212].

Data Integration - Schema Integration: Schema Integration also called schema mapping is derived from requirement VAR3.1 (Schema Integration). Integration at this levels refers to all harmonization that needs to be done because the data in heterogeneous sources are modelled differently. The goal is to convey the original schemas of two or more data sources to one global schema and to transform the data accordingly so it can be queried together based on that global schema [106]. This can include field or attribute mappings from the source schemas to the global schema [128], the harmonization of data formats (e.g. converting from type 'number' to type 'data'), representation and coding harmonization (e.g. converting flags 'male'/'female', '0'/'1' or 'm'/'f' to one global representation), string harmonization (e.g. converting 'Markus Maier' to 'Maier, Markus') as well as harmonization and transformation to one valid unit of measure [61, pp. 95-101]. In its simplest form, data integration can be implemented via a set of fixed mapping and transformation rules, but there are also efforts to automatically generate probabilistic schema mappings, e.g. based on machine learning techniques or fuzzy string matching [128, 212]. These later approaches can also be used to bootstrap the integration rules for later refinement [90]. Another approach can be to use semantic web technology and ontologies to describe the different schemas and then use inference and ontology mappings for the integration [48, 169].

Data Integration - Entity Resolution: Entity Resolution is derived from VAR3.2 (Entity Integration) and aims at integration on the entity level, that is identifying data that refers to the same entity and integrate it (e.g. matching 'Markus Maier' and 'M. Maier'). This can be done simply using key harmonization. That is, using mapping tables to map keys used in different data sources to surrogate keys in a global, common key space. However, creating such mapping tables can be tedious especially considering large volumes of data and it is often also infeasible to identify identical entities just based on keys [61, pp. 96-97]. Therefore, statistical and probabilistic record linkage approaches are used, also called entity matching or entity resolution [83, 106, 110, 174, 212, 225]. Note again, that there is some overlap with Duplicate Filtering during the data cleaning phase, but during the data integration phase the emphasis is not in just filtering out records referring to the same entity, but to merge them, as they typically hold different information and attributes based on the data source the are coming from. This merge can be easy, if the sets of attributes in both sources do not overlap, while it gets more involved, if both sources share attributes with contradicting values. Merging in the later case is treated in the field and research area of 'data fusion' [68, 106].

Data Analysis

The data analysis component is derived from VAL1 and its sub-requirements. It contains all functionality that directly aims at deriving meaning and insights from data. Generally, one can distinguish several categories of data analysis functionality. This functionality can be either directly end-user facing or it prepares information, calculate intermediate results and creates insight, which are then stored together with and enrich their input data and are input themselves for further, directly end-user focussed processing and presentation. This is the case for **value deductions** and **deep analytics** tasks. The reasons to timely separate these functions from the presentation can be performance reasons which necessitate pre-computation, especially in cases which typically involve computation in batch mode, or they can have organizational reasons, e.g. to centralize the definition of critical, enterprise-wide key figures in one place, which can then be used by different analysis tools. On the other hand, in this cases key figure definition is less flexible as a trade-off. User-facing analysis can be categorized by the level of interactivity and freedom for the user. On the one end, there are rather fixed and pre-defined **reporting & and dashboarding** solutions, on the other hand, there is **free ad-hoc analysis**, which allows users to freely query data and define their own computations.

Guided ad-hoc analysis (e.g. traditional OLAP approaches) and **data discovery & search** lie somewhere in between the two as both allow different levels of free navigation while typically limiting definition of own computations.

Note, that these distinctions can be kind of fuzzy with dashboards gaining capabilities for navigation and filtering, moving them closer to guided ad-hoc analysis. The distinction between free and guided ad-hoc analysis is fuzzy in itself depending where to draw the line considering freedom and query flexibility. Furthermore, free ad-hoc analysis also allows for statistical and predictive analysis tools sharing similar techniques with deep analytics. There the differences are very much dependent on the available hardware performance, scalability and amount of data involved to dictate the timeliness of the analysis either requiring batch computation or allowing interactive analysis. One characteristic is, that users during ad-hoc analysis often need to rely on sampling available data, while deep analytics can incorporate all available data with the possible size of the sample and input data depending on the complexity of the computation (as well as on the availability of hardware and performance and scalability of the underlying system).

Data Analysis - Value Deductions: Deducting values is the most simple of all analysis tasks. It means adding additional information with a single record or a small amount of records (e.g. if a business transaction is represented by one header and several item records) as input. This can refer to calculating key figures (e.g. deriving the profit from revenue and costs) or enrich master data information (e.g. adding a contact person based on a given company). These value deductions are typically when the data gets transformed for and loaded into an analysis data store and can then be addressed by the front-end analysis tools. Most of the time, the pre-computation is not done for performance reasons as it is not very complex and could normally be done in real-time without adding performance issues. The reason is typically to compute critical, enterprise-wide key figures in one place, so they adhere to the same definition every time they are used in some front-end tool.

Data Analysis - Deep Analytics: Deep Analytics is directly derived from requirement VAL1.1 (Deep Analytics) and refers to all data analysis tasks that are too complex or involve too much input data to be conducted interactively and are therefore batch computed. Often this includes more complex techniques like machine learning, data mining and statistical analysis [62, 83], but batch computation can also be necessary for simple computations (e.g. computating ratios) where the input data is very voluminous. Where interactive analysis often uses some sort of sampling and only takes in parts of the available data, deep analytics task can involve extensive parts or even all of it. Deep Analytics tasks typically compute and store insights, rules, models or extract relationships between data items, which are later utilized and navigated through using end-user facing analysis functions, e.g. reporting and ad-hoc analysis. Results can also be incorporated into some other tasks of the processing pipeline allowing for a feedback loop, e.g. by incorporating mined classification models into valued correction during data cleaning.

Data Analysis - Reporting & Dashboarding: Reporting & Dashboarding is derived from requirement VAL1.2 (Reporting). It describes the definition of fixed reports and dashboards, which largely limit navigation and flexibility of the end-users. Dashboarding additionally corporates visualization techniques, e.g. traffic lights and diagrams, to allow for easier and faster grasping of the presented data and key figures. They are often used for executive personnel or general employees who have few time to spend and often not the lower-level skills for sifting through data, but need a fast and comprehensive overview of available information, e.g. the company's performance in different areas. Reporting & Dashboarding often relies on pre-calculated values or incorporates deep analytics results, but can also compute simple key figures on the fly, especially if those key figures are only needed for a single report.

Data Analysis - Guided Ad-hoc Analysis: Guided Ad-hoc Analysis is derived from VAL1.3.2 (Ad-hoc Analytics). It refers to tool-based ad-hoc analytics with fixed navigational options (e.g. traditional OLAP navigation). Fixed navigational options means, that users can typically navigate and filter through dimensions and master data attributes with the values of according key figures being adjusted. This allows to investigate results by drilling down into the data and adding detail (e.g. switching from showing the revenue per country to showing the revenue per single location or per country and product) or switching the perspective (e.g. switching from showing revenue per country and product to showing revenue per country and customer). The users are, however, often limited to a pre-defined set of key figures and cannot define or change their calculation. It is also not possible to do deeper analysis of the data, e.g. by using statistical methods.Furthermore, also the input data sources to navigate through are fixed.

Data Analysis - Free Ad-hoc Analysis: Free Ad-hoc Analysis is also derived from VAL1.3.2 (Ad-hoc Analytics) and distinguishes itself from guided ad-hoc analysis by the level of flexibility it provides to its users. While they are granted a higher level of freedom, this also means, that respective tools require a higher level of skill and knowledge about underlying analysis concepts and a larger time effort to sift through the data and use these context. In exchange for this larger skill and time requirement, users can freely query available data sources usually based on directly formulating queries in some declarative language (e.g. SQL, PigLatin or HiveQL). Tools for advanced or predictive analytics using statistical and data mining techniques (e.g. SAS or SPSS) also fall into this category. The user communicates with these tools interactively, formulating a query and getting rapid responses, to sift through the data and allow analytical reasoning over data at usage time [45, 111, 195].

Data Analysis - Data Discovery & Search: Data Discovery & Search is derived from requirement VAL1.3.1 (Data Discovery). It aims less at analysing data for new insights, but more at discovering and sift through available data (structured and unstructured) and possible data sources. It is largely based on metadata to provide users with an overview of what data is available and help them in deciding which available data to use, e.g. for free ad-hoc analysis, and how to sample it. It also includes functionality to search for specific data items or documents utilizing free-text search (e.g. based on Apache Lucene / Apache Solr).

Data Distribution

The data distribution component is partly derived from requirement VOL2 and its sub-requirements. It contains all functionality that aims at distributing and making data and analysis results available either to human users through a **user interface** or to other applications that further work with them through an **application interface**.

Data Distribution - User Interface: One or several user interfaces are used for visualization and presentation of data and results to end-users. This can e.g. include portals or distributing reports as pdf by e-mail. It decoupled presentation, which is typically done at a client, from analytical functionality, which is done closer to the data in an application server layer or even in-database.

Data Distribution - Application Interface: Data stored in a 'big data' system landscape and analysis results are often provided to other applications. This can include technical interfaces and APIs to access data and processing results from the system as well as message passing and brokering functionality. Note, that data consuming applications can very well be identical to data sources, in which case the communication represents some feedback loop.

Furthermore, there are also supporting functions, which support several of those steps and cannot be assigned to a single phase. These functions are shown horizontally and while they can possibly interact with each of the functional steps, this is not included in the diagram for readability reasons, as drawing a connector from each functional step to each supporting function would mess up and complicate the diagram.

Data Storage

The data storage component is directly derived from requirement VOL1. It includes all data stores, either temporary for caching or persistent, along the data processing pipelines. There can be several of those stores and besides the processed data itself, data storage is also important for metadata. On the other hand, data storage is a supporting function and has no direct influence onto the analysis result. From a high-level view, storage is therefore a vertical function, that supports several functional components, either by storing its input, its results or both. In a detailed view this will be broken down into concrete data stores along the processing pipeline. Generally, several sub-functions of data storage can be distinguished: **staging**, **data management oriented storage**, **sandboxing** and **application optimized storage**.

Data Storage - Staging: Staging refers to temporarily store data after it got extracted from its source to allow cleaning, integration and transformation routines on this data. It effectively leads to a faster de-coupling from the source systems decreasing the strain put onto them. It also enables faster access to the raw data, so it can already be used while the cleaning, integration and transformation processes are still under way and before it got loaded into a durable data store.

Data Storage - Data Management oriented Storage: Data management oriented storage refers to long-time data stores, which are heavily managed, e.g. considering available metadata and schema. The data is typically heavily cleaned and integrated providing a centralized, enterprise-wide notion of the idea of 'single version of the truth'. The idea is very comparable to a traditional Enterprise Data Warehouse, especially the basis database[2]. It is often used to distribute data to other, distributed data stores.

Data Storage - Sandboxing: Sandboxing can directly be derived from requirement VAL3 (Support Experimentation). It refers to temporary data stores, that are created for single users, user groups or departments to play and experiment with data, data processing and analysis techniques. The data is copied from various sources along the 'big data' system, e.g. from the data management oriented data store or from source applications just for experimentation. The users of the sandbox are free in what processing techniques they want to use for experimentation, typically incorporating free ad-hoc analysis and deep analytics functionality, but also ad-hoc cleaning and integration. The sandbox allows them, to experiment with this data as well as with new ideas of processing the data and to manipulate the data without risking negative impacts onto other analytical applications. Some of

[2] compare Section 2.3.1

the gained insights and results can be incorporated back into the processing pipeline (e.g. machine learning techniques and models which were found valuable). Furthermore a sandbox environment is a good place for data analysis tasks that are non-recurring as the source data can be copied raw for one time use and processed ad-hoc without imposing high efforts for modelling it, extending a global schema or creating cleaning and integration procedures.

Data Storage - Application Optimized Storage: Application optimized storage is comparable to the concept of data marts in traditional data warehouse architecture[2]. Data is typically copied from the data management oriented storage area and loaded into the application optimized storage area, while the data is transformed and stored optimized for the respective analysis objective and application. This can either refer to schema transformation, e.g. transforming the data into a star schema[2], or to the database technologies used to store the data, e.g. the usage of column-oriented storage[3]. Furthermore, an application optimized data store only holds the subset of all available data necessary to the respective application and the data is enriched during the transformation and loading process using information from value deductions or deep analytics results (e.g. a classification of dagta records based on some machine learning model).

Metadata Management

The metadata management component is directly derived from requirement VAR4 and its sub-requirements. Metadata management refers to the extraction, creation, storage and management of structural, process and operational metadata. The first type describes structure, schema and formats of the stored data and its containers, e.g. tables. Process metadata describes the different data processing and transformation steps, data cleaning, integration and analysis techniques. Operational data desribes the provenance for each data item, when and from which sources it was extracted, which processing steps have been conducted and which are still to follow. Structural and process metadata needs to be provided for all data structures and steps along the processing pipeline and operational metadata needs to be collected accordingly. Metadata management is therefore a vertical function consisting of **metadata extraction**, **metadata storage**, **provenance tracking** and **metadata access**.

Metadata Management - Extract Metadata: This function is directly derived from requirement VAR4.2 (Extract Metadata). To store, manage and provide metadata information it is obviously first necessary to extract metadata. Extraction of metadata occurs of all data sources and storage areas within the ‘big data’ system [45, 169][203, pp. 74,77]. This can involve different techniques based on the nature of the data source, e.g. extracting it from the metadata section of microformats within a HTML page, extracting an additionally published RDF file or another schema file, extracting schema definitions form a relational database or using the XML Metadata Interchange (XMI) format. Sometimes metadata about data sources also needs to be manually prepared, e.g. adding information about expected timeliness and completeness of data from a certain source, but also adding enterprise-wide term definitions to data sources, data and key figures [203, pp. 68-71,74].

Metadata Management - Store Metadata: This is directly derived from requirement VAR4.1 (Store Metadata). The purpose is to provide a centralized metadata repository and storage area. Every metadata that is extracted or gathered during system operation is stored there and can be centrally

[3] compare Section 4.3

queried and accessed. This also means, that an overarching schema and model of the metadata needs to be established. Standard models or ontologies can help here, e.g. the Dublin Core standard [169].

Metadata Management - Track Provenance: Provenance tracking is derived from requirement VAR4.3 (Track Provenance). This function describes extraction and collection of administrative and operational metadata during the different data processing steps. This includes job logging of processing steps, run time of components and information about volume and timeliness of loaded data. It furthermore includes user statistics, data access and reporting logs[203, pp. 75-77]. Especially the logging of processing steps data items went through, probabilities and confidence intervals for statistical and machine leaning techniques and the timeliness of loaded data allows end-users to trace analysis results back to the data sources and make educated estimations about the reliability and relevance of data and analysis results [45, 169]. It is also the basis for calculating a trustworthiness score in analytical applications as stated in requirement VER1.2 (Track Tustworthiness) [45].

Metadata Management - Metadata Access: Metadata access is derived from VAR4.4 (Inspect Metadata) and allows users and administrators to access stored metadata, to manipulate and to enhance it. The later is often necessary for business metadata, which cannot be extracted from technical sources, e.g. adding a business glossary, term and key figure definitions [203, pp. 68-71].

Data Lifecycle Management

The data lifecycle management component is directly derived from requirement VAL4 and its sub-requirements. It includes all activity aimed at the management of data across its lifecycle from the creation to its discarding. It is typically based on a rules-engine (**Rule-based Data and Policy Tracking**), which determines the value of data items and accordingly automatically triggers **data compression**, **data archiving** or discarding of stale data [202, pp. 133-140]. It is also used to move data between different storage solutions and triggering processing steps according to direct rules, e.g. scheduled jobs or according to data rule-derived value, e.g. moving data from an in-memory database to a disk-based database if it is less used. Data lifecycle management needs to be conducted along the processing pipeline without having direct influence onto analysis results and is therefore a vertical function.

Data Lifecycle Management - Rule-based Data and Policy Tracking: This function is derived from requirement VAL4 (Data Lifecycle Management). Data Lifecycle Management tasks should be automated as much as possible and allow the definition of processing schedules as well as rules and triggers, e.g. based on the rule-derived relevance of data or on the volume of a certain data store. Relevance of data can itself be determined based on rules, e.g. based on the age of the data. These rules should comply to company policies and governmental regulations and they should automatically be monitored and actions should be triggered accordingly [203, pp. 134-137].

Data Lifecycle Management - Data Compression: This function is derived from requirement VAL4.1 (Data Compression) and includes techniques to compress data in the different storage areas. The compression itself can be triggered based on rules, see Rule-based Data and Policy Tracking [202, pp. 137-138].

Data Lifecycle Management - Data Archiving: Data Archiving is derived from requirement VAL4.2 (Data Archiving). This includes storage areas for archiving data, procedure to do the archiving and triggers to run these procedures based on rules [202, pp. 137-138].

Privacy

The privacy component is directly derived from requirement VAL5. It includes all methods and techniques used to ensure privacy of the stored data, namely **authentication and authorization**, **access tracking** and **data anonymization**. To be effective these techniques obviously need to be adapted during all steps and data stores along the processing pipeline with a special emphasis on the data distribution component, which is typically the point-of-access to data and functions within the system. It is therefore a vertical function. [81][203, pp. 79-99]

Privacy - Anonymization: Anonymization is derived from requirement VAL5 (Privacy). Often, governmental regulations and company policies require data about customers, employees etc. to be anonymized before it gets presented to end-users. This is even harder if several data sources are merged as cross references can be used to de-anonymize data [160]. Therefore sensitive data needs to be identified, tagged in the metadata repository and accordingly anonymized or de-identified. Options are to manipulate certain data fields replacing or even deleting values, aggregating data so it does not refer to single persons anymore, adding noise to the data or swapping values between records [160][203, pp. 84-86].

Privacy - Authentication & Authorization: This function is also derived from requirement VAL5 (Privacy). It adds centralized services to identify users, e.g. based on a password system, and selectively grant them authorization to sensitive data only in the extent which is necessary for their job function and which is unproblematic [81, 169].

Privacy - Access Tracking: Access tracking is derived from requirement VAL5 (Privacy). Access tracking is based on Authentication & Authorization and logs who request access to which data sets and items, if that access request is valid and if and how data is manipulated during each access [81][203, pp. 98-99].

Comparison to traditional data warehousing functionality

From its principle structure and the idea of pipelining data, the reference architecture looks similar to data warehousing architectures and processes as described in Chapter 2.3.1. In those functionality is also broken down into different phases or stages, which the data sequentially goes through. This is very similar to the processing pipeline used above.

Compare for example the architecture proposed by Bauer and Günzel [61, pp. 37-86]. Some of the components in their data warehouse architecture can easily be mapped to the 'big data' functions proposed here. Both contain a data extraction component and the first data transformation step in the data warehouse architecture can be mapped to the data cleaning and data integration functions. Others have more subtle similarities, the different data stores in the data warehouse architecture (staging area, basis database, derived database and analysis database) are e.g. all included in the vertical data storage function, but it exemplifies the same idea of data getting stored along the processing pipeline. Furthermore, the components in the upper part of the data warehouse architecture, namely

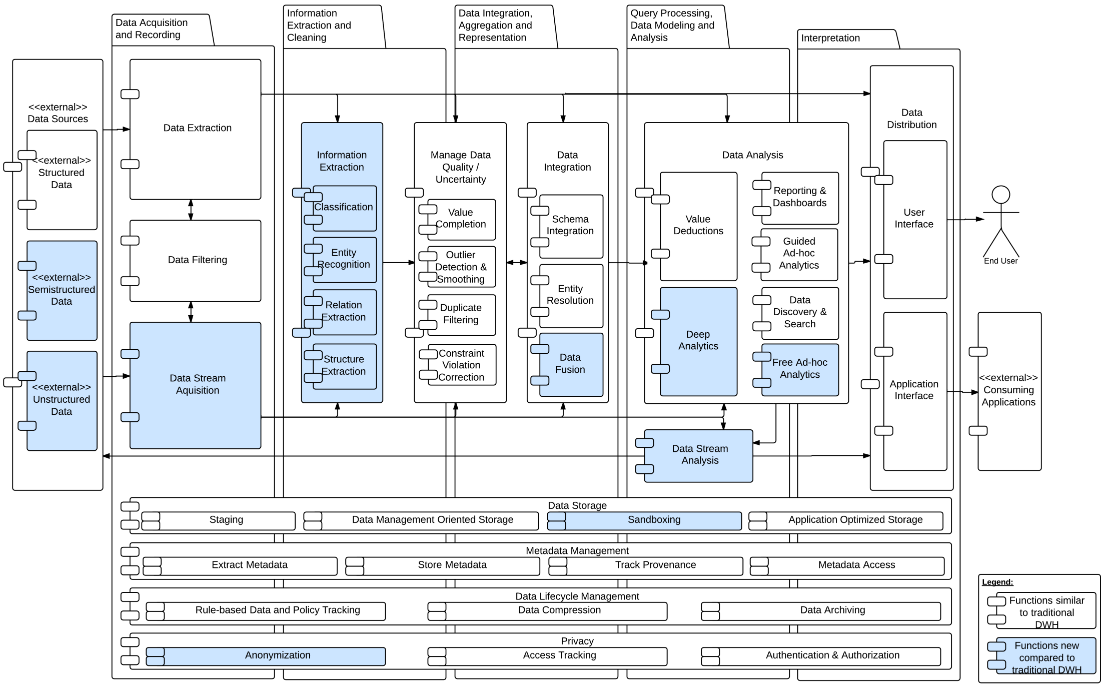


Figure 4.2: Reference Architecture: Detailed functional view

the second transformation step and the loading into the derived database and the analysis database are all subsumed in the data analysis function of the ‘big data’ reference architecture. The analysis database itself and the final analysis component are both partly included in the data analysis and the data distribution component of the ‘big data’ reference architecture.

Furthermore there are also some differences that should be emphasized. In the ‘big data’ reference architecture, the processing pipeline is coupled more loosely. While the sequence in traditional data warehouse architectures is strict and data is typically persisted after each step, for ‘big data’ several steps can be skipped and the data volume does not always allow to persist data in each layer. Furthermore, traditional data warehousing does not include stream processing components[4] including a feedback loop back to the data source. Of course there are also differences within the single components. The data extraction and storage component e.g. needs to consider unstructured data, which is not the case in traditional data warehousing. This also means, that the information extraction component is new, as it especially aims at handling unstructured data. One should however note, that there are indeed efforts to build an ‘unstructured data warehouse’, see e.g. Inmon’s Data Warehouse 2.0 approach [136].

The similarities and differences get even more apparent, once detail is added to the functional reference architecture (see Figure 4.2). The diagram also depicts functions, that are new (e.g. data stream analysis) or fundamentally more important (e.g. deep analytics and sandboxing) compared to traditional data warehouse architectures colored in blue.

Of course, this comparison is to some extent uneven as the data warehouse architecture from Bauer and Günzel [61, pp. 37-86] desribes a more implementation-oriented view, while the view discussed here emphasizes general functions. However, the comparison shows, that there is a large overlap

[4] even though there are efforts for real-time data warehousing, which typically includes direct access to the data source with virtual data cubes

between traditional data warehouse and newly emerging ‘big data’ architecture. This is the reason, why I believe that an Enterprise Data Warehouse still has a core place in a ‘big data’ architecture. This will become more apparent in section 4.2.2, where I present the implementation-oriented view of the reference architecture. Several patterns from the data warehouse architecture will also be present in there, but extended with components to tackle the new requirements of ‘big data’. In general, once can see this as an evolutionary, rather than a revolutionary rip-and-replace development.

4.2.2 Reference Architecture Development View

After preparing a view onto the system functions needed in the last section, in this section I will map them to more concrete and implementation oriented system components. An architecture diagram is depicted in Figure 4.3 and will be explained afterwards. During designing the architecture I considered and incorporated several design principles taken from literature. First I will shortly introduce those principles and later during the architecture description I will point back to them where they were applied.

One very basic idea, picked up and expressed by Marz and Warren [164, pp. 8-10], is the distinction between basic and derived data and according conclusions for the architecture of ‘big data’ systems. Basic data cannot be derived from any other data item and is often a transaction of event that triggers the data recording. The sales amount of a product is e.g. derived from the basic data of each single purchase transaction or the ‘friends’ count on a social media application is derived from the single ‘friend’ relationships which themselves are derived from the basic data of ‘add friend’ and ‘delete friend’ transactions. This idea leads to the design principle of **immutability**. The basic idea here is to keep basic data in a separate store, where data is not updated only added, therefore the basic data is immutable. Consider e.g. the social media ‘friend’ example from above, where a ‘delete friend’ record is added instead of deleting or updating the initial ‘add friend’ record. In cases, where data sources provide updated records these are just added with a timestamp, e.g. if the address of a customer is changed. All other information can then be derived from this basic data and will be stored in another data store which is either regularly and entirely recomputed or indeed enables updates.

Where applicable, this idea provides several advantages. First, due to timestamping and keeping all basic transactions, it enables analytical applications to take the entire history into account, do comparison over time and time series analysis. Considering the social media example again, it allows applications to take into account that somebody had a ‘friend’ relationship with somebody else before, but does not anymore. This is already more information than the pure absence of said relationship. Second, it provides (human) fault tolerance. The basic data cannot be deleted as it is immutable and if derived data gets accidently deleted it can always be recomputed from the basic data without the need to extract it from the source system again. This is especially important considering, that source systems might not hold and provide data forever. Third, it takes away the need for random writes in the database system and therefore reduces complexity. Locking mechanisms are e.g. not necessary anymore Marz and Warren [164, pp. 33-36].

Another design principle is to design the architecture with a **‘data highway’** [144]. This is especially useful for incorporating streaming data into manual analysis. The basic idea here is to use a series of temporary data stores (Kimball calls them caches), which are connected via ETL processes for information extraction, data cleaning and integration. Therefore the data quality increases along the ‘data highway’, but the latency of the data does so to. Analytical applications and their user can then use the data store best suited for the particular need considering the data quality / timeliness trade-off. Data used within the streaming analysis applications is therefore the one extreme being real-time but raw and not integrated, while the Enterprise Data Warehouse would be the other extreme being highly integrated and cleaned but with data being updated in batch often only once a day.

Another principle, which already was adopted in the requirements specification VEL1.1.1.2, is to **pre-compute results** and apply the where possible. This is especially useful for real-time stream processing. The idea is to compute intermediate results and models within a backend function over data acquired from a stream on a regular basis, e.g. each 5 minutes, and make them available to a frontend function, which uses them to analyse or respond to the data that is just flowing in from a stream in real-time [170]. There is also a set of principles proposed by Begoli and Horey [63], namely the support of a variety of analysis methods, the need to process structured and unstructured data in different data stores as well as to incorporate batch processing and data preparation steps and finally the accessibility of data by different user groups and target applications. I believe again, that all these principles have already been considered during the requirements specification.

Figure 4.3 shows a diagram of the proposed reference architecture. As discussed in Section 4.2.1, there are several components, that are similar to traditional data warehousing architectures. These are again coloued blue. Note also the different notations for software components that represent processing and transformation of data and data stores. From a structure point of view one can roughly interpret the diagram as depicting a data flow from left (data sources) to right (analysis applications), while the upper part of the diagram targets processing of data at-rest and the lower part targets processing of streaming data.

The distinction between **Structured Data Sources**, **Semi-Structured Data Sources** and **Unstructured Data Sources** is taken over from the functional architecture as the type of data source impacts the later data flow. Of course, there can be several data sources from each type, they are however modelled as one for the sake of simplicity. Technically, data sources can be quite diverse[5] and should not be limited considering a reference architecture. Most often, however, they refer to different types of databases. For structured data these are typical relational, for semi-structured or unstructured data these can be XML or document stores, key value stores, column-family stores or graph databases[3]. In other occasions they can e.g. refer to web sites or applications that expose data via API's, but they are not limited on that.

For each data source, there typically is one **Extraction** and one **Monitor** component, both together implementing the Data Extraction function from the functional architecture. The monitor's task is to identify changes in the data source so they can be incrementally extracted and propagated to later functions. There might be situations, where a monitor is not necessary for data extraction from a source as the data volume in the source is small enough to allow for a full extraction and load of the data with old data being entirely overwritten. Most of the times this is however infeasible due to data volume and its effect onto performance of the ETL process. Therefore monitors are needed for an incremental extraction of the according data sources. The extraction component then takes the task of reading changed data from the data source and load it into the 'big data' system environment, in which store exactly depending on the type of data. It can either be directly triggered by the monitor in case of (a certain threshold of) changes or being regularly scheduled just using the information of the monitor to decide about the delta of data to extract [61, pp. 87-94].

The implementation and functionality of both, monitor and extraction components, depend very much on the nature of the underlying data source and which techniques they support. Monitoring can e.g. be trigger-based with the data source informing the monitor about changes, based on examining log files to identify new and changed data or based on timestamping the data. Sometimes this requires adjustments to the data source, e.g. a timestamping mechanism or a mechanism to message the monitor with information about data changes. The data extraction is obviously dependent on the

[5] compare to Section 2.1.4

interface the source offers. This can mean (but is not limited to) extracting data via a standard database interface (typically ODBC), via calling a proprietary database or application interface (e.g. in the case of some so called ''NoSQL' databases[3]), via lower level file system operations in the case of file interfaces or even crawling web sites [61, pp. 87-94].

After the extraction data gets either transformed and loaded into a traditional Enterprise Data Warehouse environment or loaded into the Raw Data Archive depending on the nature and relevance of the data. The **Enterprise Data Warehouse**[6] will have his place in a 'big data' environment for several reasons and fulfil a couple of functions. First, it fulfils the function of a data management oriented storage area to persist relevant results of the processing steps Information Extraction, Manage Data Quality and Data Integration. In that it still fulfils the traditional function of a data warehouse as being the central storage area for carefully cleaned and enterprise-wide integrated data, therefore called to present the 'one version of the truth'. In a 'big data' environment this role is however constrained as the Enterprise Data Warehouse can only realize it for directly relevant and structured data, where structured data also includes structure extracted from semi- or unstructured data during the Information Extraction step. Furthermore, the Enterprise Data Warehouse has a distributional function, as it provides and is the single source for cleaned and integrated, structured data for analytical applications. The on-going importance of the Enterprise Data Warehouse is also backed up by a study of the Data Warehousing Institute from 2011, which shows high business commitment to a central data warehouse, but also a good potential growth for other technologies to augment the data warehouse for work loads it is not designed and suitable for [194].

The process of getting data into the Enterprise Data Warehouse is very comparable to an ETL process in traditional data warehousing architecture. This is reflected by the components **Staging Area**, **Data Cleaning**, **Data Integration** and **Data Load** and the ETL process as well as the role of the staging area very much compares to the explanations in Section 2.3.1. They obviously map to the according functions in the functional view from Section 4.2.1, Manage Data Quality, Data Integration and Staging. Note at this point, that especially the Data Integration, but also the Extraction, component can be a bottle-neck, not regarding performance, but regarding how quickly new data sources can be connected to the system. As discussed in Section 4.2.1, manually creating mapping rules takes time, but this is still the most widely used practice. There are two possible solutions, which were also mentioned when describing the Schema Integration and Entity Resolution functions. One is to use bootstrapping methods, that is automatically create integration rules using probabilistic schema mappings. Another is to use ontology-based schema mappings. Furthermore, note the backflow from the Enterprise Data Warehouse to the Data Integration and Data Cleansing components. Already loaded data can e.g. be used as a basis for Outlier Detection and probabilistic Record-Linkage methods within those components.

The difference is, that the Staging Area can also be loaded from a **Raw Data Archive**, optionally using the **Information Extraction** component. The Raw Data Archive is also a persistent storage area, but a counterpart to the Enterprise Data Warehouse in the sense that it acts as a storage area and archive for all raw data and data that is not directly loaded into the Enterprise Data Warehouse. The reason can be that, this data does not fit into the Enterprise Data Warehouse (e.g. unstructured data) or that it is not considered relevant enough to put the effort into cleaning and integrating it, but should still be stored for future reference[7]. In this sense, it has a function as a lower-cost (compared to the Enterprise Data Warehouse) data sink and archive. Note, that often the Raw Data Archive is also combined with the Staging Area, where all extracted data gets loaded into the Raw Data Archive, while highly relevant data gets transformed, cleaned and integrated within it before it gets further loaded into the Enterprise Data Warehouse.

[6] compare Section 2.3.1

[7] compare to the discussion in Section 2.1.2

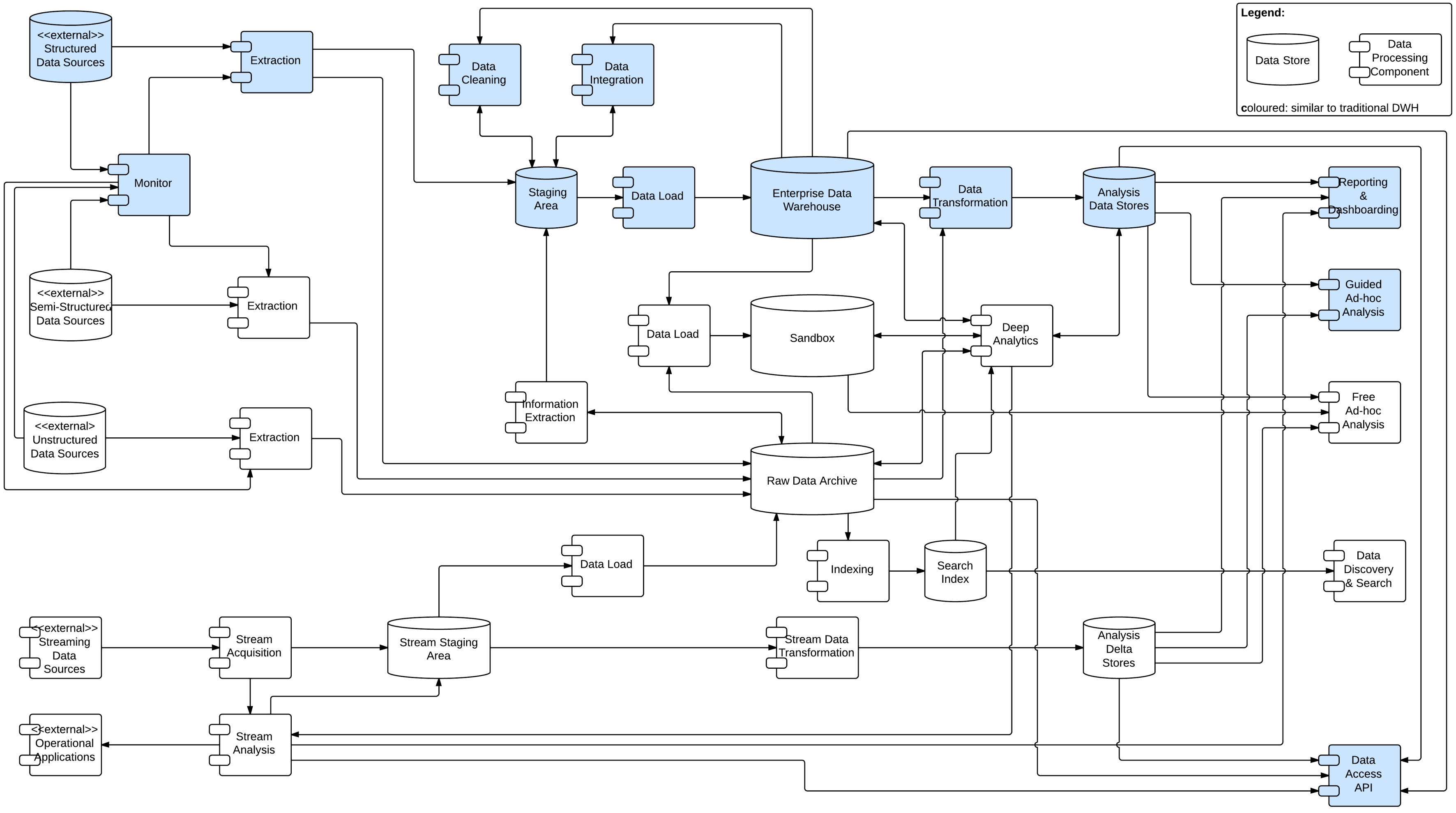


Figure 4.3: Reference Architecture: Detailed implementation-oriented view

The Raw Data Archive also fulfils the function of persistent storing and managing unstructured and semi-structured data and to integrate it with the structured data in the Enterprise Data Warehouse. This integration can be done in two ways. First, be using a common, logical key space, where data from both storage areas can reference each other. An insurance claim transaction stored in the Enterprise Data Warehouse could e.g. reference a textual claim description stored in the Raw Data Archive. Additionally, data from the Raw Data Archive and the Enterprise Data Warehouse can be further integrated during data analysis, either in the according analysis-oriented data storage areas or directly within analysis tools. Second, the **Information Extraction** component fulfils the according function from the functional architecture, that is extracting structured information from the unstructured data stored in the Raw Data Archive. This extracted information can then be loaded into the Staging Area, incorporated into the general ETL process and therefore integrated with other structured data. Note, that there can also be a back-flow of extracted information to accompany the raw data stored in the Raw Data Archive. Furthermore, in the sense of storing and providing access to raw data long term, the Raw Data Archive supports requirement VAR3.3 (Original Sources).

The main function of both, Enterprise Data Warehouse and Raw Data Archive, is storage and management of data. Analysis applications, however, often only require an extract of all available data, do not necessarily need data in the highest level of detail but aggregated and work more efficiently on top of an optimized structure, e.g. a star schema. Tackling this function of application optimized storage, as described in Section 4.2.1, is task of the **Analysis Data Stores**. These store physically persistent views, which are built from the data in both, Enterprise Data Warehouse and Raw Data Store, and loaded into the analysis data stores via the **Transformation** component. In that, the Analysis Data Stores can be seen as applications of the principle to pre-calculate intermediate results.

Furthermore, there are two more storage areas located within the system. One of these is the **Sandboxing Area**. Sandboxes obviously aim at fulfilling the function sandboxing from the functional area and therefore supports requirement VAL3 (Support Experimentation). They can be created and filled as required by loading data from the Enterprise Data Warehouse and the Raw Data Archive. The data in a sandbox is typically temporary and based on a certain experiment or analysis project. Using a sandbox can be necessary for several reasons, most of the time because there is no analysis data store available, which holds the data modelled in the way it is required for an experiment and it is not suitable to build one for the task at hand, e.g. because it is a one-time analysis and there is no need for an on-going support of that task. Another reason is if the data needs to be manipulated within the experiment which would interfere with the usage of the analysis data store by other applications.

The last data store is the **Search Index** built over the data in the Raw Data Archive using the **Indexing** component. It supports the Data Discovery & Search function by indexing unstructured data from the Raw Data Archive and therefore allowing to search through it based on free text queries. Typically this is achieved by using an inverted index extended with additional information, e.g. the author of a text or other document related metadata. The Search Index is then used by **Data Discovery & Search** component, which implements the actual, user-initiated search process through the data. The index can however also be used by other applications, e.g. as input for a deep analytics task.

Analytical applications, representing the different sub-functions of Data Analysis in the functional architecture view, are on top of these different data stores and access the data stored. This involves the directly user-facing analysis applications Reporting& Dashboarding, Guided Ad-hoc Analysis and Free Ad-hoc Analysis as well as Data Discovery & Search, but also Deep Analytics tasks. The distinction of those different applications is already explained with the respective sub-functions in Section 4.2.1, so a deeper explanation shall be omitted here. Additionally, data from the persistent data stores, Enterprise Data Warehouse, Raw Data Archive and Analysis Data Store can be made available to other applications via the **Data Access API**.

Deep Analytics applications can either be scheduled and re-occurring tasks or they can be built as part of an one-time experiment or analysis project. In both cases they typically do not include direct presentation of results to users, but results, mined rules and models are written back and stored in the respective storage areas, where they can be accessed by other analysis applications for discovery and interpretation. In the first case, the input data is taken from the persistent storage areas of Analysis Data Stores, the Raw Data Archive and possibly it Search Index, while results can be written back to the Raw Data Archive, Analysis Data Stores, but also to the Enterprise Data Warehouse if they are considered relevant enough to become part of the integrated core data. In the second case, input data is taken from a sandbox, while results are written back to the same sandbox. These results are typically not incorporated into the persistent data stores, while insights from experimentation can lead to derive and develop a re-occurring deep analytics tasks based on those.

The user-facing analysis applications, **Reporting & Dashboarding**, **Guided Ad-hc Analysis** and **Free Ad-hoc Analysis** are all based on respective Analysis Data Stores. Considering this connection, it makes sense to design Analysis Data Stores in a way, so they can support several analysis applications, to make governance and management of these stores easier. However, if an analysis applications has very special requirements this can lead to the design of an exclusive Analysis Data Store. Additionally, Free Ad-hoc Analysis applications can also be based on a sandbox for experimentation, while Data Discovery & Search applications are obviously based on the Search Index.

Considering the design of the architecture, especially the different storage areas, but also the flow of data from extraction to the analysis applications, one can identify the principles of basic and derived data and immutability described by Marz and Warren [164] as well as above. The basic or raw data is stored in the Raw Data Archive, allowing (parts of) all other data stores to be reconstructed if necessary. Data is derived and manipulated on multiple levels, first by using information extraction, cleaning and integrating the data before loading it into the Enterprise Data Warehouse. Data is however not aggregated, so data in both stores is on the same level of detail, allowing cross-referencing of data between both stores. While data in the Raw Data Archive is always immutable and append-only, the data in the Enterprise Data Warehouse typically is too. Data can however be deleted from the Enterprise Data Warehouse or timestamped version of the same data item can be collapsed if the information is not relevant enough anymore. The next level of derivation is when data is transformed and loaded into the Analysis Data Stores. Data there is typically not immutable, as aggregates are regularly changed when data items are added to them. Furthermore, Analysis Data Stores can always be reconstructed from the Enterprise Data Warehouse and the Raw Data Archive due to their derived nature.

Besides analysing data at rest as described above, there is also the need to process data in motion. As described before[8], there are two general tasks, first analysing in-flowing data in real-time and reacting to it accordingly and second acquiring streaming data and incorporating it into the regular data processing pipeline. The in-flowing data itself gets generated in some transactional system, the **Streaming Data Source**. The nature of possible data sources and in-flowing data can be quite diverse, ranging from system that record machine generated data from sensors to social media streams and high-volume transactional business systems[9].

[8] see Section 3.2.2, especially requirement VEL1 and sub-requirements, and Section 4.2.1

[9] for a overview of possible data sources see Section 2.1.4

The **Stream Acquisition** component's task is to acquire the data stream from these sources and load it into the 'big data' system. Stream acquisition can e.g. involve subscribing to a publishing service or streaming API, simulating the data stream by repeatedly calling an API or even extending the source application to create a create a message stream and push it to the Stream Acquisition component.

From the Stream Acquisition component in-flowing data gets loaded into the **Stream Staging Area**. Its function is similar to the regular Staging Area , to temporarily hold the data until they are further processed. This occurs in two ways. First, the data is written into the Raw Data Archive, from where it gets via the Information Extraction component into the Staging Area and into the regular processing pipeline. Typically, the processing in the regular pipeline however happens in periodic intervals, e.g. over night. Sometimes this waiting time is not acceptable. To avoid it, the streaming data (or important parts of it) can be preliminarily transformed using the **Stream Data Transformation** component and then be loaded into **Analysis Delta Stores**. This transformation and loading can either be an on-going process or data can be gathered for a short period (e.g. a minute) with a periodically occurring transformation. Additionally, because of timing constraints, the transformation typically does not involve an as thorough data cleaning and integration compared to the regular ETL process. In that, it trades off data quality and integration against timeliness of available data. The data in the Analysis Delta Stores can then be accessed by different analysis applications and via the Direct Access API and it can be joined with more historic data from the Analysis Data Stores. Once, the respective streaming data went through the regular ETL process and is available in an Analysis Data Store, it can be deleted from the Analysis Delta Stores.

The second task is analysing streaming data and reacting to it. It is best practice to place this component as close and ideally within the operational system, that processes the data in the first place and is responsible for handling these transactions, e.g. business transactions. The need to communicate with another system over a network is detrimental to applying analysis results in real time and reacting to them, that is adjusting the operational processing of the respective transaction or just giving feedback to the user. Therefore, the **Stream Analysis** component should be placed within the operational application and integrated with it, if this is possible, rendering it an external component from the view point of the 'big data' system.

However, this is not always possible as there are use cases, where data from several sources needs to be streamed in, the data streams need to be joined and analysed in combination and results need to be streamed back to different Operational Applications. Note, that these can be identical with the Streaming Data Source, but do not need to be. In this cases it is necessary to have a central Stream Analysis component, which can be placed within the 'big data' system. In those cases, streaming data gets acquired as before and the Stream Acquisition components directly forwards the acquired data streams into the Stream Analysis component. The data streams should be directly forwarded instead of the Data Analysis component accessing them from the Stream Staging Area as persisting the data in between takes time and works against the 'real-time' requirement.

In both cases, placing the Stream Analysis component within the data source or placing it within the 'big data' system with several data streams flowing in, it has several interfacing points with other components of the 'big data' system. First, models, rules and intermediate results created from deep analytics tasks can be made available and sent to the Stream Analysis component as required, where they can be joined with and applied to the data streams. Again, here can be referred to the principle of pre-computation of intermediate results. Second, results from the Streaming Analysis component can be written to the Stream Staging Area to be joined back with the actual streaming data from the Stream Acquisition component. Third, the Stream Analysis can lead to alerts for human actors. It can make sense, to send these alerts and integrate them into reports and dashboards for an according topic. Fourth, streaming analysis results can be made available through the Data Access API, where applications outside the ‘big data’ system can subscribe to for getting this result stream forwarded.

Additionally note, that there were several parts of the functional view, which do not directly show up in implementation-oriented view. These are most horizontal functions, which need t connected to almost every other component of the reference architecture. One of them is metadata, another one is Data Lifecycle Management. There is also no Monitoring and Management component in general. These are left out of the diagram above mainly for clarity reasons. The Metadata Management component is connected to every Data Processing as operational metadata and provenance information gets extracted while data is processed and respective jobs run. Additionally it is connected to every data store to extract structural metadata. The Metadata Management component manages the extraction of metadata, stores it within a Metadata Repository and is the single-point-of-access to this repository. It offers metadata out of the repository via an API to analytical and data discovery components, but possibly also to external applications.

Data Lifecycle Management functions need to be supported by every Data Store component. They need to support data compression and archiving (which can involve to move data from the Enterprise Data Warehouse or Analysis Data Stores to the Raw Data Archive) internally, which can then be globally triggered by a Data Lifecycle Management component. This component receives data from the Metadata Management component, uses them to check rules in a rules engine and triggers appropriate lifecycle tasks within the single data stores.

A similar statement applies for Privacy. Authentication and authorization as well as access tracking functions need to be supported by every single user-facing component and by the different data stores. Anonymization tasks can be conducted during Data Transformation before loading data to an Analysis Data Store, where they can be accessed by end-users. However, this might not be enough. Often it is not suitable to store or process non-anonymous data anywhere in the ‘big data’ system. In these cases, anonymization needs to be done as part of the Data Extraction component or as a distinct component directly following Data Extraction and Stream Acquisition.

One principle that is followed by the structure of the stream processing components it the principle of the ‘data highway’ proposed by Kimball [144] and described above. The first cache is within the streaming application itself, based on the Stream Analysis component, and allows the intermediate and automatic analysis and reaction to the raw data with the results flowing back into the operational systems and being accessible through the Data Access API or within Dashboarding & Reporting applications. The next cache are the Analysis Delta Stores, which includes partly cleaned and integrated data, which is not real-time but possibly available within minutes and between the periodic ETL runs. Finally, the last cache are the Analysis Data Stores, which provide thoroughly cleaned, integrated and transformed data, but only with a time lag depending on the scheduled period for ETL runs, typically within hours over even a nightly load. Additionally, it is possible to add another cache, by allowing access to the Stream Staging Area by analysis applications. This would allow to manually access raw data from streams that is slightly more timely, possibly within seconds, compared to the Analysis Delta Stores, but not cleaned or integrated at all.

4.3 Reference Architecture Technologies

This section will describe and discuss technologies that are connected with 'big data'. The goal is to identify strengths and weaknesses of each discussed technology and place them into the reference architecture[10]. The discussion is structured in a way, that first the focus will be on infrastructural technologies that are widely recognized and used to build 'big data' systems: database technologies considering relational as well as so called 'NoSQL' databases and Apache Hadoop and its ecosystem. Keep in mind however, that this can by no means be an extensive list of all possible options. Given the hype around 'big data' and technologies around it, but also 'NoSQL' databases the space is in a stage of flux with new products being constantly released. Therefore I will focus on general techniques, ideas and models as a discussion of each single product would first explode the scope and space of this thesis and second be stale very soon.

4.3.1 General Developments and Influences onto Database Technology

Divergence in the database industry and 'polyglot persistence': The database industry has seem a lot of movement during the last couple of years with the advent of the family of 'NoSQL' databases, more specialized relational databases sometimes referred to as 'NewSQL', but also the influence the Hadoop ecosystem had on data management. Both motivated the formation of a big start-up scene around those concept. This is especially mentionable, considering that before, except for academic research, the market was pretty stable with an ongoing development of existing systems, but without a lot of new products being released. The obvious choice for most applications were either one of the big relational, multi-purpose databases (mainly the Oracle, IBM DB2 and Microsoft SQL Server product lines) and smaller-scale relational open-source systems (mainly MySQL and PostgreSQL). The ideas of diverging technologies and products were mostly integrated into the former systems or did not create a lot of industry traction, e.g. in the case of independent XML stores[199, pp. 981-1023] and object-oriented databases[199, pp. 945-980]. The most notable exception of this rule were special-purpose, shared-nothing distributed Data Warehouse systems in the likes of Terradata, Netezza, Greenplum and others. One reasion for this new development originated from web applications and a growth of data and workload with the need for distribution to highly scale with a transactional workload of mixed read/write operations. Another reason was the need for complex analytical and read-heavy computations over a large volume of data. Besides these main reasons, there are still several small ones, which I will describe in the discussion of the respective sub-category below.

Another idea, that co-emerged with these technological divergence, drove them but is also driven by them is the idea of 'one size does not fit all' or 'polyglot persistence'. The idea of 'one size does not fit all' was first introduced by Stonebraker and Cetintemel [204] and got further explained and developed in a couple of papers [204, 206, 209, 210]. Initially focussed merely on relational database systems, the idea predicted that the time of large multi-purpose systems would be over, in favour of special-purpose systems with a leaner architecture optimized for certain workload characteristics. Namely they predicted four application areas that databases would be specially designed for, text databases, data warehouse systems, stream processing or high-frequency transactional systems and scientific applications. They further predicted, that each of this special-purpose systems would outperform multi-purpose systems by a factor of 10 in its respective field [209]. They further elaborated on some unique characteristics of data warehouse systems and stream processing systems and how the architecture of special-purpose systems would be optimized for those [204, 206]. Some examples,

[10] see Section 4.2.2

they mentioned for data warehouse systems was the use of column-oriented[11] instead of row-oriented storage, emphasis on materialized views and favouring bit-map indexes over B-trees.

With the advent of 'NoSQL' systems and new data models in those systems, the very similar concept of 'polyglot persistence' was introduced. The idea was again, that different applications within an organization would rely on different database systems depending on which data model and architectural characteristics best fit the particular application, but even more, that one application would use several database systems based on the different tasks the application has to perform [187, pp. 7] [113, pp. 133-140,147-152]. This is very different from the previous approach of using a certain relational database product almost for every data storage need. This is also very applicable to the 'big data' reference architecture considering the different storage areas involved: Staging Area, Raw Data Archive, Enterprise Data Warehouse, Analysis Data Stores, but also Stream Staging Area and Analysis Delta Stores. In the remainder of this section, I will introduce several types of database systems and map them to the mentioned storage areas based on the characteristics of the former and the requirements of the later. Before doing this, I will however explain several concepts that have an influence onto the architecture and therefore onto the characteristics of these systems.

CAP theorem: Another idea, that is often used by 'NoSQL' manufacturers and proponents to argue for their architectural choices[113, pp. 53-56], is the CAP theorem first introduced in a keynote by Brewer [76] and a subsequent formalization and proof by Gilbert and Lynch [120]. While the theorem was stated in 2000 and in the context of distributed systems in general, with the need of scalability due to data volume and workload and concluding a more distributed architecture for databases, it also received a lot of attention in the database community lately. This motivated an on-going discussion about the theorem, which is e.g. reflected by an issue of iEEE Computer[12] which contained several articles especially focussing on and discussing the CAP theorem [37, 66, 75, 121].

In its original form, the CAP theorem stated that in a distributed system it is only possible to achieve two out of three desirable properties: consistency, availability and partition tolerance. In this context Gilbert and Lynch [120] defined consistency using a linearizable consistency model which states, that there must be a 'total order on all operations such that each operation looks as if it were completed at a single instant' [120]. More intuitively, this means that a request to a distributed needs to be atomic and behave the same way, as if it would be executed on a single node or in a database context as having a 'single up-to-date copy of the data' [75] . Later, Gilbert and Lynch [121] generalized and relaxed this definition requiring that a response to a request is the right one, depending on the specification of the operating service. Availability means, that whenever a request it receives a response and if a non-failing node, this node must eventually be able to respond. And finally, partition tolerance means, that the system should be able to cope with network partitions and allow for an arbitrary number of messages to be lost. [120, 121]

There are, however, several things to note. First, consistency in the context of the CAP theorem has a smaller scope than consistency defined within the ACID property. While the later covers consistency over the schema and the state after an operation adhering to database and foreign key constraints, consistency within CAP means that read and write operations always occur on the latest version of a data item [75]. Second, availability is only influenced by non-failing nodes being able to respond. Failing nodes do not impact availability in this scenario, as long as not all nodes fail, that hold replications of a certain data item. Third, partition tolerance is different to the other two properties. While a system architecture can be developed in favour of and therefore influence either, availability and consistency, it has no influence on network or node failures that result in separated partitions.

[11] see below for a further discussion of the concept

[12] Computer 45(2), Feb 2012

Network partitions are therefore a given, if they occur, and the system must cope with them with the probability of network and node failures increasing as the cluster grows bigger [37, 75, 121].

This third observation leads to the conclusion that the CAP theorem mainly implies, that a system needs to decide for availability or consistency in the situation of an occurring partition. One can think of this in an intuitive way. Consider a partition occurs and nodes on several partitions hold a replication of a data item. If a node receives a read or write requests it now needs to decide if it processes this request or not. In the former case, the node cannot be aware, if the respective data item was already updated on a node in another partition, therefore consistency is not ensured. In the later case the system sacrifices availability. [75] This essentially assumes, that the system replicates data over several nodes. If it does not, this would however imply that it gives up availability, as a non-reachable node makes it impossible to respond to any request that refers to data items that particular node holds.

One implication of this fact is, that according to the CAP theorem there does not need be a trade-off while the network is not partitioned. During all other times a distributed system can actually maintain both, even if many ‘NoSQL’ tend to give up strong consistency also during times in which the network is completely functional [44]. Defending this choice with CAP is one if the theorem’s big misinterpretations, but there can still be good reasons to do so. One of them is to increase latency and scalability. Maintaining consistent states over several replications can be expensive in time, therefore requiring to make a trade-off between consistency and latency[13]. For this reason, the CAP theorem falls somehow short as a guideline to reason about these properties. Abadi [37] therefore proposed another means to think about them. This resulted in the formulation of PAC/ELC. This should be interpreted as follows: If a system faces a network (p)artition does it choose (a)vailability or (c)onsistency? (E)lse, does it choose (l)atency or (c)onsistency. Obviously this is valid for choosing databases within a ‘big data’ system as well.

Implications of the CAP theorem and PAC/ELC onto database systems: Based on the observations above, several databases, especially in the ‘NoSQL’ field[14], are architected around weaker consistency models [56]. therefore trading-off consistency to gain either availability, latency or both. In this case databases are often referred to as following the BASE model, which means they are (b)asically (a)vailable, but can be in a (s)oft state and only provide (e)ventual consistency [76, 179]. Keep in mind, however, that BASE is not as rigidly defined as ACID, especially considering the meaning of ‘basically available’ and ‘soft state’ [113, p. 56]. In general, basically available means, that availability is the primary goal for these systems, and both read and write operations should be , in which the database has not one concrete state, but the state is in flux, that is at points in time one replica of a data item might differ from a replica on another node. Finally, eventual consistency means, that these conflicts will ‘eventually’ be resolved and that the system guarantees that if a data item receives no more updates, that current updates will ‘eventually’ be propagated to all replicas. [76, 179, 223][199, pp. 852-853]

The first implementation change when dropping ACID is to give up on distributed transactions. Distributed transactions as implemented in most distributed, relational databases rely on commit protocols, to ensure that all nodes that are touched by a transaction (either due to replication or due to sharding) commit or roll back a transaction together, and on distributed locking [199, pp. 830-847]. Distributed transactions are however expensive over-head wise and hurt latency and scalability.

[13] Further reasoning why this trade-off is necessary can be found in Abadi [37]

[14] While it would be possible to trade-off consistency and drop full ACID requirements in a relational database, to the best of my knowledge there is no product, neither open-source nor commercial, that does so

Therefore, many ‘NoSQL’ systems do not allow distributed transactions, but only transactions that just touch one data shard on one node or even only transactions over a single data object.

Considering the replication model, eventual consistency is typically implemented, by allowing each replica to accept and conduct both, read and write operations. Writes are then asynchronously propagated to other replications and, once a network partition got resolved, they will eventually reach all replicas for that particular data item. The time until writes are propagated is the inconsistency window. Note, that if other replicas also update the same data item during the inconsistency window, write/write conflicts can occur and need to be resolved by merging the updates. Typically this needs to be done within the application layer, where the database just informs about these conflicts [199, p. 853].

There are however also several other options how to relax consistency based on different replication and distribution models with different impact onto latency and availability. That is, why Brewer [76] talked about the decision between ACID and BASE not as a binary choice, but as a spectrum where a trade-off needs to be established. Lloyd et al. [156] e.g. describe COPS, a system that guarantees causal consistency. This consistency model is stronger the eventual consistency but weaker than strong, ACID-like consistency. The trade-off in this sense is based on the implemented distribution model. Options are e.g. master-slave or quorum-based replication models[15]. For example, a quorum-based protocol provides a better consistency than what is described above if the number of replicas required to agreed on a write (W) overlaps with the number of replicas required for a read (R), considering the total number of replicas (N). This is the case for $(W + R > N)$. In this case, a read operation always touches at least one replica that is aware of the latest write operation. On the other hand, availability is decreased, because if after a partition or after a failure of several there are not enough replicas available to fulfil the read or write quorum, the system is not able to conduct the respective operation. For the case described above, which could essentially be seen as a special instance of quorum based with $(R = 1)$, $(W = 1)$ and $(R + W < N)$, one replica within a partition is enough to serve both, read and write requests. Latency is also decreased, as there is the overhead of several replicas needing to agree.

Considering this example, it gets also apparent, that one can fine-tune a distributed database not only by choosing a certain distribution model, but also by configuring its parameters. Coming back to the quorum example, one can trade-off between read availability / latency and write availability / latency by shifting the number of required quoras either the read or to the write side, while $(W + R)$ remains larger than N. A quorum can e.g. be configured Read-One, Write-All to favour read latency and availability over write latency and availability. Many databases allow to be configured and fine-tuned internally. Cassandra can e.g. be configured using one of the consistency settings ONE (one replica is enough for reads and writes), QUORUM (the majority of replicas is required for reads and writes) and ALL (all replicas are required for reads and writes) [113, pp. 103-104]. Zhu et al. [230] even describe an extension to Cassandra, that allows to define a latency limit with the system optimizing consistency settings while making sure the latency limit is not exceeded.

Application of the CAP theorem to the reference architecture: When applying the CAP theorem or the PAC/ELC model to the reference architecture, it is necessary to consider the importance of consistency, latency and availability of a database for the different storage areas. One observation is, that analytical workloads are read-heavy, often with periodic updates. This is at least true for the analysis of data at rest, the upper part of the reference architecture diagram in Figure 4.3.

[15] An extensive discussion of different distribution and replication models would go beyond the scope here. A further discussion of different models and how they impact consistency, availability and latency can be found in Abadi [37],Fowler and Sadalage [113, pp. 37-45,57-59] or Silberschatz et al. [199, pp. 839-853]

Strong consistency guarantees are therefore not as important for the Analysis Data Stores because write/write conflicts do practically not occur in those regular batch jobs. With large batch-jobs to update the Analysis Data Stores, write latency is also less of an issue, write throughput is however more important, especially it the window for running the batch-job is rather small. Read latency is however highly important for Analysis Data Stores that serve interactive analysis tasks. The same goes for availability. Scalability is definitely an issue and that goes for both, being scalable with data volume and with read workload. Sharding (for scaling with data volume and intra-query parallelization) and replication (for availability and read workload balancing) are therefore both important.

Similar observations occur, when looking at the Enterprise Data Warehouse and the Raw Data Archive. While a consistent state is very important for the Enterprise Data Warehouse, this mostly influences logical consistency, that is that the database should enforce constraints in the data model. The Raw Data Archive does not need this level of logical consistency and constraint enforcement considering that it stores raw, un-cleaned and only partly integrated data. Write/write conflicts are again improbable due the periodic loading through batch-jobs. This is even more true if the Enterprise Data Warehouse is implemented around the ‘immutable data’ and ‘append only’ principle as described in Section 4.2.2. Read and write latency are both not that important, again because neither read or write operations are conducted interactively in a larger scale. Read and write throughput, on the other hand, is, especially if the loading time window is small. Availability is important.

The situation is different for analysis of streaming data. With data streaming in with a high frequency of rather small write operations, write/write and also read/write conflicts are a lot more likely. Additionally, a high frequency of rather small write operations puts more focus onto write latency compared to writes in large batch jobs, which merely require write throughput. This is also important, as streaming data should be available for analysis in the Streaming Staging Area and the Analysis Delta Stores as soon as possible. Read latency is also important for Analysis Delta Stores, but also for the Stream Staging Area if they can be queried interactively. Availability for reads and writes is important with read availability being even more important than write availability. The premise is, that analysis results should be accessible and acted on quickly. If the data is not available, this mainly results in a waiting time until network or node failures are resolved and can therefore be seen as a very high latency impact. Consistency, on the other hand, can be slightly relaxed. As described before, the idea is to make data accessible quickly with the premise to clean and integrate it less thoroughly compared to the processing of data at rest. Some (logical) inconsistencies can therefore occur anyway and are to be expected.

4.3.2 Relational Databases

The databases that are traditionally most widely used are those, that follow the relational model. Data is modelled using tables (relations) which be seen as representing a type of objects (e.g. customers). Each column in the table refers to attributes, that are applicable to objects of that type. Each row (tuple) refers to a distinct object if that type (e.g. a certain customer) and describes the relationship of attribute values of that object. Tuples of different tables can reference each other based on attribute relationships. The relational model allows to define constraints on both, relationships of attributes within one tuple and relationships between tuples of different tables. The database system enforces that these constraints are not violated. The data is accessed using a relational query language, almost always SQL. [199, pp. 39-55]

Relational databases therefore requires data to be modelled and parsed into the relational model while it is loaded into the database. This greatly reduces the flexibility of data that can be loaded and requires to schema of data to be known in advance. High record-to-record variability can either break the schema or leads to very scarce tables. On other hand it increases the query flexibility as data is generally modelled and not with certain queries or applications in mind. Furthermore, the need for parsing the data before loading it into the database and constraint checking lead to an increased loading time [161, 176], again in exchange for faster querying and application performance as data fields do not need to be parsed at application runtime. This also means, that errors in the data (e.g. attribute type or constraint violations) pop up when loading the data, while an application on top of the database can be sure, that the data is consistent to those constraints and violations do not pop up at application runtime. Additionally, the use of a high-level, declarative language like SQL provides several advantages. High-level languages normally require less lines of code to solve a problem, they are typically also easier to read and to learn by non-programmers. The later is especially important as many data analysts use SQL to interact with data stores, but have no extensive programming background. Additionally, query optimizer based on SQL have years of extensive research effort and are very hard and only with high effort to beat by programmers. Using ODBC or JDBC to interact with the database and run SQL queries can have a significant overhead, but this overhead can often be avoided by using stored procedures [208].

Massively Parallel Processing Databases: Lately, the term Massively Parallel Processing (MPP) databases came up to describe databases based on a distributed architecture to for high scalability. The idea and technology is however not new. Typically, these databases use a 'shared nothing' architecture, where each node has its own CPU(s), memory and disk. From a scalability perspective 'shared nothing' is superior to 'shared memory' or shared disk, as for the later the interconnection bus to either, the memory or the disk becomes a bottleneck hurting scalability [62, 99, 205, 208][199, pp. 781-784]

The data is then distributed over these nodes. Essentially, there are two forms of distribution. First, different tables are distributed across nodes. Second, tables are automatically partitioned (sharded) and the tuples of one table are distributed over nodes. Scalability is then provided due to parallelism [62, 208]. Inter-Query parallelism, that is different queries hitting different nodes, balances workloads and allows for scalability concerning the workload as well as for an increased query / transaction throughput. Intra-Query parallelism, that is one query touches data on different nodes with each touched node doing parts of the work, speeds up query runtimes [199, pp. 802-815]. Query parallelism depends, however, also on the partitioning technique. Hash-partitioning, where a hash function distributes the tuples based on the value of some attribute, is good for point queries on the partitioning attribute. Range partitioning, where data is distributed depending on some attribute value being within a certain range interval, is also good for smaller range queries. The risk of range queries is, however, that attribute values are concentrated within a certain range, which hurts workload balance between the nodes [199, pp. 798-802]. Additionally, data partitions are typically not only stored on a single node, but replicated over several. This is for availability and failure-handling reasons if one node goes down. Additionally it can provide even more inter-query parallelism as queries that read the same data can be run in parallel on different nodes [62]. For the user, however, partitioning and replication over multiple machines, is transparent [176].

Relational MPP databases typically support full ACID properties. This can hurt scalability especially for write-intensive workloads as distributed transactions and locking add overhead and are expensive. This is one of the reasons, why these applications can scale to the tens or hundreds of nodes, but might hit a limit at thousands of nodes [36]. An exception to some extent are data warehouse applications and analytical databases. These are very read-heavy with only periodical writes. This makes consistency and transactional conflicts less of an issue and allows for architectures that are even higher scalable. Several commercial databases, e.g. Teradata, Greenplum, Netezza or Vertica, do this successful and report that they are able to handle multi-petabyte databases [78, 161]. Note however, that the later are declarations of the respective databases vendors and should therefore be taken with a grain of salt.

Column-wise Storage: Another concept, mainly used in analytical databases, is the idea of column-wise storage. The idea is rather simple. While traditional, relational databases physically store data row-by-row (one complete tuple after another) column-wise databases physically store data column-by-column. Assume a,b,c refer to a row, while 1,2,3 refer to attributes. Simplified traditional databases system would store tuples using the structure (a1,a2,a3;b1,b2,b3;c1,c2,c3), while column-wise database systems would store (a1,b1,c1;a2,b2,c2;a3,b3,c3). Note however, that this only affects physically storage. Logically both implement the relational model. [38, 39, 206, 208]

Both storage schemes have advantages and disadvantages. First, column-wise storage achieves higher compression rates as most compression schemes are based on homogeneity within data. Typically, values within one attribute (that is within one column) are more homogeneous. Compression also has the side-effect, that more data can be stored in main memory with less copying between disk and memory and also better CPU cache utilization. Second, consider the star schema in data warehouse applications. Typically, fact tables are very broad with many different key figures stored as attributes. Queries however, often only touch few of these key figures, but aggregate them over many rows. These queries are faster on column-wise storage as they only need to read the attribute values of the included attributes, while they need to read the complete rows on row-oriented storage. Third, column-wise storage is more flexible in cases where the data model needs to be changes by adding columns to a table. [38, 39, 206, 208]

On the other hand, column-wise databases gain this advantage by trading-in write performance. In a row-oriented storage scheme a new tuple can just be added at the end. With column-wise storage a tuple needs to be broken down to attribute values and these cannot just be added at the end, but need to be inserted in several places. [38, 39, 206, 208]

Considering the reference architecture, the advantages for read performance make column-wise databases a very good fit for Analysis Data Stores. The disadvantage considering write performance is no problem here. Another story are Analysis Delta Stores. While they would definitely benefit from increased read-performance for analytical queries, they would also suffer from the decreased write-performance. Some examples of column-wise databases are C-Store, MonetDB, SybaseIQ, Vertica, Infobright, Paraccel.

In-memory: Another technology that gets attention lately is in-memory databases [158, 159]. They can be significantly faster than disk-based, not only because they hold all data in memory, but because they assume to do so. While a disk-oriented database would also cache all data in memory as long as memory is bigger than data volume, they still have to handle overhead to synchronize data in memory with data on disk. In-memory databases often get rid of memory buffer management, latching and logging. If they are properly architected this way, they can use main memory more efficiently compared to disk-oriented databases. Note however, that data is still flushed to disk from time to time due to availability reasons. Main memory is after volatile and restarting a machine would lead to data loss. [62, 208]

The obvious advantage of memory-based databases systems is performance, both for reads and writes. The disadvantage is cost. Even if proponents of in-memory databases claim, that main memory gets cheaper and cheaper and that it gets therefore feasible to hold all data in memory, disk is still and will probably always be more cost-efficient. This is even more true in cases, where a company's data volume grows faster than memory actually gets cheaper. In this case it is just infeasible to store all data in memory.

Considering the reference architecture storage areas that make most use of in-memory technology are those, that take most out of the performance improvement. These are all storage areas, that can be queried interactively by end-users and analytical applications. It can also be used by storage areas that are queried by deep analytics batch jobs as those jobs still benefit from eliminating I/O waiting times for loading data from disk. However, considering the batch nature it might make less of an impact compared to interactive analytics. On the other hand, it might enable to run applications that were only feasible in batch mode before in an interactive manner. Still, considering this Analysis Data Stores and especially Analysis Delta Stores make in-memory databases an ideal choice. The later even more, as they do not only benefit from increased read performance, but also from increased write performance due to high-frequency writes from streaming data. Both of them also only hold parts of the data, which makes it easier and less costly to fit all data into memory compared to the large volumes of more detailed data in Enterprise Data Warehouse and especially Raw Data Archive. This, again, is even more true for Analysis Delta Stores as they only hold data for a certain interval until the next ETL run for data-at-rest. Note also, that in-memory techniques are of course not limited to relational database systems. They are just listed here, as they are recently more and more applied to those, e.g. to the 'NewSQL' systems described below.

In-database Analytics: Another idea is to move computation and application tasks to the data, instead of exporting data from the database to an application and conducting application logic there. Again, this is not a concept solely applicable to relational database systems. The idea is also inherent in Hadoop's architecture as described below and in other systems. However it gets more and more applied to relational systems, shaping the term 'Extreme SQL'. Pavlo et al. [176] e.g. state, that almost every parallel processing task can be formulated as a set of SQL queries in combination with user defined functions. The idea is to automatically use parallelism of distributed databases and to profit from the databases system's query optimizer. Additionally, query outputs are typically smaller then all query imports. Processing application logic close to the data and only responding with the results therefore means less network traffic and decreased network latency.

Based on this idea several efforts were made to implement data mining, machine learning and statistical techniques either directly within a database or based on SQL [161]. One of the efforts to do this in SQL is the MADlib project. It aims at implementing statistical methods using SQL scripts and user defined functions [87, 131]. Considering the use case for the reference architecture, in-database analytics makes obviously sense to be included in Analysis Data Stores and Analysis Delta Stores. Another instantiation of this idea is the inclusion of the MapReduce programming model[16] also into relational databases. One example is Greenplum. Note however, that MapReduce system typically require some overhead to start a job and are therefore mainly suitable for batch-oriented and less for interactive tasks.

'NewSQL': So called 'NewSQL' databases combined several of the technologies described above. There developed was driven by the idea of 'one size does not fit all' and with the goal of defining special-purpose databases for write-heavy, OLTP-like workload with high-frequent, but rather small write operations. These efforts aim at building OLTP systems with high scalability, but without giving up full ACID compliance and transactions. They also strive to provide better per-node performance compared to some systems the focus merely on scalability [78, 129, 210]. However, these systems largely favour small-scope transaction over few notes. Transactions and joins over a large number of

[16] A broader discussion of the MapReduce paradigm is described below when discussing Hadoop

nodes are still possible, however these are less scalable and efficient and therefore suffer a performance penalty. In other words, these databases support full ACID compliance, while only suffering from according scalability impacts if transactions span multiple nodes [78]. Considering the characteristics of the workload with highly frequent, but rather small requests, this seems to be an appropriate join for specialized OLTP systems. Contrary to analytical systems, large, node-spanning joins are rather rare.

To achieve high node performance and scalability most of these systems are completely in-memory. They additionally get rid of logging, locking, latching shared data structures and buffer management. Harizopoulos et al. [129], Stonebraker and Cattell [208] claim, that this overhead accounts for almost 80% of typical transaction runtime. Logs need to be written to disk to be durable if a node fails. Instead 'NewSQL' system do node fail-over and recovery from data replications on other nodes. This is also necessary, as in-memory databases would just not allow to write logs to disk. The buffer pool is completely avoided by the database being solely in-memory. Locking is avoided by using techniques like the two-phase commit protocol for distributed, node-spanning transactions (which notably decreases scalability in those cases, as mentioned above), ordering through timestamps and multi-version concurrency control. Finally, the avoid latching by using improved B-trees for indexing and often being single-threaded per CPU core with main memory being divided into buckets, one for each CPU core. [78, 207, 208]

Considering the reference architecture, the optimization for OLTP and high-frequent, low volume transactions places 'NewSQL' databases as an option for the stream acquisition components. However, due to the performance penalty for distributed joins they might not be an ideal choice for Analysis Delta Stores, but the seem very suitable for the Stream Staging Area. Both, the support of ACID-like consistency and scalability and performance for high-frequent write operations, make them a good fit. However, data needs to conform to the relational schema. If the model of data items in the stream are not known in advance or highly variable, this might lead to problems.

Some examples of 'NewSQL' systems are VoltDB, MySQL Cluster, ScaleDB, Clustrix, ScaleBase, NimbusDB, Google Megastore, MemSQL, NuoDB, SolidDB, TimesTen [78].

4.3.3 'NoSQL' Databases

'NoSQL' databases receive some kind of hype lately. The term is, however, ill-defined with different databases labelled that way being very different considering their architecture. The biggest similarity is probably, that none of them is built on the relational model, though the incorporated data models can be rather different. A classification of these data models and a selection of databases confirming to that respective model can be found below. 'NoSQL' databases also do not support SQL, but often proprietary query languages, some declarative some not. This is notable as most problems they try to solve are not caused by SQL as a high-level language, but by the relational model itself or traditional database architecture. It is also notable, that some of them actually added high-level languages that are similar to SQL. While 'NoSQL' databases are often categorized and discussed according to the data model they support (key-value stores, document stores, column-family stores, graph databases[17] [70, 126][113, pp. 20-23,26-28][187, pp. 3-7]), there are still major differences between single databases within these categories. This refers to the architecture, replication and distribution models they use and therefore they trade-offs they make considering consistency, latency and availability [134]. These differences make it difficult to discuss 'NoSQL' databases in a general way. Take the following discussion therefore with a great of salt where many points are valid for a lot of these systems but not necessarily to all.

[17] For a discussion of the different data models see below

One goal ‘NoSQL’ databases are designed for, is to simplify application development on top of them and to improve programmer productivity. First, they aim at resolving the impedance mismatch, that results from difference between data structures within applications and the relational model. They avoid the translation between both by allowing to store application objects directly within the database, e.g. as binary large objects (BLOBs) or using nested data objects. Many ‘NoSQL’ databases e.g. use JSON documents for internal storage. JSON can be easily imported and serialized into objects or data structures in several programming language [113, pp. 5-6]. Additionally, they allow for a flexible schema. Data can be added to BLOBs or new attribute to JSON documents without the need to change the data model of the database [78, 126]. This is often described as ‘schemaless’ [70]. However, data is rarely completely schemaless. An implicit schema is typically inherent in the data and just hidden within a BLOB as it contains different attribute values, e.g. separated by commas. It is then on the application to assume a certain structure within the data and impose this schema at application runtime and to extract data out of a BLOB. [113, pp. 28-30] ‘Schema-at-read’ is therefore typically a better term to describe this concept.

Another goal of ‘NoSQL’ databases is to provide a high level of scalability [70][113, pp. 8-9]. They are often based on a ‘shared nothing’ architecture with automatic sharding of data, distribution of these shards across all nodes and automatically balancing shards across nodes with new records or nodes added to the system. Additionally, data is typically replicated for availability and workload balancing [78, 112].

As indicated above many of them also relax consistency guarantees to achieve this goal. Some of them differ in how much consistency they give up, but many implement eventual consistency as described above [44, 70, 126]. Very few of them support ACID compliance and transactions across objects[18]. Often, they neither allow distributed joins (or even no joins at all) or complex queries, but only provide a rather simple CRUD interface [78, 134]. Instead, many of them rely on data models, that Fowler and Sadalage [113, pp. 13-24] call ‘aggregate oriented’. The idea here is, that while only simple operations on objects are allowed, these objects can have a rather complex, often implicit structure internally. An object can include several nested sub-objects and attributes.

In this sense an object, or attribute, groups together sub-objects that are related and are likely to be accessed together. This is different to the relational model, where these different sub-objects would typically be separated over several tables with foreign key relationships to reference each other. An example is e.g. to group respective orders within a customer object. This makes it easy for applications, that need to access all orders of a particular customers, but it makes it difficult to e.g. query all orders, that include a certain product. Aggregate-oriented databases are therefore modelled with a focus on a certain application and give up query flexibility [134]. Aggregate oriented databases are therefore a poor fit, if data is based on a lot of relationships and operation across aggregates is necessary. On the other hand, as mentioned above, aggregates often can be easier important into respective applications as they can just be serialized as a data structure or object in many programming languages. Additionally, aggregates are often a good unit for sharding and replication with data, that is typically accessed to gather, being stored on the same node. This can improve scalability and latency as it avoids distributed joins (or joins at all), which are not possible anyway. As indicated above, while not offering transactions and ACID compliance across aggregates or objects, many ‘NoSQL databases’ allow ACID transactions within one aggregate. [113, pp. 13-24]

With this approach, consistency handling and joins or more complex queries are merely pushed to the application level. With that in mind, scalability gained should be taken with a grain of salt, as the scalability issues are just moved to the application layer, if those things are needed. One can think about this as a turning away from a classical 3-tier architecture with tasks of the databases tier

[18] One exception of a ‘NoSQL’ database, that does so, is RavenDB [134]

being moved to the application tier. The databases tier still conducts file operations and management, but higher level tasks (as mentioned processing more complex queries, transaction processing and sometimes secondary indexes to access non-key attributes) are moved to the application tier. Often, this make it a more difficult problem. As mentioned when discussing relational databases, it is hard for programmers and takes a lot of effort to equal or beat well-implemented query optimizers. While it can provide better scalability for the database tier, scalability in the application tier can suffer. Additionally, for large analytical queries the network between application and database can become a bottleneck as input data for those queries is typically larger than output data. Therefore, aggregation-oriented databases, if implemented this way, are more suitable for high-frequent, low-volume transactions with few inter-object relationships. In these cases they work well. [148]

One option, several of those databases offer, to work around the lack of joins, is the use of materialized views. These are often based on the MapReduce programming model integrated into the database[16], e.g. in MongoDB [70]. Joins and queries across aggregates can then be pre-calculated, stored and are therefore accessible by applications. While this helps to some extent, interactive querying is still prohibitive as materialized need to be defined in advance and their calculation is typically done in batch mode. The definition of materialized view is therefore still focussed on expected queries and query flexibility is still limited. [113, pp. 30-31,67-78]

There is however one big exception to the points discussed above, that should be noted. Graph databases are often mentioned as one category of ‘NoSQL’ databases, though they are fundamentally different to aggregate-oriented databases. First, they support a data model of small records and complex interactions. Relationships are supported as 1st-class citizens and graph databases are therefore ideal for applications heavily based on relationships between data items and traversing these relationships. While they also provide a flexible schema and loading of variable and complex data, they additionally provide high query flexibility across relationships. They often also provide stronger consistency guarantees compared to other ‘NoSQL’ databases, sometimes full ACID compliance, but their data model is very difficult to shard or partition. Therefore they suffer in scalability and most graph databases are actually deployed on a single server. A deeper discussion of graph databases and other ‘NoSQL’ data models can be found below. [113, pp. 25-26][187, p. 6][190, pp. 3-7,25-39]

Key-Value Stores: The most basic data model among ‘NoSQL’ systems is provided by the category of so called key-value stores. They are aggregated oriented, so the discussion above applies. Aggregates get access by a key and consist of a key-load (the value), that is completely opaque to the database system. For accessing data, typically a simple interface is provided mainly allowing create, read and delete operations. Updates are often not possible, as the value is opaque to the database. In those cases, old data is simply overwritten or both values are stored using a versioning scheme [78]. [187, p. 4] [113, pp. 20,81-88]

Internally the value can be almost everything, a BLOB, text, JSON, XML. As it is opaque to the system, it cannot enforce any constraints onto the data. While this provides a high schema flexibility and allows to easily store unstructured and semi-structured data, it also means, that it is completely in the responsibility of the application to parse the data and to interpret it. This also means, that data is only accessible by primary keys. There is no possibility to select data based on some other attribute. Again, this ensures read performance and scalability, but renders query flexibility mainly in-existent. In cases where data needs to be selected on some attribute within the aggregate, with key-value stores all data would need to be loaded to the application and selected there. This is clearly non-feasible and one big issue limiting key value applicable stores only in few situations. [187, p. 4] [113, pp. 20,81-88]

Considering the consistency, typically only operations on a single key can be considered atomic. Transactions across keys are not supported and most key value stores implement eventual consistency, however are often tunable (e.g. by using quorums to trade-off read against write performance). Due to the relaxation of consistency, most key value stores instead provide high availability for both, reads and writes. If write conflicts occur, most of them provide the option to either let the newest write (based on a timestamp) win or to store both key value pairs and then return both to the client application to be resolved there. Sharding and replication is done over the key, often using hash partitioning. Range queries can therefore also be tricky, focussing the use case onto single key accesses. [187, p. 4] [113, pp. 20,81-88]

Additionally, some key value stores are strictly in-memory with only periodically pushing snapshots to disks, e.g. Redis or Memcached. This provides another use case with those databases used as simple caching servers or even as a cache integrated within an application and running on the same machine. Considering the reference architecture, the later can be a valuable use case to cache data within analytical applications. Apart from that, key value stores are hardly applicable for any of the storage areas due to their opaqueness which limits data management capabilities. Due to their write performance and as they implement the append-only principle, they could be used as a staging area for unstructured and even semi-structured data. Another possibility is, to use key value stores as one part of the Raw Data Store. They can be used to store unstructured data there, especially text, and are combined with a more structured database which references to single key value pairs. Additionally, by indexing the value into the Search Index before it is stored, opaqueness can be worked around as the Search Index can be used to search through the data e.g. using a full text search. Some examples of key value stores are Redis [70, 78][187, pp. 261-306], Riak [78, 134][187, pp. 51-92], Tokyo Cabinet [78, 134], Voldemort [78, 134], DynamoDB [134] and Memcached [78].

Document-Oriented Databases: Similar to key value stores, document-oriented databases (also called document stores) are aggregate oriented, so again, the general discussion above applies. Aggregates (called doccuments) are also associated with a key. However, the difference is, that documents are not opaque to the database. They typically use a self-describing, hierarchical tree structure. Most of them use JSON, some use XML and some use a proprietary format, e.g. BSON (binary JSON) in the case of MongoDB, which is obviously a binary version of JSON optimized for storage space and scan speed. Documents are then grouped in collections. One can imagine a collection as a table in the relational model and a document as a row. There could e.g. be collections for customers or products with each single customer's or product's data being stored in a document. The main difference is, that documents have flexible and more complex structure. Flexible in the sense, that documents within a collection do not need to adhere to each other concerning their internal structure and attributes and the database system does not enforce a schema or constraints. This makes it easy to add attributes to a single object without changing a schema first. 'Empty' attributes can simply be left out, which saves storage spaces in situations of very sparse data. Complex means, that documents can include nested attributes, lists of attributes and even sub-documents. As described above in the discussion about aggregate orientation, a document holding customer data can e.g. hold several orders of this customer. This make document stores a good fit to hold semi-structured data, especially considering that data sources (e.g. web data sources, web services, APIs) often provide data directly as JSON documents. However, most document stores do not allow documents to include pointers to other documents, at least not in a way that the database directly uses them, e.g. for joins. [78][187, pp. 5-6,309-310][113, pp. 20,89-98]

As noted above, the internal structure of documents is not opaque to the database system. Besides basic CRUD operations, all of them therefore allow to query and select documents based on their attributes. They also allow for secondary indexes over document attributes to speed up those queries.

To formulate the queries, most of those query languages are proprietary, some of them use Javascript or a proprietary language based on Javascript. Additionally some, e.g. CouchDB, integrate MapReduce[16] to compute complex queries with document crossing joins into views, which can themselves be queried. The views can either be materialized in advance or completely computed at query runtime. Note however, that the later case can be a latency issue, especially as MapReduce computations are typically batch-oriented with some start-up overhead and low latency. With that, they provide more query flexibility than key value stores, but applications are dependent on the data being modelled according to their needs as joins across documents are mostly not possible. [78][187, pp. 5-6,309-310][113, pp. 20,89-98]

Considering consistency, most Document Stores fit into the general impression of 'NoSQL' databases as discussed above. The distribution models differ, almost all provide sharding, while often using a master-slave or quorum-based replication model. Typically they relax consistency to achieve high availability and better scalability, and mostly provide eventual consistency. But some are tunable, e.g. based on quorums. Transactions are atomic within one document, which makes consistency similar to query flexibility dependent to some extent on the data being modelled according to the application's needs. Documents provide a unit for sharding and provides good read and write scalability assuming the application needs fit how the data is modelled within documents. Read scalability and possibly write scalability (depending on the number of writeable replicas or on quorum settings) is additionally improved by replication, though there is the possibility that reads from replicas are stale due to the eventual consistency model. [78][187, pp. 5-6,309-310][113, pp. 20,89-98]

Considering the reference architecture, document make little sense for analysis oriented storage areas as query possibilities and performance largely depend on the model of the aggregates and are not flexible enough for ad-hoc queries. They can however be an option for staging semi-structured data and due to their possible write performance and scalability also for the Stream Staging Area, e.g. to cache semi-structured event stream or data from web logs. Keep however the consistency model in mind. Depending on how much a certain document store actually relaxes consistency write/write conflicts might occur, or data might get lost, e.g. if the master node crashes in a master-slave model before data is propagated to any slave. They are therefore ideal for streaming workloads that are append-only, which is typically the case for event logs. Additionally, they might be an option to store semi-structured data in the Raw Data Archive. Considering that some integrate the MapReduce programming model, some deep analytics tasks on the Raw Data Archive could then directly be implemented within the database. Some examples for document stores are MongoDB [78][187, pp. 135-176], CouchDB [78][187, pp. 177-218], RavenDB [134], SimpleDB [78] and Terrastore [78].

Column Family Stores: Column-family stores are databases modelled after Google's BigTable [80] and Amazon's Dynamo or DynamoDB. Discussing them in general is however a bit tricky. While in most other 'NoSQL' the data model between most of included systems is very similar or even identical with only the implementation as well as distribution and replication models differing, the data models of column-family stores can slight differ. This depends mainly on if they are implemented based on Google's BigTable paper or based on Amazon's Dynamo and Dynamo DB. Both of them share common concepts of rows, column families and columns, but differ in how they nest rows and column families.

The general concepts of both is, that a table is implement as a multi-dimensional map of rows and columns with columns that are related to each other or commonly accessed together grouped into column families. While the terminology sounds very similar to the relational model, the main difference is, that columns are not prescribed by a schema. Most of the time column-families need however to be registered with the database, but this does not prescribe which concrete columns are grouped into a column family. Columns can therefore vary between rows. Sparse data, that is attributes without a value, do therefore not infer storage cost. Some column family stores allow however to register common columns as metadata about a column family without enforcing them. This is e.g. necessary if secondary indexes need to be defined over these columns. [78][113, pp. 21-22,99-110][187, pp. 5,309]

Columns are comprised by a column key and a value and values are accessed using this column key together with the respective row key, similar to accessing a two-dimensional map. Values are typically uninterpreted strings or BLOBs, but some column family stores also support basic data types, e.g. integers or dates, and even lists or sets. Remember, that they are not necessarily registered with a schema to prescribe a data type. Additionally, in many cases column values within a row are timestamped. A column within a row can therefore hold several values with different timestamps. These are e.g. used to resolve write conflicts and identify stale data, but also to automatically expire data. If a row / column combination which has several timestamped values is accessed, the response typically just includes the latest value. [78][113, pp. 21-22,99-110][187, pp. 5,309]

Some column-family stores additionally know the concept of super columns. A super-column is then itself a map of columns. With this concepts columns can be nested, essentially increasing the dimensionality of the map. Most databases however only allow to add one level of nesting, that is super-columns can only contain normal columns, but not super-columns themselves. Additionally, columns within a super column cannot be accessed fine-granularly, but only a super column as a whole. [113, pp. 21-23,99-110]

The difference between the models mentioned above is, that in databases modelled after Google's BigTable (e.g. Apache HBase) column families are nested inside rows, typically in the way that they are merely used as prefixes to column names to indicate which columns are grouped together. A table has therefore several rows and A row has several columns, which are grouped into column families via name prefixes. Again, while column families need to be registered in a schema and each row of a table uses the same column families, columns within these column families differ from row to row. [187, pp. 5,93-105]

In databases modelled after Amazon's Dynamo and DynamoDB (e.g. Apache Cassandra) the nesting is the other way around with rows being nested inside column families. A column family therefore has several rows, which again have several columns. Data is always accessed against a certain column family with a row and a column key. In this sense column families are therefore comparable to tables in the relational model and most systems with this data model do not support an additional table structure around column families. Databases using this second column family model are however those, that typically support super columns as described above. These are less used in the first model. [113, pp. 21-22,99-110]

As different rows and column families can contain differing columns, both models provide schema flexibility and allow to store data with variable attributes. They, however, impose more structure onto the data than document stores and allow for less complexity of objects, as nesting is limited. As mentioned above, data can be accessed via a combination of row and column key. It is also possible not to specify a column key, but to retrieve an entire column family. Additionally, it is often possible to define secondary indexes over columns, so data can be retrieved by querying column values. Note however, that secondary indexes are typically not allowed for columns within a super column. To formulate queries, most column family stores implement some proprietary query language, e.g. CQL in the case of Apache Cassandra. Query flexibility is however limited, because column family stores normally do not allow joins and the discussion about aggregate oriented databases applies, that they need to be modelled based on application needs. It would e.g. be possible to model customers as rows with a column family for orders and each order being modelled as a column within this column family. The data about that order is then simply stored as a BLOB value of these columns. This makes sense if an application commonly accesses orders grouped by customers. If an application typically accesses single orders, it would make more sense to model them as rows. [113, pp. 21-22,99-110][187, pp. 5,309]

Most column family stores provide good scalability and read/write performance by using partitioning across both dimensions, rows and column family. Rows are typically sharded on the primary using range-partitioning with rows themselves being partitioned into column families and sharded across nodes. The unit of distribution is therefore the combination of row and column family. This makes sense considering that column families group columns that are typically used together. From a modelling viewpoint it can additionally make sense to use a naming scheme for row keys so that related rows are within the same range and therefore stored on the same nodes. Otherwise, rows are just alphabetically ordered over the keys. Additionally most of them use replication and typically relax consistency to eventual consistency to gain scalability and availability if facing network partitions. This can often be tuned, e.g. in Cassandra with consistency settings ONE (standard mode, read from one replica even if data is stale, write to one replica), QUORUM (majority of replicas need to respond to reads and writes), ALL (all nodes need to respond to reads and writes). Transactions are often atomic on the row level, but not across rows. Additionally, many column family stores buffer writes in commit logs in memory and only periodically flush them to disk. This can lead to data loss if a node goes down, before updates have been written to disk, especially if writes can be submitted to a single node. On the other hand, this largely increases write performance. [78][113, pp. 21-22,99-110][187, pp. 5,309]

Considering the reference architecture, it makes little sense to use column family stores for analysis oriented storage areas. They do not allow the necessary query flexibility as the data modelling needs to be optimized for expected queries and they do not provide joins for more complex queries. The are often reported to be a good fit for read and write heavy transactional systems over transaction with somehow variable and flexible data model or sparse data. Examples are event logging, content management systems or some other web applications [113, pp. 107-108], Facebook e.g. uses HBase for its messaging solution. None of these use cases really apply to the reference architecture. They could be used for the Stream Staging Areas to stage event streams and maybe even for storing event data in the Raw Data Archive as their data model fits event data well. Note however, that it is possible that data can be lost if a nodes fails before data updates are flushed to disk. Therefore they should not be used to stage highly relevant and important data. Some column family stores are e.g. Apache Cassandra [9, 78][113, pp. 99-110], Apache HBase (which is also part of the Hadoop Ecosystem) [14, 78][187, pp. 93-133] and Hypertable, which is somehow an exception in the way that it chooses stronger consistency over Availability for Partition Tolerance [70, 78]

Graph Stores: The last category of ‘NoSQL’ databases is very different to the categories and data models described above. This again shows, how ill-defined the term ‘NoSQL’ actually is. Whereas key value stores, document databases and column family stores rely on an aggregate-oriented data model, graph stores work on small records (nodes) with complex interactions and relationships between these records (edges). Put differently the data model is all about nodes connected to each other by edges, all together forming a graph. Nodes represent real world entities and edges represent relationships between them.

Based on nodes and edges there are however, several graph models that differ slightly. The one most widely used in graph stores is called property graph. In this model, nodes and edges are both first-class citizens. Entities (nodes) can have multiple properties (e.g. a name, the age of a person), where a property can be seen as a key-value pair. Relationships (edges) have one start and one end node, a type / label (e.g. is_friends_with) and can also have multiple properties (e.g. since).

Additionally relationships (edges) are directional and the direction has significance (e.g. if Paul loves Mary, that does not necessarily imply that Mary loves Paul). [113, pp. 25-26,111-122][187, p. 6][190, p. 5-7,32-39,48-51]

Other graph data models are the hypergraph, which additionally allows edges to have multiple start and end notes, and RDF, which originated in the Semantic Web community, is a W3C standard and often the underlying data model of so-called triple stores. In RDF data is modelled in triples of a subject, a predicate and an object. Subject and object can be seen as nodes (start and end node) which are connected by the predicate as an edge. Nodes and edges are identified by URI references, which can also be used to link to RDF data from other sources[19]. One difference between RDF and the property graph model is, that properties in RDF are modelled as literal objects with a predicate connecting the subject to the object. In other words, properties are nodes themselves, with a node connecting to entity node to the property node. The label of the edge identifies the property, while the literal node holds the property value. In the property graph, properties are simply attached to the nodes and not nodes themselves. Furthermore, in the property graph edges can have properties themselves, while they cannot in RDF. [5][190, 48-51,77-79]

Additionally, the physical storage model of a graph can differ using e.g. linked lists or indexes. Describing the concrete physical storage pattern would however take the much space here. I therefore refer to Robinson et al. [190, pp. 77-87] for a deeper discussion. A general remark is, that different storage architectures makes it easier for triple stores to shard the data, distribute it across machines and therefore improve horizontal scalability. The storage architecture in graph stores like Neo4J typically allows no automatic sharding at the databases level. If sharding is necessary for scalability reasons, effort needs to be made within the application to partition the graph based on domain knowledge and to manually distribute the partitions to separate databases on different machines. Horizontal scalability is therefore largely limited. This is however a trade-off and as a benefit traversal in those graph stores is typically faster than in triple stores providing lower query latency. [113, p. 119][190, pp. 78-79]

Additionally, the graph in a graph store can still be replicated over several machines to improve availability and scalability with read workload by adding more read slaves. Replication is also done for availability and fail-over. Most graph stores use master/slave replication, where the master node is necessary to respond to writes. Write request can be sent to slaves, but are conducted in a synchronous transaction with the master node, where the respective slave node only commits after the master node committed. The writes are then propagated to other slaves. This way write/write conflicts are avoided and write consistency is increased. Reads can be stale though, if they are made against a slave and an according update has not been propagated, yet. If the master fails, typically an elected hand-off to one of the slaves is conducted. Additionally, most graph databases support ACID transactions within the database node. Keep in mind, that distributed transactions are not necessary as graph databases do not support sharding. [113, pp. 114-115,119]

Schema flexibility is generally high. New nodes or new edges to current nodes can be easily added at runtime. Schema definition is typically not necessary. It is however difficult to update several entities at once. Considering query flexibility, graph stores are very well suited for problems that are heavily based on relationship and require traversing a graph, that is starting at one node and following its edges. This is very fast in graph stores, much faster than in relational database systems where these kind of queries require a lot of joins. It is possible to create indexes over nodes, edges and properties and it is also possible to select single entities or relationships based on their properties or to select sub-graphs based on graph patterns. Another use case is path finding. Analytical queries that involve any form of aggregation over properties of several entities or relationships are however typically not

[19] See the Semantic Web and the Linked Open Data project [2]

supported. To do these things, there are several graph processing language available. Some graph stores even support several and provide the choice to the user. Neo4J e.g. supports Cypher, which is a declarative graph traversal language, and Gremlin, which is more low-level and imperative. Triple stores based on RDF support SPARQL as a query language, which is also a W3C standard and the official RDF query language [6]. [113, pp. 115-119][190, pp. 55-59,91-92,179-205]

Considering the reference architecture, a good fit for graph stores are Analysis Data Stores and Analysis Delta Stores that are used for application which can be formulated as a graph or path finding, routing or dispatching problem and requires traversing of complex relationships, e.g. distribution and routing optimization in logistics [113, pp. 120-121][190, pp. 142, 164-178]. It can also be used to store a network of complex relationship or another link-rich domain (e.g. social connections, product preferences or eligibility rules) within the Raw Data Archive. Additionally, if one data source is RDF data from the Linked Open Data project or generally the semantic web, RDF triple stores are the natural choice to store the data in the Raw Data Archive. Another use case would be to use graph traversal for recommendations, e.g. in the sense of 'your friends also liked' (traversing first the 'friend' and then the 'like' relationship) [113, pp. 120-121][190, pp. 141,145-156]. In this case, the graph traversal would however be best suited as an Analysis component within the operational application that makes the recommendation to the end user. Apart from RDF triple stores, the most widely used graph store is Neo4J [187, pp. 219-260].

4.3.4 Apache Hadoop

Apache Hadoop Core

Apache Hadoop is a governing body of several open-source software projects, which are, as the name suggests, all organized under the roof of the Apache Software Foundation [12, 15, 226]. The initial project was created by Doug Cutting with the goal to create a distributed computing framework and programming model to provide for easier development of distributed applications. the philosophy is to provide for scale-ou scalability over large clusters of rather cheap, commodity hardware. Its creation was motivated and is largely based on papers published by Google to describe some of their internally systems, namely the Google File System [119] and Google MapReduce [93]. Accordingly, Hadoop's core consists of the Hadoop Distributed File System (HDFS) [198] and Hadoop MapReduce (MR)[20]. Note, that this section can only provide an overview about these components and a discussion according the match with the reference architecture. For a deeper description of Hadoop's architecture and internals, I refer to White [226] and Hadoop's documentation [12].

Typically, both of them, HDFS and MapReduce, are deployed together in a cluster. Input data for MapReduce needs to be stored in an HDFS instance on the same cluster and outputs are written back there. As HDFS's datanodes and MapReduce's worker nodes, called Tasktrackers, are running on the same physical nodes, MapReduce's job management (JobTracker) component can take locality of data into account. That means, it tries to schedule tasks to Tasktrackers that run on the same physical node as the HDFS datanode that holds the necessary input data. Computation is pushed to the data, not the other way round. If this is not possible, e.g. because all TaskTrackers local to necessary input data are already busy with another job, data needs to be sent from the datanode that holds the necessary data to the datanode local to the TaskTracker that runs the particular task. As this requires loading data over the network, it obviously slows down job execution considerably. [226, p. 28]

[20] Note, that this discussion emphasizes on Hadoop 1.x. There is another thread of development versioned as Hadoop 2.x. This lately (August, 25th) evolved from alpha into beta version. The stable release is, however, still considered to be Hadoop 1.2.1. I will list some of the changes and architectural differences after the discussion of Hadoop 1.x.

Note however, that HDFS and MapReduce do not necessarily need to be deployed together. HDFS can generally be used as a distributed file system without MapReduce on top. On the other hand, Hadoop can integrate several distributed file systems. HDFS is just one of them, KFS is e.g. another one. MapReduce can work on top of every of these file systems, though using another one can lead to performance decreases. Not all of the usable file systems allow for data locality optimization and HDFS is in general designed and optimized to work together with Hadoop MapReduce. [226, pp. 47-49]

Hadoop Common: Hadoop Common contains a set of components, libraries and interfaces, that mainly support other Hadoop sub-projects [12]. As these components do not have any functional implications, I will skip a deeper discussion.

Hadoop Distributed File System: The Hadoop Distributed File System is, as the name obviously suggests, a distributed file systems modelled after the Google File System [119]. As mentioned above it is the primary distributed data storage component used by other Hadoop applications, but can also serve as a stand-alone file system. HDFS makes distribution to users completely transparent. To end-users it provides the view of a traditional file system over a hierarchy of directories and files. Only internally does it map and distribute files to different nodes in the cluster and translates directory and file operations to operations distributed over several nodes. [12, 198][226, pp. 41-73]

In general, HDFS is optimized to handle very large files and a write-once read-many workload. It can have performance issues when operating on a large number of rather small files. HDFS files are broken into chunks of 64 megabyte each and these chunks are distributed and replicated over nodes. The replication factor is typically set to '3', but can be freely configured to adjust for the required level of availability. HDFS knows two typyes of nodes based on a master-slave distribution model. Typically there exists one master node, the so-called Namenode[21] and several slave-nodes, so-called Datanodes. [12, 198][226, pp. 41-73]

The Namenode holds all metadata about the file system and stored files. It manages the directory tree, maintains the mapping of data blocks to Datanodes and it manages all namespace operations (e.g. renaming files) as well as file access by clients. Datanodes store the data blocks, serve client requests and provide access to data. It also executes operations on the data block, which get triggered by the Namenode, e.g. deletion. If applications need to access data in HDFS, they use the HDFS client, a library that transparently provides them with an interface for file operations. The HDFS client manages the data access. It first requests metadata about a certain file from the Namenode. This metadata includes information on which nodes blocks of the specific file are stored and the client can use this information to directly request these blocks, which are then streamed into the client application. If a client wants to write data, it also needs to first send a request to the Namenode, which then provides him with node locations to write to. Using these scheme, HDFS achieves a decoupling of data and metadata, which allows for better load balancing and avoids, that workload spikes in data access does not interfere with metadata operations. [12, 198][226, pp. 41-73]

Considering this single Namenode design, HDFS can get limited as data volumes grow. With metadata being stored in a single node's memory, there is a limit of how many metadata objects can be stored, which is only possible to scale up by adding more memory to the Namenode server. Additionally, this is a potential performance bottleneck if metadata operations' workload gets to large and overwhelms the single Namenode server, and it is also an availability issue considering the Namenode as single-point-of

[21] This is at least true for Hadoop 1.x. Hadoop 2.x adds Namenode replication to HDFS. It also adds the possibility of federated file systems consisting of several HDFS sub-systems, each with its own Namenode [16]

failure. The use of a so-called Backupnode, which acts as a journal store for the Namenode and holds an additional, up-to-data image of the file system, can be used to soften the availability issue as the Backupnode can be used for fail-over in case of a failing namenode. Nevertheless, special maintenance effort aimed at the Namenode server is required. However, these issues are recognized and solved with Hadoop 2.x, which allows for a multi-Namenode design. [12, 16, 198][226, pp. 41-73]

Regarding data storage capacity and data access workload, HDFS is designed for linear scalability. If more nodes are added to the cluster, these can function as Datanodes, increase storage capacity and balance data read workload. Data blocks are replicated over several Datanodes, so HDFS is also fault-tolerant and has built-in recovery capabilities. If a Datanode fails, data blocks it holds are typically available on another Datanode (assuming that not all replicas fail at once), the system can continue to serve requests and the respective data block can be replicated to another Datanode to achieve the configured replication factor again. Considering read and write performance, HDFS is optimized for batch processing, favouring overall throughput over individual operations' latency. One reason for that is e.g. the large block size and the fact, that data blocks need to be read as a whole. HDFS also does not allow random writes or updates, but only appends. [12, 198][226, pp. 41-73]

In general, HDFS provides for low cost storage, that is good for sequentially reading whole data blocks, so for tasks that require sequential data access or conduct operations over a large range of sequential data points. It is not good for searching a single point of information in the data. After all HDFS is just a file system and no database. It does not provide any structure over the data and does not support indexing, at least not on its own. Considering the reference architecture, this makes HDFS a good fit for the Raw Data Archive given its ability to store large volumes of unstructured data with a low cost per terrabyte ratio. It is definitely not suitable, to use it for data stores that serve for interactive queries and analysis doe to its batch nature. Additionally, HDFS gives little support according to data management. It simply provides access to a collection of files. It is in the responsibility of client applications and users to ensure consistency of those files, do data maintenance and ensure compatibility of client applications as data evolves over time [98, 161].

Hadoop MapReduce: Hadoop MapReduce is a programming framework modelled after Google's MapReduce paper [93] and typically deployed over a HDFS instance. It aims to simplify development of distributed applications and data processing. It does this, by providing developers with a programming model to define and orchestrate distributed processing steps. The programming model defines two basic functions, map and reduce, which need to be implemented by the programmer. MapReduce then manages breaking jobs down into different map and reduce tasks and the distribution of each of these tasks over nodes in the cluster and parts of the input data. [12, 133][226, pp. 18-39,153-174]

During the map phase, the system breaks input data from HDFS into independent chunks of key value pairs and distributes them to the map tasks implemented by a programmer, each map task getting one chunk of the data. These tasks get processed in parallel over the independent data chunks with the results again being represented as key value pairs. The system then sorts the output of the map tasks based on their key and submits them to reduce tasks, which are again distributed over several nodes. The distribution to the reduce tasks works that way, that all key value pairs (from the map tasks' output set) with an identical ket get submitted to the same reduce task. The reduce tasks, again implemented by a programmer, combines and merges the different key value pairs into one final output set. [12, 133][226, pp. 18-39,153-174]

Considering the distribution model, MapReduce uses a master-slave architecture. The master, a so-called JobTracker, runs on a single node, while slave, so-called TaskTrackers, run on all remaining nodes. A MapReduce library manages the communication between the client application and the difeerent nodes. Before the MapReduce client can submit a job to the JobTracker, it needs to store all necessary input data into the underlying file system (most typically HDFS) and compute input splits for the job. Input splits identify partitions of the input data, which are submitted to a single map task. A JobTracker then sets up data structures required to manage the state of the job, retrieves the input split from the file system and then breaks the submitted job down into several tasks, one map task for each input split, and a number of reduce tasks that is defined in the job configuration. Finally, it sends the tasks to TaskTrackers, which are most local to the required input data stored in HDFS. The TaskTrackers retrieve the input data from HDFS and run the map tasks over it. Afterwards the respective TaskTrackers shuffles the output of the map tasks to the already set-up reduce tasks. Once the reduce tasks are finished, results are written back to HDFS and the respective TaskTrackers send a message to the JobTracker, which finally reports the job successful to the client application. [12, 133][226, pp. 18-39,153-174]

Hadoop MapReduce and Hadoop in general has a number of advantages and disadvantages. First, Hadoop is typically rather cheap and fast in deployment and maintenance [211]. The software itself is open source and free and Hadoop runs on commodity servers. This enables Hadoop with a good cost per terrabyte ration for storing data and processing. Additionally, the use of commodity hardware, automatic data and task distribution and respective optimizations provide Hadoop with good scalability. Data locality for MapReduce tasks reduces overhead of large data transports through the network, which can deem the network a bottleneck and affect total throughput. Hardware can be added or removed to adjust for storage needs.

Considering performance, Hadoop MapReduce works well for trivially parallelizable data processing tasks, that is, tasks where the input data can be easily split into chunks which can then be processed in parallel. It is less suitable for iterative workloads and for workloads, where single output records involve computation over all or large parts of the input data or where parallel tasks need to communicate with each other. It is possible to chain several map-reduce jobs together into a processing pipeline to simulate communication [113, pp. 72-77]and algorithms can be adjusted or replaced by others that better fit the MapReduce model [84, 153, 154]. However, neither of these is ideal. One problem with chaining map-reduce stages together is, that data is automatically persisted after each reduce task and needs to be accessed in HDFS as input for the next map-reduce stage. This creates I/O overhead and can slow down the processing. It is also not possible to keep an overarching state or buffer data in memory between chained tasks. On the other hand, loading data into Hadoop and storing it in HDFS is very fast as it does not require any translation into a certain schema or any data parsing. Data is just stored in its native format [176].

One limitation of Hadoop and MapReduce is its batch nature. It is simply not designed for highly interactive analytical workload. The biggest reason is, that MapReduce has an inherent contention and start-up time to create and distribute tasks and possibly data over the different nodes. Another reason is the lack of indexes to support selective queries and the need to always read entire data blocks and sequential ranges of data. This renders it infeasible for any latency-dependent tasks, interactive analysis or near real-time analysis of streaming data. [112, 170, 176]

Considering the reference architecture, Hadoop is a good fit for Deep Analytics components, due to their match in being batch-oriented. To help with the implementation, there are several libraries of machine learning and data mining algorithms implemented in Hadoop, e.g. Apache Mahout [8]. Apache MapReduce is also a good fit for Information Extraction as well as Data Cleaning, Data Integration and Data Transformation components, or put differently for ETL processes in general [155, 211]. These typically do not read in data selectively, but sequentially read in and process whole data chunks. Additionally, they typically do not create output results from a large range of input data points. Of course, using Hadoop MapReduce for such workloads works especially well, where data is already stored in HDFS, so e.g. for processing data stored in the Raw Data Archive. This also emphasizes the idea, of combining Raw Data Archive and Staging Area, where all extracted data is directly loaded into the Raw Data Archive, cleaned, transformed and integrated there using MapReduce with relevant data further loaded into the Enterprise Data Warehouse, while the entire data is kept in the Raw Data Archive.

Apache Hadoop Ecosystem

The ecosystem of sub-projects related to Hadoop or usable in its context is large and growing. As organizations tackle some of the data management issues and extend Hadoop a variety of sub-projects get created and open-sourced. This makes the ecosystem rather complex and makes it hard to keep an overview. In the following, I list some of these MapReduce extensions and give an indication for what they can be used. Note, that this is only an overview, considering the scope of this thesis, and by no means a deep technical discussion of the internals of those tools. For this kind of discussion I refer to literature and papers that describe certain technologies in-depth. Additionally White [226, pp. 12-13,301-403] gives an overview of some of the sub-projects and describes some the most important ones, namely Pig, HBase and Zookeeper.

Structure on top of HDFS and integration of database functionality :
HBase applies a column-family data model[22] on top of Hadoop and HDFS, so it is essentially a non-relational database that runs on top of HDFS. This allows data access based on a more fine-granular structure, adding access to specific data items based on their key and of certain attributes and columns within a data set instead of a mere sequential scanning of data. It also allows users to update, insert and delete data items. Additionally, it adds transactional capabilities to Hadoop. Considering the reference architecture it can be used on top of HDFS within a Raw Data Archive to add more fine-granular access and data management capabilities. It also provides good read and write performance and can be used as a Stream Staging Area. [14, 56]

HadoopDB is a project that combines relational databases with Hadoop. It places database instances, namely PostgreSQL,on the nodes of a Hadoop cluster next to TaskTracker and Datanode. These database instances can be used to store structured data within a Hadoop cluster, provide indexing and therefore increase performance of selective queries. HadoopDB further provides a Database Connector for MapReduce to access data in those database instances, a translator to transform SQL queries into MapReduce jobs and back which can also be used to query Hadoop using SQL and a catalog to store metainformation about the database instances. [36, 40]

Hive adds a virtual structure and table schemas on top of HDFS and maps this structure to data on the Datanodes of HDFS. It can then be queried using a query language similar to SQL, called HiveQL, and translates the queries into MapReduce jobs. While Hive is optimized for scalability, due to the translation of queries and execution as MapReduce jobs it still suffers the respective overhead and similar latency issues. It therefore mainly provides comfort to formulate MapReduce jobs in a higher-level language and typically simplifies the development of these jobs. Additionally, the metadata Hive creates about the table schemas can be shared with other components and be used for data exploration. Considering the reference architecture Hive can be used for every storage area, where HDFS is used for, to increase programmer productivity for MapReduce jobs and to provide metadata for data exploration. [13, 219, 220]

[22] See Section 4.3.3 for a description of the column-family model

Metadata in Hadoop :

HCatalog is part of the Hive project and responsible for providing the metadata layer and maintaining the tables schemas within Hive. It also allows to share metadata and interoperate across Hadoop tools, e.g. with Pig, but also with applications outside of Hadoop through a Representational State Transfer (REST) interface, e.g. with a general Metadata Management component for the entire 'big data' system. [17]

Pipelining MapReduce Jobs :

Pig is another abstraction layer on top of MapReduce. It provides a platform and execution framework for complex data flows for ETL processing and data analysis. Pig internally generates MapReduce jobs and chains them together to execute the data flow. To formulate the data flows, Pig uses its own language, Pig Latin, and provides primitives for common operations. Similar to Hive it allows to develop MapReduce jobs and workflows in a higher-level language and increases programmer productivity. For the reference architecture it can be used wherever Hadoop MapReduce is used especially for ETL-type use cases, to create the necessary workflows. [117, 170]

Programming models and high-level languages on top of Hadop MapReduce :

Pig Latin is the high-level data flow language on top of Pig. It abstracts away from low level MapReduce Java implementation. It provides common operations, e.g. groupings, joins and filters, and can be extended with user defined functions written in Jave. A Pig Latin script defines input data sets and operations to run on this input data sets and the output sources to write results to. These operations are connected with each other in a graph. Pig automatically finds the optimal data flow through this graph. [117, 175]

HiveQL is the high-level, declarative query language on top of Hive. Programming HiveQL is very similar to SQL.

Jaql is another declarative language on top of Hadoop that translates into MapReduce jobs. It provides a flexible, sime-structured data model based on JSON and is e.g. used in IBM's InfoSphere BigInsights product, which is also based on Hadoop. [65]

Additionally, there are several other programming models that can be used with Hadoop and translate into MapReduce jobs. **Cascading** is a high-level Java API that abstracts from MapReduce complexities using a more intuitive pipes and data flow models. **Scalding** is a Scala API, which can additionally be placed on top of Cascading and provides pre-implemented, common operations similar to the data flow abstractions in Pig. Finally, **Cascalog** is a Clojure API on top of Cascading, adding logic programming concepts similar to Datalog.

Querying data in HDFS without MapReduce :

Drill allows to scan through smaller data sets very quickly. It is inspired by Google's publication about their internally used query engine Dremel [167]. Apache Drill scans data in parallels and uses different algorithms to fasten scans doing filtering operations, aggregation etc. in parallel. It is still a full-data scan tool, but does not impose the task management overhead, that MapReduce causes. It is especially fast for tasks as data aggregation, sorting and top-x measurements and its use case are interactive and analytical queries. Considering the reference architecture it again can be used, where HDFS is used to accelerate some of these operations, especially for the Deep Analytics components. Apache Drill is still in the incubator phase [18].

Hadoop Management and Administration :
Oozie is a workflow management and scheduling system to coordinate jobs written in different languages and originating from different tools, e.g. Pig, Hive and native MapReduce. It allows linking of jobs and specifying order, triggers and dependencies between them, but also scheduling of jobs based on the existence of certain data in HDFS. [19]

Note, that this is only a selection and there are still several other projects related to Hadoop. Sqoop e.g. connects HDFS with relational databases to load data from one into another. There are also several projects, that try to integrate the statistical platform and language R with Hadoop, e.g. Ricardo [89] and RHadoop. Apache Mahout provides a library of machine learning and data mining algorithms implemented with Hadoop MapReduce [8].

Additionally, software for stream acquisition is important for the reference architecture. There are also several Apache projects to support that. Flume [20] is a system for distributed log collection from many sources. Flume consists of several agents deployed across an IT environment to collect logging data and send it to HDFS. Chukwa [21, 180] is a similar system but focussed on acquiring data more for periodic real-time analysis (within minutes), while Flume emphasizes on continuous real-time analysis (within seconds) and on enabling batch analysis on the acquired data. Apache Kafka is a distributed publish-subscribe system that can be used to publish in-streaming data and forward it to applications subscribed to a certain data stream [22, 149].

4.3.5 Comparing Storage Options

In general, there are three classes of storage options to choose from for implementing the reference architecture. These are distributed, relational databases, 'NoSQL' databases and finally HDFS. Relational databases mostly use SQL for data access, processing and analytics. 'NoSQL' databases use some proprietary API for data access and possibly MapReduce for data processing. Still, processing and analytical tasks typically and more so than in relational database systems require the data to be accessed and transferred into an analytical application and processing to be conducted here. Finally, processing of data stored in HDFS mostly involves Apache MapReduce.

Unfortunately, there are no overarching benchmark results available in literature, that would allow a close performance comparison of these systems. While there are benchmarks reports available, these are scattered, just involving a small collection of products and often not comparable as they use different hardware configurations and different algorithms and measurements to test the performance. Therefore, some of the next discussion should be taken with a grain of salt.

However, several authors publish benchmark results, which seem to agree that generally distributed, relational systems still have a slight performance advantage. This is especially true for complex analytical queries, that involve joins and aggregation or very selective data access. Some reasons of relational databases outperforming Hadoop is their support of indexing, Hadoop's overhead for starting and managing tasks especially with smaller data sets and the better usage of buffering data in memory. Additionally, Hadoop needs to parse data at run time to extract attribute values, while relational databases already do this at load time. Some of these point also indicate, that relational database systems are especially in favour considering smaller data sets that largely fit in memory and in selective queries, while Hadoop catches up as data sets get bigger and sequential scans are the norm, so in typical batch workloads [226, pp. 4-6]. Together with their query flexibility, this makes relational databases a good fit for Analysis Data Stores, which typically only contain smaller, aggregated parts of the data. On the other hand, as Hadoop does not need to parse data at load time, check constraints and fit it into a pre-defined structure, write performance of HDFS is better than in relational databases. This also means, that HDFS can store unstructured and semi-structured data, which does not directly fit into relational database management systems. [36, 98, 112, 176, 211]

Comparing relational and 'NoSQL' databases, somehow similar argumentation applies. Again, enforcing a structure on load time hurts schema flexibility and write performance of relational databases, but this is largely a trade-off to achieve query flexibility and query performance. While 'NoSQL' databases are as fast or even faster for simple, data access via a key or a range of keys, relational databases have a performance advantage for complex, analytical queries, that require joins and aggregation. For most 'NoSQL' databases, this requires to ship the input data sets completely to the application to join them there. Typically, however, the output set of a query is smaller than its input set, especially where aggregation and query conditions are involved. The higher communication volume therefore leads to a performance disadvantage for 'NoSQL' databases. Additionally, the query optimizer of distributed, relational databases typically considers the network traffic and data locality, essentially favouring local joins where possible, e.g. by replicating small tables over all nodes. Again, this decreases communication overhead and it applies, that it is very difficult for application programmers to implement a certain join or query as efficient as a highly developed query optimizer does. This point also accounts for Hadoop.

Mapping this to the reference architecture, as mentioned above, relational databases are a good fit for the Enterprise Data Warehouse, as they impose structure onto the data, enforce data constraints and therefore help managing the data and keep it clean. They are also a good fit for Analysis Data Stores due to their good query performance and flexibility. However, their schema is inflexible and they cannot store large volumes of unstructured data. This is, where HDFS can shine and makes it a good candidate for the Raw Data Archive also considering, that the Raw Data Archive is expected to have a larger data volume than the Enterprise Data Warehouse, and the scalability and low cost per terrabyte ratio of HDFS. It also works great as a Staging Layer with MapReduce jobs conducting ETL tasks like Information Extraction, Data Integration and Data Cleaning. 'NoSQL' databases on the other hand, typically have a very good write and read performance on keys. This makes them a good choice for capturing streaming data, e.g. implementing the Stream Staging Area, also considering, that streaming data is often semi-structured, which 'NoSQL' systems are well-prepared to handle.

5 Verification of the Reference Architecture

After developing the reference architecture, it is important to verify if it is relevant for practice and if it fits concrete architectures. There exist several frameworks for software architecture verification & evaluation, the most known being the Scenario-Based Architecture Analysis Method (SAAM), the Architecture Level Modifiability Analysis (ALMA), the Performance Assessment of Software Architecture (PASA) and the Architecture Trade-off Analysis Method (ATAM) [54, 105]. However, all of them are designed for evaluating concrete and not reference architectures, and the level of abstraction of the later makes it hard to apply them in this context. One reason is, that most of them are heavily based on stakeholder interviews and scenario analysis. According to Angelov et al. [50] due to their abstractness reference architectures do not have a clearly defined group of stakeholders and make it difficult to generate a concrete set of scenarios. To verify a reference architecture, Galster and Avgeriou [114] therefore suggest to either apply the reference architecture in concrete projects and conduct case studies, to conduct reference implementations and prototyping or to derive the validation from concrete architectures by mapping them onto the reference architecture, with the later method especially useful for classical reference architectures based on concrete systems.

Due to the scope of this master thesis and broadness of the proposed reference architecture, conducting an extensive reference implementation is not feasible. Additionally, I do not have access to a concrete project in the space neither direct contact to practitioners for interviews. However, as described in Section 2.2.3 the reference architecture is a classical one and needs to reflect knowledge from existing systems and be based on concepts proven in practice. Therefore, I decided to follow the third path and validate the reference architecture by mapping it to concrete ones.

In this chapter I will do this mapping of concrete 'big data' architectures from industry against the proposed reference architecture to evaluate if there is a match with proven concepts and to give some indication if the reference architecture is complete, that is if it can be used to describe a variety of systems in the space. I base the verification on concrete architecture descriptions by Facebook [221] and LinkedIn [216]. Additionally, I will check if the architectural patterns, presented in an Oracle white paper [217], can be described with the reference architecture. This verification mainly tackles the implementation-oriented view of the reference architecture. The functional view takes its justification from the set of requirements specified in Section 3.2. I acknowledge, that this is a rather small selection and biased towards end user facing web companies. To the best of my knowledge, there are no publications of concrete 'big data' architectures for companies or enterprises in other industries available. I consider this a serious limitation and propose that future work is necessary to make the verification more thorough. This can involve mapping concrete 'big data' architecture from companies in other industries, presenting the reference architecture to practitioners and interview them how they judge its applicability as well as doing case studies and applying the reference architecture in concrete project contexts.

5.1 Data Warehousing and Analytics Infrastructure at Facebook

Facebook's architecture for data warehousing and analytics is described by Thusoo et al. [221]. The main software products they use within the architecture are Scribe for log collection, HDFS for storage and Hadoop MapReduce together with Hive for analytics, both batch and ad-hoc. The architecture is depicted in Figure 5.1. The description is from 2010, so it certainly evolved and does not look the same any more at this point in time. However, facebook is one of the most data-intensive companies in the world, the described architecture served a 15PB data warehouse with a daily increase in data volume of about 60TB. From this point of view, data management problems facebook faced 3 years ago might still be similar to the problems other companies face today.

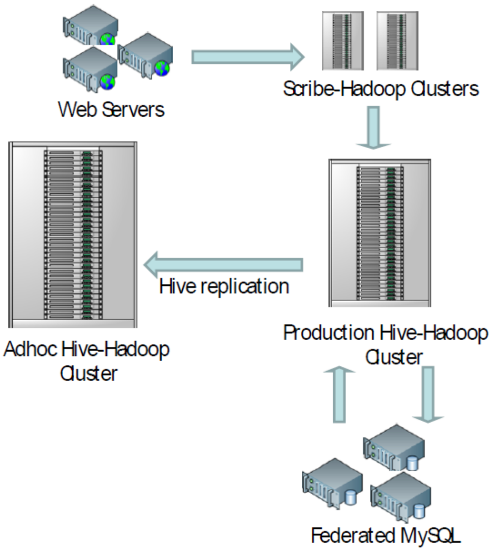


Figure 5.1: Concrete 'Big Data' Architecture: Data Flow Architecture at Facebook [221]

The analytical system in this case has two major data sources. The first is a **structured data source**, a tier of federated MySQL instances, that contain all data about facebook's website and the data that is operationally served to its users. The data gets loaded daily. The second data source are the web servers that generate log data about how the web-site is used. This data is semi-structured and constantly streaming in as the web-site is used, making this a **streaming data source**.

The data from the federated MySQL tier is scraped daily and loaded to the Hive-Hadoop clusters. This scraping can mapped to the **Extraction** component. The log data from the web servers streams into a cluster of Hadoop-Scribe servers. Scribe aggregates the in-flowing data and stores the aggregated logs in HDFS from where they are periodically compressed and loaded to the Production Hadoop-Hive Cluster. Mapping this to the reference architecture, the aggregation in Scribe refers to the **Stream Acquisition** component, whereas HDFS on the same cluster servers as **Stream Staging Area**.

The log data in the Hive-Hadoop clusters gets published either hourly or daily. With this delay it does not qualify as a Analysis Delta Store. The but best maps to the Raw Data Archive or the Enterprise Data Warehouse in the Reference Architecture. The mapping of Production Hive-Hadoop cluster and Adhoc Hive-Hadoop cluster is not completely clear. However, as far as the paper describes it, there seems to be no further data cleansing and logical data integration. Therefore I tend to map both, Production Hive-Hadoop cluster and Adhoc Hive-Hadoop cluster, to the **Raw Data Archive** as the combination of the HDFS instances in both clusters. Note however, that the Production Hive-Hadoop cluster. Essentially, the Raw Data Archive is divided into two storage sub-areas, which are physically separated, but kept synchronized due to the replication process. Additionally, they are used to serve different analysis tasks.

The Production Hive-Hadoop cluster executes highly-important batch jobs with strict deadlines, while the Adhoc Hive-Hadoop cluster executes lower-priority batch jobs. Both of them qualify as **Deep Analytics** tasks which are executed over data in the Raw Data Archive and written back to the same. They are also written back to federated MySQL tier, which is a backflow to the data sources. Additionally, users can conduct ad-hoc analysis using HiPal or Hive CLI over tables defined in Hive and mapped to data in HDFS. HiPal and Hive CLI can therefore be mapped as Free Ad-hoc Analysis applications, while Hive (or the logical structure Hive imposes over HDFS) can be mapped as an **Analysis Data Store**.

Considering this discussion, Figure 5.2 shows how the concrete architecture from facebook can be mapped and expressed using the reference architecture. While it was possible to map it to the reference architecture and it was generally a good fit, there are however some things that cannot be easily mapped or expressed with the reference architecture or some observations that should be noted. First, the division of the Raw Data Archive (or any other storage area) into several physically distinct parts to serve different classes of jobs is not presentable with the components in the reference architecture. Though, this is no issue affecting correctness and utility of the reference architecture. The different components are not defined to be atomic and can themselves contain sub-components. This is mainly a consequence of the abstractness of reference architectures in general. It is however important to note, that this is actually the case. Second, the reference architecture does not include a back-flow to general data sources, just a back-flow from the Stream Analysis component. The data is available through the Data Access API and be accessed by external applications, but this is a pull rather than a push mechanism. Third, Analysis Data Stores and Raw Data Archive or Enterprise Data Warehouse do not necessarily need to be physically distinct. Here, the difference is just a logical table structure Hive layers over low-level HDFS data. In other situations, free ad-hoc analysis might even be possible directly over the data in the Raw Data Archive.

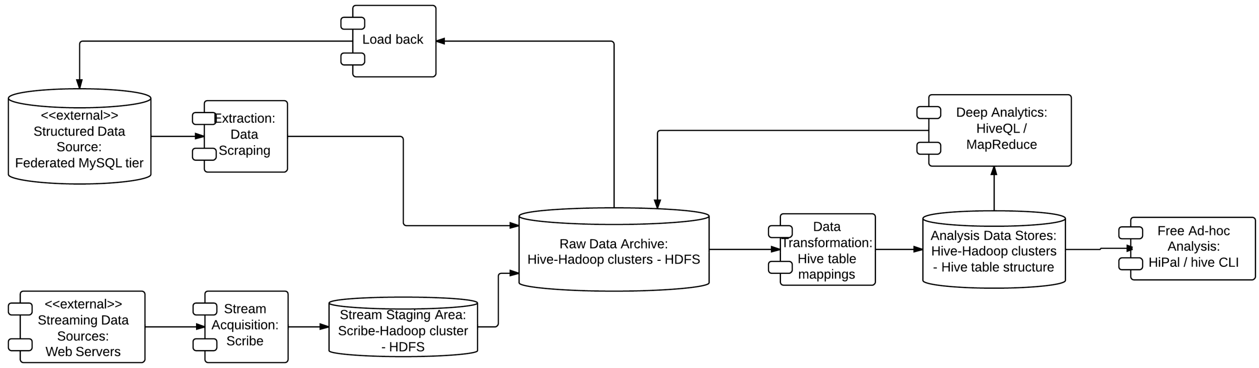

Load back
<<external>> Structured Data Source: Federated MySQL tier
Extraction: Data Scraping
Deep Analytics: HiveQL / MapReduce
Raw Data Archive: Hive-Hadoop clusters - HDFS
Data Transformation: Hive table mappings
Analysis Data Stores: Hive-Hadoop clusters - Hive table structure
Free Ad-hoc Analysis: HiPal / hive CLI
<<external>> Streaming Data Sources: Web Servers
Stream Acquisition: Scribe
Stream Staging Area: Scribe-Hadoop cluster - HDFS

Figure 5.2: Mapping to the Reference Architecture: Data Flow Architecture at Facebook

5.2 The ‘Big Data’ Ecosystem at LinkedIn

LinkedIn’s ‘big data’ ecosystem is described by Sumbaly et al. [216]. The ecosystem ingests both, streaming activity data and data at rest from relational databases. It ensures that the data adheres to some structure guidelines, runs analytical batch jobs over the data and makes it accessible in several ways. First, it streams data back into operational systems. Second, it loads data into a databases for application and users access. Third, it creates data cubes and allows for OLAP operations over the data. The technology it uses is Kafka for stream processing and for streaming results back, HDFS for storing the data and Pig and Hadoop MapReduce for batch analytics, Azkaban to define workflows, Voldemort as database to provide results back to applications and finally Avatara for cube construction and access. Azkaban is a workflow scheduling tool developed by LinkedIn to manage, trigger, execute and monitor workflows with several jobs formulated either in Pig, Hive or natively in Hadoop MapReduce.

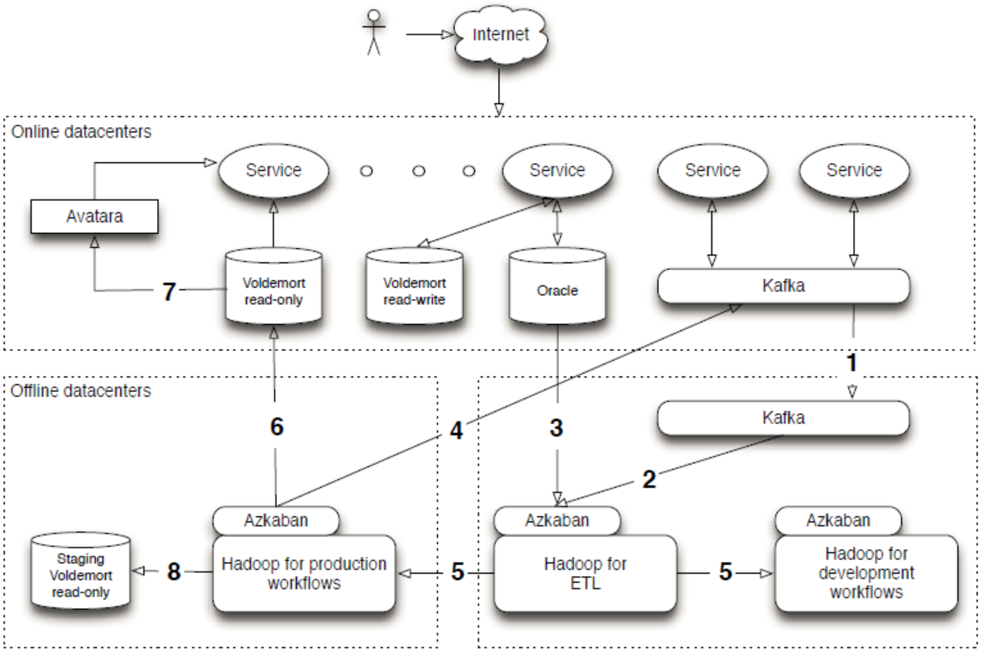


Figure 5.3: Concrete ‘Big Data’ Architecture: Big Data Ecosystem at LinkedIn [216]

LinkedIn has generally two types of data sources. First, there is activity data that is constantly flowing from online applications as **Streaming Data Sources**. Second, there is data from operational, relational databases as **Structured Data Sources**. The later is directly loaded into a ‘Hadoop for ETL’ cluster and stored in HDFS. Here the data runs through an ETL workflow and is then distributed to other storage areas. HDFS on the ‘Hadoop for ETL’ cluster can therefore be mapped to a **Staging Area**.

The activity data is acquired from the online services and propagated into the ‘big data’ system using Kafka. Kafka is an Apache project and essentially a publish-subscribe service. Online services publish the activity data in Kafka, where they are grouped into topics, which other applications can subscribe to. LinkedIn uses two separated Kafka clusters. The primary cluster is used to receive activity data, check its schema against a schema registry, filter out data whose schema does not fit the definitions and publish it back to other operational sources that might need it. A mirror process keeps the secondary cluster in synch, which publishes the data to the ‘big data’ ecosystem. Mapping this to the reference architecture, the primary Kafka cluster resembles the **Stream Acquisition** component including a filtering function. The secondary Kafka cluster holds the data and provides it to the ‘big data’ system, therefore resembling the **Stream Staging Area**.

Activity data from this second Kafka server is then pulled every 10 minutes by and loaded into the ‘Hadoop for ETL’ cluster mentioned above. An Azkaban /MapReduce workflow on this cluster checks for new data and pulls it into its HDFS directory. This part of the workflow can be seen as a **Load** component. Once the data is loaded the workflow runs an aggregator job to combine and deduplicate data and another job to conduct retention policies. This part of the workflow can be seen as a combined **Data Cleaning** and **Data Integration** component.

Once this workflow has run and the data is available in the ETL HDFS instance, it is replicated to two more instances, ‘Hadoop for development workflows’ and ‘Hadoop for production workflows’. The Hadoop development instance is used, for workflow developers to deploy new workflows, test them and get them reviewed. Only afterwards can they be deployed at the Hadoop production instance. The HDFS instance on cluster therefore clearly qualifies as a **Sandbox** with the newly tested workflows being preliminary **Deep Analtyics** components over the data.

In the ‘Hadoop for production workflows’ cluster HDFS holds the data to be served for analytical workflows (again using Azkaban to manage Pig, Hive and native MapReduce jobs), once they are tested and reviewed, to derive value from it, e.g. in the form of some predictive application. Considering the ETL processing and schema matching done before, one can map this HDFS instance to the E**nterprise Data Warehouse** of the reference architecture with the analytical workflows representing **Deep Analytics** components. The results are then written back into the HDFS instance as a derived data-set, but also directly pushed to several other systems.

First, analysis results can be streamed back to online applications by publishing it in the first Kafka cluster, from where it can be accessed by applications, if they subscribe the respective topic, and steamed into the application e.g. in form of a news feed. To some extent this is a backflow of data to the data source, which resembles the observation from discussing facebook’s data warehouse architecture above, that this is a missing connection in the reference architecture. It also does not fit the backflow of an analytical model into a Stream Analysis component, as it aims more at providing the data for presentation in an operational application. On the other hand, it is not only a backflow to the data source, but it is made available to all applications subscribed to the respective topic in Kafka. Comparing the intention, it fits the **Data Access API** in the reference architecture.The reference architecture, however, does not include an intermediary (Kafka in this case) between data access and Enterprise Data Warehouse, but the API directly access the later. Additionally, Kafka as used in LinkedIn’s infrastructure combines propagation of data to operational systems with the extraction of data from the operational systems into the ‘big data’ system. It therefore fulfils more of an two-way communication function.

Additionally, there are two other access and usage scenarios for data out of the ‘big data’ ecosystem. For these, data is loaded into multiple Project Voldemort instances. Project Voldemort is a key value store initially developed and open-sourced by LinkedIn and in this case extended with optimization for bulk-loading read-only data [215]. Some of these Voldemort instances can be directly accessed by operational applications to read data and analysis results. Other Voldemort instances are used to serve Avatara, an OLAP system developed by LinkedIn for scalable cube construction and materialization, for which it internally leverages Hadoop, as well as query serving [229]. Dashboarding applications can then query the Avatara cubes to present, to enable end-users to navigate through, and to visualize the data. Avatara and its serving Voldemort instances can therefore clearly be mapped to **Analysis Data Stores** with the dashboard applications on top as **Guided Ad-hoc Analysis** components. The directly accessible Voldemort instances could also be identified as **Analysis Data Stores** with Voldemort’s internal API as **Data Access API** to provide data for all kind of operational systems.

It is, however, arguable if these Voldemort instances and the data within fit the critera of being analysis or application optimized. It could be argued that they do due to Voldemort being extended and highly optimized for read access.

Additionally, there is a Staging Voldemort instance in the LinkedIn ecosystem, which is loaded from the 'Hadoop for production workflows' cluster. This is used for debugging and to understand results and outputs of workflows. This is a component, that is completely missing in the reference architecture.

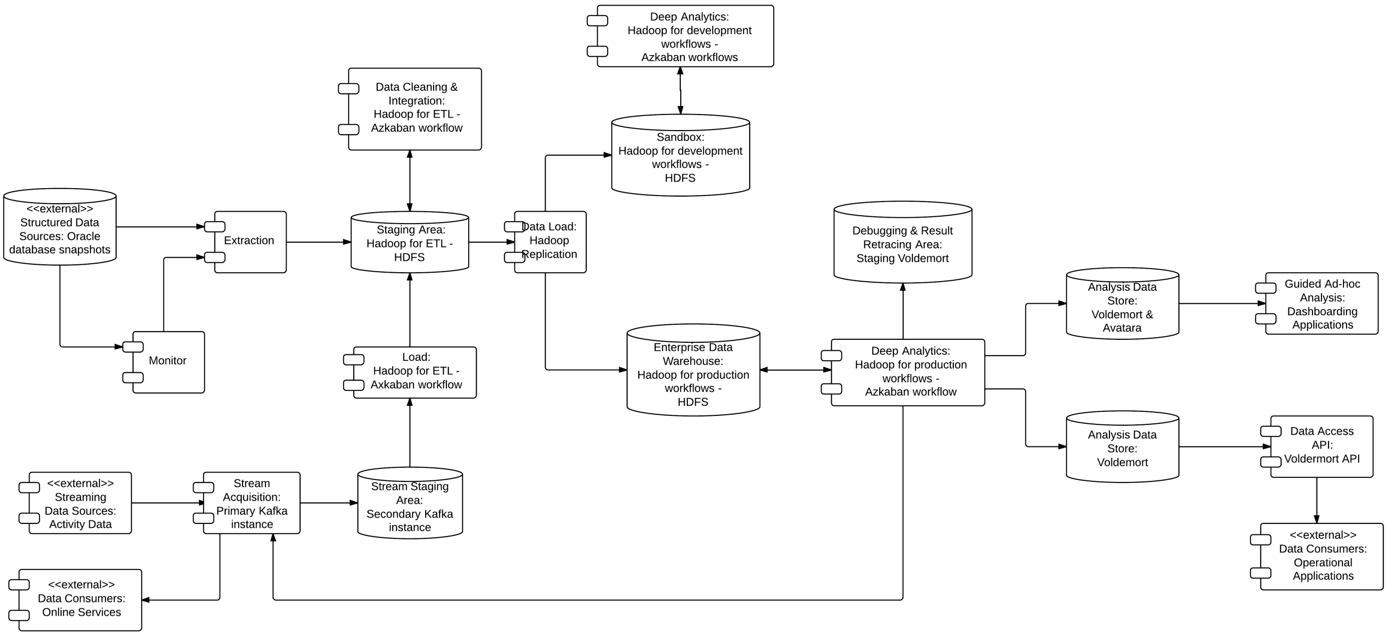


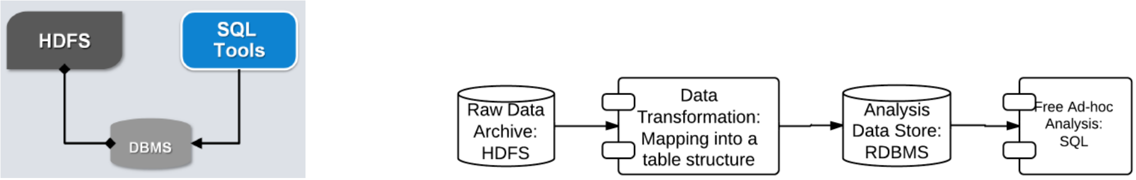
Figure 5.4: Mapping to the Reference Architecture: Big Data Ecosystem at LinkedIn

Again, it was largely possible to map concrete architecture and reference architecture. Figure 5.4 shows, how LinkedIn's system can be modelled using the reference architecture. However, as with facebook's data warehousing and analytics infrastructure, there were some components that did not completely fit. First, as with facebook's architecture, a backflow of data to the sources cannot easily be mapped, as the reference architecture only considers a immediate backflow of Stream Analysis results. The Data Access API does not completely make up for this as it is defined to enable pull but no push data delivery. Second, the reference architecture does not include a serving area where data is temporarily stored to be accessed by operational applications. This shows as the mapping of the Voldemort instance for data serving to an Analysis Data Store is no complete fit as there is no transformation to optimize data structure for data analysis and additionally as the data is not necessarily used for further analysis, but just to be included and presented in operational applications. The third point is connected to this as the data streaming into the primary Kafka instance, intermediate storage there and provision of the data back to online services (giving it the function of a two-way communication layer) cannot directly be modelled using the initial reference architecture. Fourth, the Staging Voldemort instance fulfils a debugging and output retracing function that is not considered in any component of the reference architecture even if it servers the same intention as requirement VAR4.3, that is to make results understandable and traceable.

5.3 Oracle ‘Big Data’ Architecture Patterns

In an Oracle whitepaper Sun and Heller [217] describe several patterns for ‘big data’ processing, which should be included in and be representable by the reference architecture.

Figure 5.5a shows **Pattern #1 - Initial Data Exploration**. The idea is to define a virtual table structure with a mapping to the underlying HDFS structure and to mount this pattern into a relational database management system. First, the reference architecture is more abstract, so it does not prescribe concrete technology choices like HDFS. However, the pattern can be applied to map data from the **Raw Data Archive**, which can be implemented using HDFS, to an **Analysis Data Store**, which can then be accessed using a **Free Ad-hoc Analysis** component, which SQL actually is. The mapping of potentially semi-structured raw data in HDFS into relational tables is essentially a virtual **Data Transformation** task. Figure 5.5b shows the modelling of the pattern using the reference architecture.

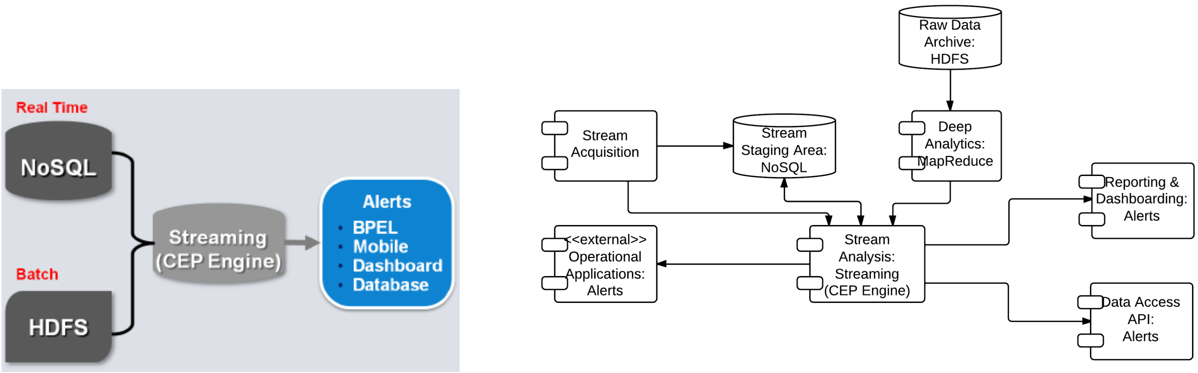

HDFS
SQL Tools
DBMS
Raw Data Archive: HDFS
Data Transformation: Mapping into a table structure
Analysis Data Store: RDBMS
Free Ad-hoc Analysis: SQL

(a) Big Data Pattern #1 [217]

(b) Mapping to the Reference Architecture

Figure 5.5: Mapping to the Reference Architecture: Big Data Pattern #1 - Mount HDFS into a RDBMS

Figure 5.6a shows **Pattern #2 - Big Data for Complex Event Processing**. Here, the concept is to acquire streaming data from different applications to do stream analysis within a complex event processing engine. Additionally, streaming data is captured into a write optimized ‘NoSQL’ database. Now data streaming into the complex event processing (CEP) engine can be joined with streaming data that was flowing in before during a certain time window and got captured in the ‘NoSQL’ database, but also with data from different data streams. Additionally, Hadoop MapReduce is used to pre-calculate models (risk profiles in this example) from detailed data stored in an HDFS cluster and to send them to the complex processing engine.


Real Time
NoSQL
Batch
HDFS
Streaming (CEP Engine)
Alerts
BPEL
Mobile
Dashboard
Database
Raw Data Archive: HDFS
Stream Acquisition
Stream Staging Area: NoSQL
Deep Analytics: MapReduce
Reporting & Dashboarding: Alerts
<<external>> Operational Applications: Alerts
Stream Analysis: Streaming (CEP Engine)
Data Access API: Alerts

(a) Big Data Pattern #2 [217]

(b) Mapping to the Reference Architecture

Figure 5.6: Mapping to the Reference Architecture: Big Data Pattern #2 - Complex Event Processing

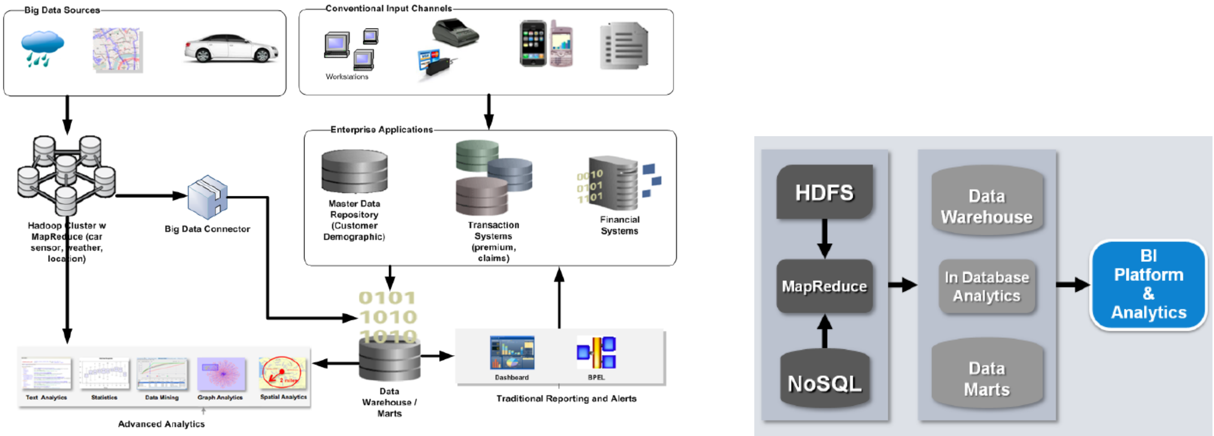
Again, the pattern is more concrete than the reference architecture concerning technology choices, but it is also less concrete concerning the function of the different components. Mapping the ‘NoSQL’ instance to a **Stream Staging Area** component and the CEP engine to a **Stream Analysis** component is rather straightforward. Mapping the HDFS instance is more difficult as the whitepaper does not give much information about the characteristics of the data in there. This generally leaves room for mapping it to either a Raw Data Archive or an Enterprise Data Warehouse. Considering the technology, however, using a plain distributed file system with a (relational) databases management system on top makes a mapping to a **Raw Data Archive** more likely.

Additionally, mapping the pattern to the reference architecture implies several components, that are not in pattern diagram in Figure 5.6a, but described in the whitepaper. First, data from HDFS is not just integrated into the CEP engine, but Hadoop MapReduce is used to create profiles, which are sent and can be used in the later. This clearly maps to a **Deep Analytics** component. Second, the diagram shows no equivalent for stream acquisition, but the text implies, that different data streams are streamed into the CEP engine and into the 'NoSQL' database. This is the function of the **Stream Acquisition** component, which is therefore added to the mapping. Third, the 'Alerts' component in the diagram actually includes several systems, which need to be mapped to different reference architecture components. The BPEL engine refers to a business process management system and can therefore be mapped to an external **Operational Application**. The Dashboard straightforwardly maps to a **Reporting & Dashboarding** component. The Mobile interface is a bit more tricky as the text mentions that it is part of the dashboarding solution to push alerts to mobile devices. It could therefore be just included into the Reporting & Dashboarding component, which then externally pushes data to the mobile device. It could, however, also be modelled via a **Data Access API**, which is used by the phone to pull this information. The decision of how to model this in the reference architecture can therefore not directly be derived from the pattern description, but depends on the concrete implementation. Finally, the 'Alerts' component of the pattern diagram contains a database and the description implies that the output of the CEP engine is pushed there to enable further processing and analysis. This can be modelled as a backflow into the Stream Staging Area, from which this data is loaded into the Raw Data Archive and then integrated into the processing pipeline.

Pattern #3 - Big Data for Combined Analytics - is more complex and extensive than the other two. It is about combining structured with unstructured (and possible streaming data) by using a Hadoop cluster together with an Enterprise Data Warehouse. In the whitepaper the pattern is shown in two different diagrams, first a data flow diagram (see Figure 5.7a) and second, a conceptual diagram (see Figure 5.7b). However, the later is rather ambiguous as the interplay between Data Warehouse, In Database Analytics and Data Marts in the second box is not implicitly shown. Therefore, I will focus the mapping onto the data flow diagram. Additionally, both diagrams of the whitepaper are to some extent inconsistent as the 'NoSQL' in the conceptual diagram does not show up in the data flow diagram and there is no explanation in the text. Therefore this requires some speculation.

Considering the data flow diagram there are some components that can be mapped in a rather straightforward manner. First, there is a Hadoop cluster for sensor, weather and location data. This Hadoop cluster is fed with data from 'Big Data Source'. The text identifies this data as sensor data that needs to be stored in flexible structures. Therefore I assume them to be mainly semi-structured. Considering the reference architecture, in conclusion HDFS on the Hadoop cluster can be mapped to a **Raw Data Archive** with data extraction from **Semi-Structured Data Sources**. This resembles the HDFS component in the conceptual diagram.

Additionally, data is extracted from Enterprise Applications into a Data Warehouse. Neither the text nor the data flow diagram give any indication of the Enterprise Data Warehouse including an explicit Staging Area, however, considering the general best practices in Data Warehouse architecture and the need to integrate data from several sources, I assume this as a given. Both, text and data flow diagram, also mention, that the Data Warehouse component includes Data Marts. With the reference architecture being more fine-grained in this context, the Data Warehouse component described in the



(a) Big Data Pattern #3a: Data Flow [217]

(b) Big Data Pattern #3b: Conceptual [217]

Figure 5.7: Big Data Pattern #3: Big Data for Combined Analytics

pattern can be modelled with a **Staging Area**, an **Enterprise Data Warehouse** and **Analysis Data Stores**, with the later resembling the Data Marts. The data is extracted and loaded into the Staging Area from transactional, master and reference data, that is from **Structured Data Sources**. This refers to Data Warehouse and Data Marts in the conceptual diagram of the pattern.

The data marts then provide the data to two kinds of analytical applications, the first for Traditional Reporting and Alerts and the second for Advanced Analytics. The second type can be directly mapped to **Free Ad-hoc Analysis** components. The first, however, includes a dashboard application and a BPEL engine, according to the data flow diagram, as well as a backflow to the Enterprise Applications. While the first can again easily mapped to a **Reporting & Dashboarding** component, the BPEL engine and the backflow is odd. The reason is, that a BPEL engine indicates a reaction to data analysis within some business processes. Even considering the Big Data Sources and even the Enterprise Applications produce streaming data, there can hardly be an on-time response if the data is first going through the Hadoop cluster and / or the Enterprise Datawarehouse. The function of the BPEL engine is not described in the text, while it is mentioned, that end results are integrated with the Enterprise Applications. From this point of view, the best guess to model this is using a **Data Access API** for these applications to access the data.

Furthermore, the text states, that MapReduce is used to identify patterns and trending insights from the sensor data loaded into the Hadoop cluster and that these are then integrated with the structured data in the Data Warehouse. The diagram shows, that a component called Big Data Connector is used to transform the results, that are written back to the Hadoop cluster, into a structured form and load them into the Data Warehouse. With this in mind, these MapReduce jobs (which are also shown in the conceptual diagram) can be mapped to **Deep Analytics** components, while the Big Data Connector resembles an **Information Extraction** component to impose structure and load the data into the Enterprise Data Warehouse.

The remaining parts are less clear to map. As Data Warehouse and Data Marts are modelled as one component, it is not clear, if all the sub-components within Traditional Reporting and Alerts and within Advanced Analytics build on top of data marts or if there are applications that directly access the Data Warehouse. Considering typical data warehouse architecture, I assume the former. Additionally, it is not clear, how exactly the Hadoop cluster interacts with the Advance Analytics applications. The edge in the data flow diagram indicates, that it does, while the conceptual diagram would imply, that it does not. Therefore, I make the assumption, that the Hadoop cluster includes

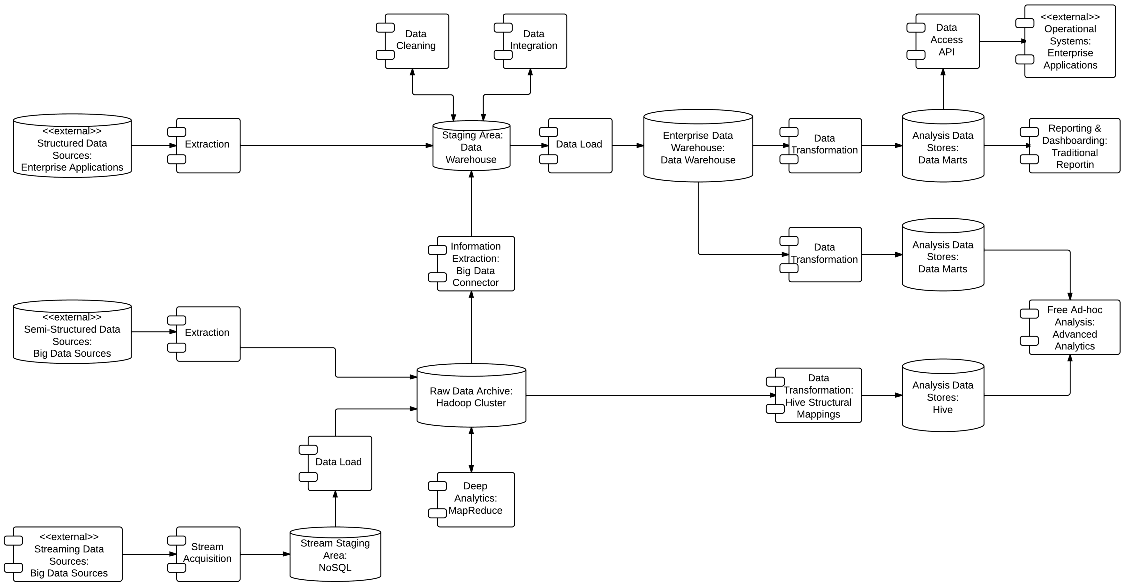

Data Cleaning
Data Integration
Data Access API
<<external>> Operational Systems: Enterprise Applications
<<external>> Structured Data Sources: Enterprise Applications
Extraction
Staging Area: Data Warehouse
Data Load
Enterprise Data Warehouse: Data Warehouse
Data Transformation
Analysis Data Stores: Data Marts
Reporting & Dashboarding: Traditional Reportin
Information Extraction: Big Data Connector
Data Transformation
Analysis Data Stores: Data Marts
<<external>> Semi-Structured Data Sources: Big Data Sources
Extraction
Free Ad-hoc Analysis: Advanced Analytics
Raw Data Archive: Hadoop Cluster
Data Transformation: Hive Structural Mappings
Analysis Data Stores: Hive
Data Load
Deep Analytics: MapReduce
<<external>> Streaming Data Sources: Big Data Sources
Stream Acquisition
Stream Staging Area: NoSQL

Figure 5.8: Mapping to the Reference Architecture: Big Data Pattern #3 - Combined Analytics

some more structural mapping layer on top of HDFS, e.g. Hive, that can then be accessed and queried by the Advanced Analytics. This can be modelled with a virtual **Analysis Data Store**.

Another component shown in the conceptual diagram is a NoSQL database, which is mentioned in the text to capture low-latency data with flexible structure. This database is not included at all in the data flow diagram. Considering the description, I assume, it is used to acquire and store streaming data from the sensors mentioned in the Big Data Sources. I therefore model it as a **Stream Staging Area**, whose data is then integrated with the Hadoop cluster. The complete modelling of the pattern using the reference architecture is shown in Figure 5.8.

Considering the mapping of all three patterns, I come to the conclusion, that in general all of them can be described using the reference architecture. Put differently, one can conclude, that all the patterns are present. Pattern #3 was more difficult to map, but mainly because it is ambiguously and not completely described. There is, however, one shortcoming. The backflow into Enterprise Applications in this pattern is mapped using a Data Access API component. Assuming that the backflow consists largely of alerts, this is probably a push mechanism, while the Data Access API is defined as a pull mechanism where applications access different data store in the 'big data' system. This observation is congruent to the mapping of facebook's and linkedin's architectures, where the backflow of data was also an issue.

Additionally, the whitepaper lists several best practices. Most of them are however data governance or management related. The two architecture related best practices are 'Top Payoff is Aligning Unstructured with Structured Data' and 'Plan Your Sandbox For Performance'. Both of them are supported in the reference architecture. The first is represented by the different Data Source components and by the Information Extraction component to extract structure from semi-structured and unstructured data and use this structure to integrate data into the Enterprise Data Warehouse. The second is supported due to the Sandbox component.

5.4 Verification Summary

In conclusion, the reference architecture was generally a good fit to describe the presented concrete architectures. This is a good indication for correctness and utility of the reference architecture, for its compliance to concrete system and that it can be effectively adapted and instantiated into a concrete architecture. There are, however, still some adjustments to make based on the insights above. The biggest issue of the reference architecture is how the backflow or propagation of data into other systems is modelled. A mere pull-oriented Data Access API seems not to be enough. Considering the information from the concrete architectures this can best be fixed by adding an optional serving storage area in front of the Data Access API. Additionally, the definition of the Data Access API needs to be changed to include push-like data propagation and a publish-subscribe messaging system needs to be included as an alternative to the Data Access API. Another necessary adjustment is to change the definition of Analysis Data Stores and possibly Analysis Delta Stores to allow them to be virtual. That is, that they are structural mapping layers on top of the Enterprise Data Warehouse or the Raw Data Archive. This idea especially occurred with the usage of Hive as a query layer which imposes structure on top of HDFS. These changes are reflected in the implementation-oriented view of the reference architecture. An adjusted diagram can be found in Figure 5.9.

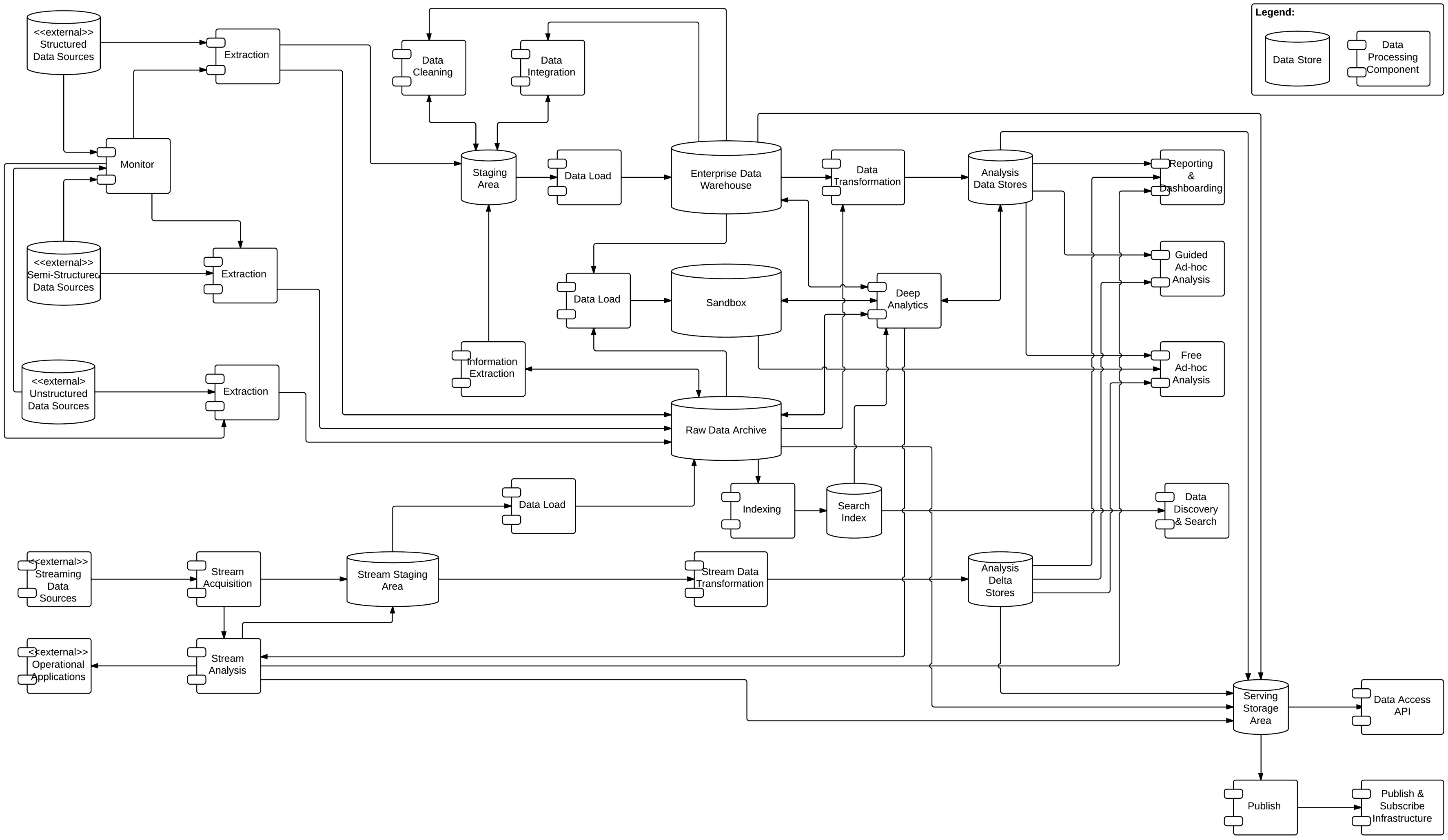


Figure 5.9: Adjusted Reference Architecture: Detailed implementation-oriented view

6 Conclusion

6.1 Conclusion and Outlook

In this master thesis I provided an overview of the ‘big data’ space. This is a rather broad and complex environment considering the four characteristics of data volume, velocity, variety, veracity and value. I started by researching common requirements from literature, classified them based on these five characteristics and identified relationships between those requirements. Based on this specification I developed a reference architecture including a functional and a more implementation-oriented view and discussed different technologies and how they can be applied to components within the reference architecture. Finally, the verification gave a good indication that the reference architecture is a proper fit to the space, relevant and provides utility for designing systems within the space.

One observation from the reference architecture is, that traditional data warehousing systems build the core and still play a large role. The architecture is therefore more evolutionary than revolutionary adding components and technology to the core of enterprise data warehousing to tackle the new challenges resulting from the newly emerging data characteristics mentioned above. This e.g. involves using heavily distributed and scalable software to tackle data volume, integrating new data source and using a raw data archive for storing semi-structured and unstructured data to tackle data variety and adding a ‘data highway’ to tackle data velocity. This observation is supported by a study published by the Data Warehouse Institute in 2011 [194]. The study predicts a growth in advanced analytics and data mining, predictive analytics, text analytics and real-time reports, all use cases associated with ‘big data’. It also shows a growing interest of related technologies discussed in this work, e.g. Hadoop and MapReduce, in-database analytics and in-memory databases. Besides that, the study however still shows a strong industry commitment into a central enterprise data warehouse and analytics processed within this enterprise data warehouse. I still want to note the possible bias of the Data Warehousing Institute due to its involvement in traditional data warehousing to put this in perspective.

For organizations this is good news. They can build on their existing investments into an analytical infrastructure based on an enterprise data warehouse and design a roadmap to extend it time by time to tackle their specific ‘big data’ related requirements and their particular situation. This is especially important considering the high amount of movement present in this space. This includes the industrial side, where a plethora of start-ups is emerging around ‘big data’ technologies developing a new kind of systems and where major software companies expand their business into this direction. It also includes academia, where researchers e.g. study characteristics of Hadoop and propose extensions, but also similar frameworks designed from scratch to tackle some of Hadoop’s shortcomings.

I also believe, that the reference architecture proposed here can help to discuss technologies already existing and newly emerging in the space, to reason about them, categorize them, map them to requirements and functional components and guide in applying them. Therefore, and based on the verification, I consider this work successful within its scope and assuming the limitations discussed above.

6.2 Limitations and Future Work

While the verification proved the proposed reference architecture relevant and provides good utility, there are still a number of limitations, which should be noted. The biggest issue, that pulls through the whole work is the absence of direct contact to practice. This shows in several points. The requirements had to been taken from literature without concrete stakeholders involved. It can therefore be questioned if the specification completely captures all relevant requirement or if certain points are overemphasized. Additionally, it was not possible to double check with stakeholders during the design phase to make slight adjustments based on practical experience. As the reference architecture should be based on best practices and proven concepts from practice, this is definitely an issue, that cannot completely made up from taking those best practices from literature. Finally, this did not allow for a more rigid, scenario- and interview-based verification of the reference architecture to check for relevance or to apply the reference architecture in a concrete project situation.

Possible follow-up work should therefore be aiming at reviewing different parts, requirements specification, the requirements architecture itself and the verification with practitioners who can contribute a lot with deep implementation experience. Especially the verification should be extended to include this experience. Possibilities are reviews of the document and description of the reference architecture by practitioner, interviews to focus on certain points of the reference architecture and to gather scenarios to apply the reference architecture to and test its fit. Application within a concrete project situation or a reference application would also be great measures to check suitability and applicability.

Obvious other limitations lie in the scope of this work and what could be covered within this scope. First, the technology description is rather coarse-grained and emphasizes on general concepts (e.g. column-oriented storage for databases) and product families (e.g. Key Value Stores) than concrete products. With this I give a starting point, some general advantages and disadvantages that can be expected in products implementing certain concepts and being part of a certain product family and some indication how those fit into the reference architecture. However, different products can combine those concepts in different ways and can have diverse characteristics (e.g. considering scalability and consistency) even within a product families based on how they are implemented (e.g. using different distribution models).

A very useful anchor point for further work would therefore be to deepen the discussion into a more fine-grained product dimension. This would involve the development of a fine-grained criteria catalogue and placing different products within this criteria space considering concepts they implement and characteristics they offer. It would also involve benchmarking of these products. That could mean to develop a general benchmark and run experiments over it with comparable hardware configuration. While there is literature that benchmarks different products, e.g. by Rabl et al. [181], this is largely scattered with a lack of a standard benchmark. TCP-H is e.g. no candidate as it immediately excludes all system that are not ACID-compliant and therefore a large amount of the data management systems discussed in this work. Benchmarks in literature are typically focussed on a small amount of systems and difficult to compare as they use different benchmarking methods (e.g. different workloads and different analytical algorithms) and as they are run with different hardware configurations.

Second, I focus on infrastructure software. There are however several other aspects of a ‘big data’ system. One would be to discuss which components of the reference architecture are best suited for certain analytical methods and algorithms. MapReduce is e.g. good for algorithms that are ‘trivially parallelizable’. Granville [124] provides some examples of cases, where it is not. There is research available that discusses which analytical algorithms can be implemented using MapReduce, how they are best implemented and for which algorithms MapReduce is not optimal [84, 154]. There is also a large body of literature about parallel algorithms and statistical models over massive amounts of data in general. A currently published, joint report of several researchers and academic committees offers a good overview of this research direction [88]. It would be valuable to put this research into perspective and create an overview. It would also be valuable to consolidate methods and algorithms to implement some of the reference architecture’s components, e.g. for data integration and information extraction, and to create an overview of their advantages and disadvantages. These methods can also have an influence on the architecture. Manually creating data integration mappings is e.g. a bottle-neck for data source adoption. Using methods as bootstrapping and ontology-based mappings can therefore improve evolvability of the architecture.

Additionally, I leave out considerations about hardware, deployment and hardware-related software. Especially as most of the infrastructure software I discuss works on cluster and aims at parallel computation, hardware architecture considerations are an important piece of the puzzle. This can include a discussion about cloud-based deployment and for which components of the architecture it is suitable. It also includes discussion of more low-level software components, e.g. for cluster management and monitoring.

Bibliography

[1] The Friend of a Friend (FOAF) project. Website, FOAF project. URL http://www.foaf-project.org/. Accessed: 05-05-2013.

[2] Linked Data - Connect Distributed Data across the Web. Website, Linked Open Data Project. URL http://linkeddata.org/. Accessed: 02-05-2013.

[3] W3C Semantic Web Activity. Website, W3C. URL http://www.w3.org/2001/sw/. Accessed: 05-05-2013.

[4] IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. Standard Specification IEEE 1471-2000, IEEE Computer Society, 2000u.

[5] RDF Primer. W3c recommendation, W3C, February 2004. URL http://www.w3.org/TR/rdf-primer/.

[6] SPARQL Query Language for RDF. W3c recommendation, W3C, January 2008. URL http://www.w3.org/TR/rdf-sparql-query/.

[7] Systems and software engineering — Architecture description. Standard Specification ISO/IEC/IEEE 42010:2011, ISO/IEC/IEEE, 2011. URL http://www.iso-architecture.org/ieee-1471/.

[8] What is Apache Mahout? Online documentation, Apache Software Foundation, 2012. URL http://mahout.apache.org/. Accessed: 17-09-2013.

[9] Apache Cassandra 1.2 Documentation. Online documentation, Datastax, 2012. URL http://www.datastax.com/docs/1.2/index. Accessed: 27-03-2013.

[10] Global Technology Outlook 2012. Whitepaper, IBM Research, 2012. URL http://www.zurich.ibm.com/pdf/isl/infoportal/GTO_2012_Booklet.pdf.

[11] Big Data Now: 2012 Edition. Techreport, O'Reilly Media, Inc., 2012.

[12] Hadoop 1.2.1 Documentation. Online documentation, Apache Software Foundation, 2013. URL http://hadoop.apache.org/docs/r1.2.1/index.html. Accessed: 27-03-2013.

[13] Hive Documentation. Online documentation, Apache Software Foundation, 2013. URL https://cwiki.apache.org/confluence/display/Hive/Home. Accessed: 27-03-2013.

[14] The Apache HBase Reference Guide. Online documentation, Apache Software Foundation, 2013. URL http://hbase.apache.org/book/book.html. Accessed: 27-03-2013.

[15] Welcome to Apache™ Hadoop®! Online documentation, Apache Software Foundation, 2013. URL http://hadoop.apache.org/. Accessed: 27-03-2013.

[16] Apache Hadoop 2.1.0-beta Documentation. Online documentation, Apache Software Foundation, 2013. URL http://hadoop.apache.org/docs/r1.2.1/index.html. Accessed: 17-09-2013.

[17] Welcome to HCatalog! Online documentation, Apache Software Foundation, 2013. URL http://hive.apache.org/hcatalog/index.html. Accessed: 17-09-2013.

[18] Drill Overview. Online documentation, Apache Software Foundation, 2013. URL http://incubator.apache.org/drill/drill_overview.html. Accessed: 17-09-2013.

[19] Apache Oozie Workflow Scheduler for Hadoop. Online documentation, Apache Software Foundation, 2013. URL http://incubator.apache.org/drill/drill_overview.html. Accessed: 17-09-2013.

[20] Welcome to Apache Flume. Online documentation, Apache Software Foundation, 2013. URL http://flume.apache.org/. Accessed: 27-03-2013.

[21] Welcome to Chukwa! Online documentation, Apache Software Foundation, 2013. URL http://incubator.apache.org/chukwa/. Accessed: 27-03-2013.

[22] Apache Kafka - A high-throughput distributed messaging system. Online documentation, Apache Software Foundation, 2013. URL http://kafka.apache.org/. Accessed: 27-03-2013.

[23] What the Enterprise Requires: The Platform for Big Data. Company website, Cloudera, Inc., 2013. URL http://www.cloudera.com/content/cloudera/en/products.html. Accessed: 05-04-2013.

[24] Google Flutrends. Website, Google, 2013. URL http://www.google.org/flutrends/. Accessed: 05-04-2013.

[25] HP Reference Architecture for MapR M5. Whitepaper, Hewlett-Packard, January 2013.

[26] Big Data Solutions. Company website, Hewlett-Packard, 2013. URL http://www8.hp.com/us/en/business-solutions/big-data.html. Accessed: 05-04-2013.

[27] InfoSphere Platform: Big Data Analytics. Company website, IBM, 2013. URL http://www-01.ibm.com/software/data/infosphere/bigdata-analytics.html. Accessed: 05-04-2013.

[28] The Big Data Hub. Company website, IBM, 2013. URL http://www.ibmbigdatahub.com/. Accessed: 05-04-2013.

[29] IBM Stream Computing. Company website, IBM, 2013. URL http://www-01.ibm.com/software/data/infosphere/stream-computing/. Accessed: 05-04-2013.

[30] What is big data? Company website, IBM, 2013. URL http://www-01.ibm.com/software/data/bigdata/. Accessed: 05-04-2013.

[31] Big Data. Company website, Microsoft, 2013. URL http://www.microsoft.com/enterprise/it-trends/big-data/. Accessed: 05-04-2013.

[32] Oracle and Big Data: Big Data for the Enterprise. Company website, Oracle, 2013. URL http://www.oracle.com/us/technologies/big-data/index.html. Accessed: 05-04-2013.

[33] Put big data to work for your business – with SAP solutions and technology. Company website, SAP, 2013. URL http://www54.sap.com/solutions/big-data/software/overview.html. Accessed: 05-04-2013.

[34] SAP HANA integrates predictive analytics, text and big data in a single package. Company website, SAP, 2013. URL http://www54.sap.com/solutions/tech/in-memory-computing-hana/software/analytics/big-data.html. Accessed: 05-04-2013.

[35] Big Data - What Is It? Company website, SAS, 2013. URL http://www.sas.com/big-data/. Accessed: 05-04-2013.

[36] Daniel J. Abadi. Tradeoffs between Parallel Database Systems, Hadoop, and HadoopDB as Platforms for Petabyte-Scale Analysis. In *Proceedings of the 22nd International Conference on Scientific and Statistical Database Management*, SSDBM '10, pages 1–3, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13817-9, 978-3-642-13817-1. URL http://dl.acm.org/citation.cfm?id=1876037.1876039.

[37] Daniel J. Abadi. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer*, 45(2):37–42, February 2012. doi: 10.1109/MC.2012.33.

[38] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 967–980, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376712.

[39] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column-Oriented Database Systems. In *Proceedings of the VLDB Endowment*, volume 2 of *PVLDB*, pages 1664–1665. VLDB Endowment, August 2009. URL http://dl.acm.org/citation.cfm?id=1687553.1687625.

[40] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Avi Silberschatz, and Alexander Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *Proceedings of the VLDB Endowment*, volume 2 of *PVLDB*, pages 922–933. VLDB Endowment, August 2009. URL http://dl.acm.org/citation.cfm?id=1687627.1687731.

[41] Charu C. Aggarwal, editor. *Data Streams: Models and Algorithms*, volume 31 of *Advances in Database Systems*. Springer US, 2007. doi: 10.1007/978-0-387-47534-9.

[42] Charu C. Aggarwal. An Introduction to Data Streams. In Charu C. Aggarwal, editor, *Data Streams: Models and Algorithms*, volume 31 of *Advances in Database Systems*, pages 1–8. Springer US, 2007. ISBN 978-0-387-28759-1. doi: 10.1007/978-0-387-47534-9_1.

[43] Charu C. Aggarwal and Philip S. Yu. A Survey of Synopsis Construction in Data Streams. In Charu C. Aggarwal, editor, *Data Streams: Models and Algorithms*, volume 31 of *Advances in Database Systems*, pages 169–207. Springer US, 2007. ISBN 978-0-387-28759-1. doi: 10.1007/978-0-387-47534-9_9.

[44] Vijay Srinivas Agneeswaran. Big-Data – Theoretical, Engineering and Analytics Perspective. In *Big Data Analytics*, volume 7678 of *Lecture Notes in Computer Science*, pages 8–15. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-35541-7. doi: 10.1007/978-3-642-35542-4_2.

[45] Divyakant Agrawal, Philip Bernstein, Elisa Bertino, Susan Davidson, Umeshwar Dayal, Michael Franklin, Johannes Gehrke, Laura Haas, Alon Halevy, Jiawei Han, H. V. Jagadish, Alexandros Labrinidis, Sam Madden, Yannis Papakonstantinou, Jignesh M. Patel, Raghu Ramakrishnan, Kenneth Ross, Cyrus Shahabi, Dan Suciu, Shiv Vaithyanathan, and Jennifer Widom. Challenges

and Opportunities with Big Data: A community white paper developed by leading researchers across the United States. Whitepaper, Computing Community Consortium, March 2012. URL http://cra.org/ccc/docs/init/bigdatawhitepaper.pdf.

[46] Rakesh Agrawal, Anastasia Ailamaki, Philip A. Bernstein, Eric A. Brewer, Michael J. Carey, Surajit Chaudhuri, Anhai Doan, Daniela Florescu, Michael J. Franklin, Hector Garcia-Molina, Johannes Gehrke, Le Gruenwald, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, Hank F. Korth, Donald Kossmann, Samuel Madden, Roger Magoulas, Beng Chin Ooi, Tim O'Reilly, Raghu Ramakrishnan, Sunita Sarawagi, Michael Stonebraker, Alexander S. Szalay, and Gerhard Weikum. The Claremont Report on Database Research. *Communications of the ACM*, 52(6):56–65, June 2009. ISSN 0001-0782. doi: 10.1145/1516046.1516062.

[47] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Nicola Onose, Pouria Pirzadeh, Rares Vernica, and Jian Wen. ASTERIX: An Open Source System for "Big Data" Management and Analysis (Demo). In *Proceedings of the VLDB Endowment*, volume 5 of *PVLDB*, pages 1898–1901. VLDB Endowment, August 2012. URL http://dl.acm.org/citation.cfm?id=2367502.2367532.

[48] Amineh Amini, Hadi Saboohi, and Nasser B. Nemat. A RDF-based Data Integration Framework. *The Computing Research Repository*, abs/1211.6273, 2012.

[49] Chris Anderson. The End of Theory: The Data Deluge Makes the Scientific Method Obsolete. *Wired Magazine*, 16.07, July 2008. URL http://www.wired.com/science/discoveries/magazine/16-07/pb_theory.

[50] Samuil Angelov, Jos J.M. Trienekens, and Paul Grefen. Towards a Method for the Evaluation of Reference Architectures: Experiences from a Case. In Ron Morrison, Dharini Balasubramaniam, and Katrina Falkner, editors, *Software Architecture*, volume 5292 of *Lecture Notes in Computer Science*, pages 225–240. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-88029-5. doi: 10.1007/978-3-540-88030-1_17.

[51] Samuil Angelov, Paul Grefen, and Da Greefhorst. A Classification of Software Reference Architectures: Analyzing Their Success and Effectiveness. In *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, WICSA/ECSA '09, pages 141–150, 2009. doi: 10.1109/WICSA.2009.5290800.

[52] Samuil Angelov, Paul Grefen, and Danny Greefhorst. A framework for analysis and design of software reference architectures. *Information and Software Technology*, 54(4):417–431, April 2012. doi: 10.1016/j.infsof.2011.11.009. URL http://www.sciencedirect.com/science/article/pii/S0950584911002333.

[53] Thilini Ariyachandra and Hugh Watson. Key organizational factors in data warehouse architecture selection. *Decision Support Systems*, 49(2):200–212, 2010. ISSN 0167-9236. doi: http://dx.doi.org/10.1016/j.dss.2010.02.006. URL http://www.sciencedirect.com/science/article/pii/S0167923610000436.

[54] Muhammad A. Babar and Ian Gorton. Comparison of Scenario-Based Software Architecture Evaluation Methods. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, APSEC '04, pages 600–607, 2004. doi: 10.1109/APSEC.2004.38.

[55] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *Proceedings of the Twenty-First ACM SIGMOD-*

SIGACT-SIGART Symposium on Principles of Database Systems, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM. ISBN 1-58113-507-6. doi: 10.1145/543613.543615.

[56] Kapil Bakshi. Considerations for Big Data: Architecture and Approach. In *Proceedings of the IEEE Aerospace Conference*. IEEE, March 2012. doi: 10.1109/AERO.2012.6187357.

[57] Wolf-Tilo Balke. Introduction to Information Extraction: Basic Notions and Current Trends. *Datenbank-Spektrum*, 12(2):81–88, 2012. ISSN 1618-2162. doi: 10.1007/s13222-012-0090-x.

[58] Dominic Barton and David Court. Making Advanced Analytics Work for You. *Harvard Business Review*, October 2012:78–84, October 2012.

[59] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, 2nd edition edition, 2003.

[60] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, 3rd edition edition, 2012.

[61] Andreas Bauer and Holger Günzel, editors. *Data Warehouse Systeme: Architektur, Entwicklung, Anwendung*. dpunkt.verlag GmbH, 4th edition edition, 2013.

[62] Edmon Begoli. A Short Survey on the State of the Art in Architectures and Platforms for Large Scale Data Analysis and Knowledge Discovery from Data. In *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, WICSA/ECSA '12, pages 177–183, New York, NY, USA, 2012. ISBN 978-1-4503-1568-5. doi: 10.1145/2361999.2362039.

[63] Edmon Begoli and James Horey. Design Principles for Effective Knowledge Discovery from Big Data. In *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, WICSA/ECSA '12, pages 215–218, New York, NY, USA, 2012. doi: 10.1109/WICSA-ECSA.212.32.

[64] Alexander Behm, Vinayak R. Borkar, MichaelJ. Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J. Tsotras. ASTERIX: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29:185–216, 2011. ISSN 0926-8782. doi: 10.1007/s10619-011-7082-y.

[65] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl-Christian Kanne, Fatma Özcan, and Eugene J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. In *Proceedings of the VLDB Endowment*, volume 4 of *PVLDB*, pages 1272–1283, 2011.

[66] Kenneth P. Birman, Daniel A. Freedman, Qi Huang, and Patrick Dowell. Overcoming CAP with Consistent Soft-State Replication. *Computer*, 45(2):50–58, February 2012.

[67] Christian Bizer, Peter Boncz, Michael L. Brodie, and Orri Erling. The Meaningful Use of Big Data: Four Perspectives – Four Challenges. *SIGMOD Record*, 40(4):56–60, January 2011. doi: 10.1145/2094114.2094129.

[68] Jens Bleiholder and Felix Naumann. Data Fusion. *ACM Computing Surveys*, 41(1):1:1–1:41, January 2009. ISSN 0360-0300. doi: 10.1145/1456650.1456651.

[69] Dario Bonino and Luigi De Russis. Mastering Real-Time Big Data With Stream Processing Chains. *XRDS*, 19(1):83–86, September 2012. ISSN 1528-4972. doi: 10.1145/2331042.2331050.

[70] L. Bonnet, A. Laurent, M. Sala, B. Laurent, and N. Sicard. Reduce, You Say: What NoSQL Can Do for Data Aggregation and BI in Large Repositories. In *Proceedings of the 22nd International*

Workshop on Database and Expert Systems Applications, DEXA '11, pages 483–488, 29 2011-sept. 2 2011. doi: 10.1109/DEXA.2011.71.

[71] Vinayak Borkar, Michael J. Carey, and Chen Li. Inside "Big Data management": Ogres, Onions, or Parfaits? In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 3–14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0790-1. doi: 10.1145/2247596.2247598.

[72] Vinayak R. Borkar, Michael J. Carey, and Chen Li. Big Data Platforms: What's Next? *XRDS*, 19(1):44–49, September 2012. doi: 10.1145/2331042.2331057.

[73] Danah Boyd and Kate Crawford. Six Provocations for Big Data. In *A Decade in Internet Time: Symposium on the Dynamics of the Internet and Society*, September 2011. doi: 10.2139/ssrn.1926431.

[74] Mary Breslin. Data Warehousing Battle of the Giants: Comparing the Basics of the Kimball and Inmon Models. *Business Intelligence Journal*, 9(1):6–20, 2004.

[75] E. Brewer. CAP Twelve Years Later: How the "Rules" Have Changed. *Computer*, 45(2): 23–29, February 2012. doi: 10.1109/MC.2012.37. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=6133253&contentType=Journals+%26+Magazines.

[76] Eric A. Brewer. Towards Robust Distributed Systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, New York, NY, USA, 2000. ACM. ISBN 1-58113-183-6. doi: 10.1145/343477.343502. URL http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

[77] Doug Cackett. Information Management and Big Data: A Reference Architecture. Whitepaper, Oracle, October 2012.

[78] Rick Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Record*, 39(4):12–27, May 2011. ISSN 0163-5808. doi: 10.1145/1978915.1978919.

[79] Amit Chakrabarti. *CS85: Data Stream Algorithms - Lecture Notes*. Dartmouth College, December 2011. URL http://www.cs.dartmouth.edu/~ac/Teach/CS49-Fall11/Notes/lecnotes.pdf.

[80] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2): 4:1–4:26, June 2008. ISSN 0734-2071. doi: 10.1145/1365815.1365816.

[81] Surajit Chaudhuri. What Next? A Half-Dozen Data Management Research Goals for Big Data and the Cloud. In *Proceedings of the 31st Symposium on Principles of Database Systems*, PODS '12, New York, NY, USA, 2012. ACM. doi: 10.1145/2213556.2213558. URL http://doi.acm.org/10.1145/2213556.2213558.

[82] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, March 1997. ISSN 0163-5808. doi: 10.1145/248603.248616.

[83] Jinchuan Chen, Yueguo Chen, Xiaoyong Du, Cuiping Li, Jiaheng Lu, Suyun Zhao, and Xuan Zhou. Big data challenge: a data management perspective. *Frontiers of Computer Science*, 7 (2):157–164, April 2013. doi: 10.1007/s11704-013-3903-7.

[84] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for Machine Learning on Multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems*, NIPS '06, pages 281–288. MIT Press, 2006.

[85] Robert Cloutier, Gerrit Muller, Dinesh Verma, Roshanak Nilchiani, Eirik Hole, and Mary Bone. The Concept of Reference Architectures. *Systems Engineering*, 13(1):14–27, 2010. doi: 10.1002/sys.20129. URL http://dx.doi.org/10.1002/sys.20129.

[86] E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP (On-Line Analytical Processing) to User-Analysts: An IT Mandate. E. F. Codd and Associates, 1993.

[87] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. MAD Skills: New Analysis Practices for Big Data. In *Proceedings of the VLDB Endowment*, volume 2 of *PVLDB*, pages 1481–1492. VLDB Endowment, August 2009. URL http://dl.acm.org/citation.cfm?id=1687553.1687576.

[88] Committee on the Analysis of Massive Data, Committee on Applied and Theoretical Statistics, Board on Mathematical Sciences and Their Applications, Division on Engineering and Physical Sciences, and National Research Council. *Frontiers in Massive Data Analysis*. The National Academies Press, 2013. ISBN 9780309287784. URL http://www.nap.edu/openbook.php?record_id=18374.

[89] Sudipto Das, Yannis Sismanis, Kevin S. Beyer, Rainer Gemulla, Peter J. Haas, and John McPherson. Ricardo: Integrating R and Hadoop. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 987–998, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807275.

[90] Anish Das Sarma, Xin Dong, and Alon Halevy. Bootstrapping Pay-As-You-Go Data Integration Systems. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 861–874, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376702.

[91] Thomas H. Davenport and D.J. Patil. Data Scientist: The Sexiest Job of the 21st Century. *Harvard Business Review*, October 2012:70–76, October 2012.

[92] Thomas H. Davenport, Paul Barth, and Randy Bean. How 'Big Data' Is Different. *MIT Sloan Management Review*, Fall 2012, July 2012. URL http://sloanreview.mit.edu/article/how-big-data-is-different/.

[93] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation*, volume 6 of *OSDI '04*, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251254.1251264.

[94] Jeffrey Dean and Sanjay Ghemawat. MapReduce: A Flexible Data Processing Tool. *Communications of the ACM*, 53(1):72–77, January 2010. ISSN 0001-0782. doi: 10.1145/1629175.1629198.

[95] Tom Deutsch. Experimentation as a Corporate Strategy for Big Data. Blog Entry, October 2012. URL http://ibmdatamag.com/2012/10/experimentation-as-a-corporate-strategy-for-big-data/. Accessed: 17-09-2013.

[96] Barry Devlin. The Big Data Zoo - Taming the Beasts. Technical report, 9sight Consulting, October 2012.

[97] Dr. Barry Devlin, Shawn Rogers, and John Myers. Big Data Comes of Age. Research report, EMA Inc. and 9sight Consulting, November 2012.

[98] David DeWitt. MapReduce: A major step backwards. Blog Entry, January 2008. URL http://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html. Accessed: 13-08-2013.

[99] David DeWitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, June 1992. ISSN 0001-0782. doi: 10.1145/129888.129894.

[100] Maria M. Dias, Tania C. Tait, André Luís A. Menolli, and Roberto C.S. Pacheco. Data Warehouse Architecture through Viewpoint of Information System Architecture. In *Proceedings of the 2008 International Conference on Computational Intelligence for Modelling Control Automation*, CIMCA '08, pages 7–12, 2008. doi: 10.1109/CIMCA.2008.129.

[101] Jean-Pierre Dijcks. Oracle: Big Data for the Enterprise. June, Oracle, 2013.

[102] Thomas W. Dinsmore. Analytic Applications (Part One). Blog Entry, January 2013. URL http://portfortune.wordpress.com/2013/01/04/analytic-applications-part-one/. Accessed: 17-09-2013.

[103] Thomas W. Dinsmore. Analytic Applications (Part Two): Managerial Analytics. Blog Entry, January 2013. URL http://portfortune.wordpress.com/2013/01/10/analytic-applications-part-two-managerial-analytics/. Accessed: 17-09-2013.

[104] AnHai Doan, Jeffrey F. Naughton, Raghu Ramakrishnan, Akanksha Baid, Xiaoyong Chai, Fei Chen, Ting Chen, Eric Chu, Pedro DeRose, Byron Gao, Chaitanya Gokhale, Jiansheng Huang, Warren Shen, and Ba-Quy Vuong. Information Extraction Challenges in Managing Unstructured Data. *SIGMOD Record*, 37(4):14–20, March 2008. ISSN 0163-5808. doi: 10.1145/1519103.1519106. URL http://doi.acm.org/10.1145/1519103.1519106.

[105] L. Dobrica and E. Niemela. A Survey on Software Architecture Analysis Methods. *IEEE Transactions on Software Engineering*, 28(7):638–653, July 2002. ISSN 0098-5589. doi: 10.1109/TSE.2002.1019479.

[106] X.L. Dong and D. Srivastava. Big Data Integration. In *Proceedings of the 29th IEEE International Conference on Data Engineering*, ICDE '13, pages 1245–1248, 2013. doi: 10.1109/ICDE.2013.6544914.

[107] Andrea Freyer Dugas, Yu-Hsiang Hsieh, Scott R. Levin, Jesse M. Pines, Darren P. Mareiniss, Amir Mohareb, Charlotte A. Gaydos, Trish M. Perl, and Richard E. Rothman. Google Flu Trends: Correlation With Emergency Department Influenza Rates and Crowding Metrics. *Clinical Infectious Diseases*, 54(4):463–469, January 2012. doi: 10.1093/cid/cir883.

[108] Edd Dumbill. Planning for Big Data: A CIO's Handbook to the Changing Data Landscape. Technical report, O'Reilly Media, Inc., 2012.

[109] George Dyson, Kevin Kelly, Stewart Brand, W. Daniel Hillis, Sean Carroll, Jaron Lanier, Joseph Traub, John Horgan, Bruce Sterling, Douglas Rushkoff, Oliver Morton, Daniel Everett, Gloria Origgi, Lee Smolin, and Joel Garreau. On Chris Anderson's 'The End of Theory'. Edge.org Comments, June 2008. URL http://www.edge.org/discourse/the_end_of_theory.html. Accessed: 12-04-2013.

[110] Wenfei Fan, Xibei Jia, Jianzhong Li, and Shuai Ma. Reasoning about Record Matching Rules. In

Proceedings of the VLDB Endowment, volume 2 of *PVLDB*, pages 407–418. VLDB Endowment, August 2009. URL http://dl.acm.org/citation.cfm?id=1687627.1687674.

[111] Danyel Fisher, Rob DeLine, Mary Czerwinski, and Steven Drucker. Interactions with Big Data Analytics. *interactions*, 19(3):50–59, May 2012. ISSN 1072-5520. doi: 10.1145/2168931.2168943.

[112] Avrilia Floratou, Nikhil Teletia, David J. DeWitt, Jignesh M. Patel, and Donghui Zhang. Can the Elephants Handle the NoSQL Onslaught? In *Proceedings of the VLDB Endowment*, volume 5 of *PVLDB*, pages 1712–1723. VLDB Endowment, August 2012. URL http://dl.acm.org/citation.cfm?id=2367502.2367511.

[113] Martin Fowler and Pramod J. Sadalage. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, 2012.

[114] Matthias Galster and Paris Avgeriou. Empirically-grounded Reference Architectures: A Proposal. In *Proceedings of the Joint ACM SIGSOFT Conference on Quality of Software Architectures and ACM SIGSOFT Symposium on Architecting Critical Systems*, QoSA-ISARCS '11, pages 153–158. ACM, 2011. doi: 10.1145/2000259.2000285.

[115] John Gantz and David Reinsel. THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. Study report, IDC, December 2012. URL www.emc.com/leadership/digital-universe/index.htm.

[116] David Garlan and Dewayne E. Perry. Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, April 1995. ISSN 0098-5589. URL http://dl.acm.org/citation.cfm?id=205313.205314.

[117] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a High-Level Dataflow System on top of MapReduce: The Pig Experience. In *Proceedings of the VLDB Endowment*, volume 2 of *PVLDB*, pages 1414–1425. VLDB Endowment, August 2009. URL http://dl.acm.org/citation.cfm?id=1687553.1687568.

[118] Anne E. Gattiker, Fade H. Gebara, Ahmed Gheith, H. Peter Hofstee, Damir A. Jamsek, Jian Li, Evan Speight, Ju Wei Shi, Guan Cheng Chen, and Peter W. Wong. Understanding System and Architecture for Big Data. Research Report RC25281 (AUS1204-004), IBM Research Division, April 2012.

[119] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, October 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945450.

[120] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601.

[121] Seth Gilbert and Nancy A. Lynch. Perspectives on the CAP Theorem. *Computer*, 45(2):30–36, February 2012. doi: 10.1109/MC.2011.389.

[122] Jeremy Ginsberg, Matthew H. Mohebbi, Rajan S. Patel, Lynnette Brammer, Mark S. Smolinski, and Larry Brilliant. Detecting influenza epidemics using search engine query data. *Nature*, 457:1012–1014, February 2009. URL http://www.nature.com/nature/journal/v457/n7232/full/nature07634.html.

[123] Ian Gorton. *Essential Software Architecture.* Springer Heidelberg Dordrecht London New York, 2011.

[124] Vincent Granville. What MapReduce can't do. Blog Entry, January 2013. URL http://www.analyticbridge.com/profiles/blogs/what-mapreduce-can-t-do. Accessed: 17-09-2013.

[125] Paul Grefen, Nikolay Mehandjiev, Giorgos Kouvas, Georg Weichhart, and Rik Eshuis. Dynamic business network process management in instant virtual enterprises. *Computers in Industry*, 60 (2):86–103, February 2009. ISSN 0166-3615. doi: http://dx.doi.org/10.1016/j.compind.2008.06.006. URL http://www.sciencedirect.com/science/article/pii/S0166361508000675.

[126] Rajeev Gupta, Himanshu Gupta, and Mukesh Mohania. Cloud Computing and Big Data Analytics: What Is New from Databases Perspective? In Srinath Srinivasa and Vasudha Bhatnagar, editors, *Big Data Analytics*, volume 7678 of *Lecture Notes in Computer Science*, pages 42–61. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-35541-7. doi: 10.1007/978-3-642-35542-4_5.

[127] A. Halevy, P. Norvig, and F. Pereira. The Unreasonable Effectiveness of Data. *IEEE Intelligent Systems*, 24(2):8–12, March 2009. ISSN 1541-1672. doi: 10.1109/MIS.2009.36.

[128] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data Integration: The Teenage Years. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, pages 9–16. VLDB Endowment, September 2006. URL http://dl.acm.org/citation.cfm?id=1182635.1164130.

[129] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 981–992, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376713.

[130] Pat Helland. If You Have Too Much Data, then 'Good Enough' Is Good Enough. *Communications of the ACM*, 54(6):40–47, June 2011. doi: 10.1145/1953122.1953140.

[131] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. The MADlib Analytics Library or MAD Skills, the SQL. In *Proceedings of the VLDB Endowment*, volume 5 of *PVLDB*, pages 1700–1711. VLDB Endowment, August 2012. URL http://dl.acm.org/citation.cfm?id=2367502.2367510.

[132] Yin Huai, Rubao Lee, Simon Zhang, Cathy H. Xia, and Xiaodong Zhang. DOT: A Matrix Model for Analyzing, Optimizing and Deploying Software for Big Data Analytics in Distributed Systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 4:1–4:14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038920.

[133] S. Humbetov. Data-Intensive Computing with Map-Reduce and Hadoop. In *Proceedings of the 6th International Conference on Application of Information and Communication Technologies*, AICT '12, pages 1–5, October 2012. doi: 10.1109/ICAICT.2012.6398489.

[134] M. Indrawan-Santiago. Database Research: Are We at a Crossroad? Reflection on NoSQL. In *Proceedings of the 15th International Conference on Network-Based Information Systems*, NBiS'2012, pages 45– –51, September 2012. doi: 10.1109/NBiS.2012.95.

[135] William H. Inmon. *Building the Data Warehouse.* John Wiley & Sons, Inc., New York, NY, USA, 1992. ISBN 0471569607.

[136] William H. Inmon and K. Krishnan. *Building the Unstructured Data Warehouse.* Technics Publications, LLC, jan 2011.

[137] Adam Jacobs. The Pathologies of Big Data. *ACM Queue*, 52(8):36–44, August 2009.

[138] Bin Jiang. Is Inmon's Data Warehouse Definition Still Accurate? Blog Entry, May 2012. URL http://www.b-eye-network.com/view/16066. Accessed: 31-07-2013.

[139] Jeff Jonas. There Is No Such Thing As A Single Version of Truth. Blog Entry, March 2006. URL http://jeffjonas.typepad.com/jeff_jonas/2006/03/there_is_no_suc.html. Accessed: 05-05-2013.

[140] Jeff Jonas. Data Beats Math. Blog Entry, April 2011. URL http://jeffjonas.typepad.com/jeff_jonas/2011/04/data-beats-math.html. Accessed: 05-04-2013.

[141] Stephen Kaisler, Frank Armour, J. Alberto Espinosa, and William Money. Big Data: Issues and Challenges Moving Forward. In *Proceedings of the 46th Hawaii International Conference on System Sciences*, HICSS '13, pages 995–1004, 2013.

[142] Ralph Kimball. *The Data Warehouse Toolkit.* John Wiley & Sons, In, 1996. ISBN 9780471153375. URL http://books.google.nl/books?id=VlBqcgAACAAJ.

[143] Ralph Kimball. The Evolving Role of the Enterprise Data Warehouse in the Era of Big Data Analytics. Whitepaper, Kimball Group, April 2011. URL http://www.kimballgroup.com/2011/04/29/the-evolving-role-of-the-enterprise-data-warehouse-in-the-era-of-big-data-analytics/.

[144] Ralph Kimball. Newly Emerging Best Practices for Big Data. Whitepaper, Kimball Group, September 2012. URL http://www.kimballgroup.com/2012/09/30/newly-emerging-best-practices-for-big-data/.

[145] Ralph Kimball, Laura Reeves, Margy Ross, and Warren Thornthwaite. *The Data Warehouse Lifecycle Toolkit.* John Wiley & Sons, Inc., 1998.

[146] Gerald Kotonya and Ian Sommerville. *Requirements Engineering - Process and Techniques.* John Wiley & Sons, Ltd, 1998.

[147] T. Kraska. Finding the Needle in the Big Data Systems Haystack. *IEEE Internet Computing*, 17(1):84–86, 2013. ISSN 1089-7801. doi: 10.1109/MIC.2013.10.

[148] T. Kraska and B. Trushkowsky. The New Database Architectures. *IEEE Internet Computing*, 17(3):72–75, 2013. ISSN 1089-7801. doi: 10.1109/MIC.2013.56.

[149] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: A Distributed Messaging System for Log Processing. In *Proceedings of the NetDB*, NetDB '11, Athens, Greece, June 2011. ACM.

[150] P.B. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, 1995. ISSN 0740-7459. doi: 10.1109/52.469759.

[151] Karl E. Kurbel. Information Systems Architecture. In *The Making of Information Systems*, pages 95–154. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-79260-4. doi: 10.1007/978-3-540-79261-1_3.

[152] Doug Laney. 3D Data Management: Controlling Data Volume, Velocity and Variety. Technical report, META Group, Inc (now Gartner, Inc.), February 2001. URL http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf.

[153] Jimmy Lin. MapReduce is Good Enough? If All You Have is a Hammer, Throw Away Everything That's Not a Nail! *The Computing Research Repository*, abs/1209.2191, 2012.

[154] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce.* Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, Sep 2010.

[155] Xiufeng Liu, Christian Thomsen, and Torben Bach Pedersen. MapReduce-based Dimensional ETL Made Easy. In *Proceedings of the VLDB Endowment*, volume 5 of *PVLDB*, pages 1882–1885. VLDB Endowment, August 2012. URL http://dl.acm.org/citation.cfm?id=2367502.2367528.

[156] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043593.

[157] Steve Lohr. The Age of Big Data. *The New York Times*, February 12th, 2012:SR1, February 2012. URL http://www.nytimes.com/2012/02/12/sunday-review/big-datas-impact-in-the-world.html.

[158] Peter Loos, Jens Lechtenbörger, Gottfried Vossen, Alexander Zeier, Jens Krüger, Jürgen Müller, Wolfgang Lehner, Donald Kossmann, Benjamin Fabian, Oliver Günther, and Robert Winter. In-memory Databases in Business Information Systems. *Business & Information Systems Engineering*, 3(6):389–395, 2011. doi: 10.1007/s12599-011-0188-y.

[159] Peter Loos, Stefan Strohmeier, Gunther Piller, and Reinhard Schütte. Comments on "In-Memory Databases in Business Information Systems". *Business & Information Systems Engineering*, 4 (4):213–223, 2012. doi: 10.1007/s12599-012-0222-8.

[160] Ashwin Machanavajjhala and Jerome P. Reiter. Big Privacy: Protecting Confidentiality in Big Data. *XRDS*, 19(1):20–23, September 2012. ISSN 1528-4972. doi: 10.1145/2331042.2331051.

[161] Sam Madden. From Databases to Big Data. *iEEE Internet Computing*, 16(3):4–6, 2012.

[162] Lev Manovich. Trending: The Promises and the Challenges of Big Social Data. In Matthew K. Gold, editor, *Debates in the Digital Humanities.* The University of Minnesota Press, 2011. URL http://lab.softwarestudies.com/2011/04/new-article-by-lev-manovich-trending.html.

[163] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela Hung Byers. Big data: The next frontier for innovation, competition, and productivity. Analyst report, McKinsey Global Institute, May 2011. URL http://www.mckinsey.com/insights/mgi/research/technology_and_innovation/big_data_the_next_frontier_for_innovation.

[164] Nathan Marz and James Warren. *Big Data - Principles and best practices of scalable realtime data systems.* Manning Publications, manning early access program - big data version 9 edition, 2013.

[165] Viktor Mayer-Schönberger and Kenneth Cukier. *Big Data - A Revolution That Will Transform How We Live, Work and Think.* John Murray (Publishers), 2013.

[166] Andrew McAfee and Erik Brynjolfsson. Big Data: The Management Revolution. *Harvard Business Review*, October 2012:60–68, October 2012.

[167] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *Communications of the ACM*, 54(6):114–123, June 2011. ISSN 0001-0782. doi: 10.1145/1953122.1953148. URL http://doi.acm.org/10.1145/1953122.1953148.

[168] Camille Mendler. M2M and big data. Website, The Economist: Intelligence Unit, 2013. URL http://digitalresearch.eiu.com/m2m/from-sap/m2m-and-big-data. Accessed: 05-05-2013.

[169] H.G. Miller and P. Mork. From Data to Decisions: A Value Chain for Big Data. *IT Professional*, 15(1):57–59, 2013. ISSN 1520-9202. doi: 10.1109/MITP.2013.11.

[170] Gilad Mishne, Jeff Dalton, Zhenghua Li, Aneesh Sharma, and Jimmy Lin. Fast Data in the Era of Big Data: Twitter's Real-Time Related Query Suggestion Architecture. *The Computing Research Repository*, abs/1210.7350:http://arxiv.org/abs/1210.7350, October 2012. URL http://arxiv.org/abs/1210.7350.

[171] Gerrit Muller and Piërre Laar. Researching Reference Architectures. In Pierre Van de Laar and Teade Punter, editors, *Views on Evolvability of Embedded Systems*, Embedded Systems, pages 107–119. Springer Netherlands, 2011. ISBN 978-90-481-9848-1. doi: 10.1007/978-90-481-9849-8_7.

[172] Elisa Yumi Nakagawa and Lucas Bueno Ruas de Oliveira. Using Systematic Review to Elicit Requirements of Reference Architectures. In *Anais do WER11 - Workshop em Engenharia de Requisitos*, Rio de Janeiro-RJ, Brasil, April 2011.

[173] Elisa Yumi Nakagawa, Martin Becker, and José Carlos Maldonado. A Knowledge-based Framework for Reference Architectures. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1197–1202, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0857-1. doi: 10.1145/2231936.2231964.

[174] Mathias Niepert. Statistical Relational Data Integration for Information Extraction. In Sebastian Rudolph, Georg Gottlob, Ian Horrocks, and Frank Harmelen, editors, *Reasoning Web. Semantic Technologies for Intelligent Data Access*, volume 8067 of *Lecture Notes in Computer Science*, pages 251–283. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-39783-7. doi: 10.1007/978-3-642-39784-4_7.

[175] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376726.

[176] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559865.

[177] Alex Pentland. Reinventing Society in the Wake of Big Data. Edge.org Conversation, August 2012. URL http://www.edge.org/conversation/reinventing-society-in-the-wake-of-big-data. Accessed: 11-04-2013.

[178] Christy Pettey and Laurence Goasduff. Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data. Press Release, June 2011. URL http://www.gartner.com/newsroom/id/1731916. Accessed: 27-02-2011.

[179] Dan Pritchett. BASE: An Acid Alternative. *Queue*, 6(3):48–55, May 2008. ISSN 1542-7730. doi: 10.1145/1394127.1394128.

[180] Ariel Rabkin and Randy Katz. Chukwa: a system for reliable large-scale log collection. In *Proceedings of the 24th International Conference on Large Installation System Administration*, LISA'10, pages 1–15, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924976.1924994.

[181] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving Big Data Challenges for Enterprise Application Performance Management. In *Proceedings of the VLDB Endowment*, volume 5, pages 1724–1735. VLDB Endowment, August 2012. URL http://dl.acm.org/citation.cfm?id=2367502.2367512.

[182] Erhard Rahm and Hong Hai Do. Data Cleaning: Problems and Current Approaches. *IEEE Data Engineering Bulletin*, 23, 2000.

[183] Anand Rajaraman. More data usually beats better algorithms. Blog Entry, March 2008. URL http://anand.typepad.com/datawocky/2008/03/more-data-usual.html. Accessed: 05-04-2013.

[184] Anand Rajaraman. More data usually beats better algorithms, Part 2. Blog Entry, April 2008. URL http://anand.typepad.com/datawocky/2008/04/data-versus-alg.html. Accessed: 05-04-2013.

[185] Anand Rajaraman. More data beats better algorithm at predicting Google earnings. Blog Entry, April 2008. URL http://anand.typepad.com/datawocky/2008/04/more-data-beats.html. Accessed: 05-04-2013.

[186] Thomas C. Redman. In a Big Data World, Don't Forget Experimentation. Blog Entry, May 2013. URL http://blogs.hbr.org/2013/05/in-a-big-data-world-dont-forge/. Accessed: 17-09-2013.

[187] Eric Redmond and Jim R. Wilson. *Seven Databases in Seven Weeks*. Pragmatic Bookshelf. Pragmatic Programmers, LLC, 1st edition edition, 2012.

[188] Mohammad Rifaie, K. Kianmehr, R. Alhajj, and M.J. Ridley. Data Warehouse Architecture and Design. In *Proceedings of the 2008 IEEE International Conference on Information Reuse and Integration*, IRI '08, pages 58–63, 2008. doi: 10.1109/IRI.2008.4583005.

[189] Suzanne Robertson and James Robertson. *Mastering the Requirements Process, 3rd Edition*. Pearson Education, Inc, 2012.

[190] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Inc, early release edition, 2013. Early Release: raw & unedited.

[191] Kim Rose. Hadoop and the age of experimentation. Blog Entry, July 2013. URL http://hortonworks.com/big-data-insights/hadoop-and-the-age-of-experimentation/. Accessed: 17-09-2013.

[192] Mattias Rost, Louise Barkhuus, Henriette Cramer, and Barry Brown. Representation and Communication: Challenges in Interpreting Large Social Media Datasets. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, CSCW '13, pages 357–362, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1331-5. doi: 10.1145/2441776.2441817. URL http://doi.acm.org/10.1145/2441776.2441817.

[193] Nick Rozanski and Eoin Woods. *Software Systems Architecture - Working With Stakeholders Using Viewpoints and Perspectives.* Pear, 2005.

[194] Philip Russom. Big Data Analytics. Best practices report, The Data Warehousing Institute, 2011.

[195] Philip Russom. Analytic Databases for Big Data. Technical report, The Data Warehousing Institute, October 2012.

[196] Michael Saecker and Volker Markl. Big Data Analytics on Modern Hardware Architectures: A Technology Survey. In Marie-Aude Aufaure and Esteban Zimányi, editors, *Business Intelligence*, volume 138 of *Lecture Notes in Business Information Processing*, pages 125–149. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-36317-7. doi: 10.1007/978-3-642-36318-4_6.

[197] Shashi Shekhar, Viswanath Gunturi, Michael R. Evans, and KwangSoo Yang. Spatial Big-Data Challenges Intersecting Mobility and Cloud Computing. In *Proceedings of the Eleventh ACM International Workshop on Data Engineering for Wireless and Mobile Access*, MobiDE '12, pages 1–6, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1442-8. doi: 10.1145/2258056.2258058.

[198] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies*, MSST '10, pages 1–10, 2010. doi: 10.1109/MSST.2010.5496972.

[199] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw-Hill, 6th edition edition, 2011.

[200] Hassan A. Sleiman and Rafael Corchuelo. Information Extraction Framework. In Juan M. Corchado Rodríguez, Javier Bajo Pérez, Paulina Golinska, Sylvain Giroux, and Rafael Corchuelo, editors, *Trends in Practical Applications of Agents and Multiagent Systems*, volume 157 of *Advances in Intelligent and Soft Computing*, pages 149–156. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-28794-7. doi: 10.1007/978-3-642-28795-4_18.

[201] Rick Smolan and Jennifer Erwitt. *The Human Face of Big Data.* Against All Odds Productions, January 2013.

[202] Sunil Soares. *Big Data Governance - An Emerging Imperative.* MC Press Online, LLC, 1st edition edition, 2012.

[203] Sunil Soares. *IBM InfoSphere: A Platform for Big Data Governance and Process Data Governance.* MC Press Online, LLC, February 2013.

[204] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Engineering*, ICDE '05, pages 2–11, April 2005. doi: 10.1109/ICDE.2005.1.

[205] Michael Stonebraker. The Case for Shared Nothing. *Database Engineering*, 9:4–9, 1986.

[206] Michael Stonebraker. Technical Perspective - One Size Fits All: An Idea Whose Time has Come and Gone. *Communications of the ACM*, 51(12):76–76, December 2008. ISSN 0001-0782. doi: 10.1145/1409360.1409379.

[207] Michael Stonebraker. SQL Databases v. NoSQL Databases. *Communications of the ACM*, 53 (4):10–11, April 2010. ISSN 0001-0782. doi: 10.1145/1721654.1721659.

[208] Michael Stonebraker and Rick Cattell. 10 Rules for Scalable Performance in 'Simple Operation' Datastores. *Communications of the ACM*, 54(6):72–80, June 2011. ISSN 0001-0782. doi: 10.1145/1953122.1953144.

[209] Michael Stonebraker, Chuck Bear, Uğur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stan Zdonik. One Size Fits All? – Part 2: Benchmarking Results. In *Proceedings of the Conference on Innovative Data Systems Research*, CIDR '07, pages 173–184, January 2007.

[210] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, September 2007. ISBN 978-1-59593-649-3. URL http://dl.acm.org/citation.cfm?id=1325851.1325981.

[211] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and Parallel DBMSs: Friends or Foes? *Communications of the ACM*, 53(1):64–71, January 2010. ISSN 0001-0782. doi: 10.1145/1629175.1629197.

[212] Michael Stonebraker, Daniel Bruckner, Ihab Ilyas, George Beskales, Mitch Cherniack, Stan Zdonik, Alexander Pagan, and Shan Xu. Data Curation at Scale: The Data Tamer System. In *Proceedings of the Conference on Innovative Data Systems Research*, CIDR '13, January 2013. URL http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper28.pdf.

[213] Michael Stonebraker, Sam Madden, and Pradeep Dubey. Intel "Big Data" Science and Technology Center Vision and Execution Plan. *SIGMOD Record*, 42(1):44–49, March 2013.

[214] Zhiquan Sui and Shrideep Pallickara. A Survey of Load Balancing Techniques for Data Intensive Computing. In Borko Furht and Armando Escalante, editors, *Handbook of Data Intensive Computing*, pages 157–168. Springer New York, 2011. ISBN 978-1-4614-1414-8. doi: 10.1007/978-1-4614-1415-5.

[215] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving Large-scale Batch Computed Data with Project Voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST '12, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2208461.2208479.

[216] Roshan Sumbaly, Jay Kreps, and Sam Shah. The "Big Data" Ecosystem at LinkedIn. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1125–1134, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2463707.

[217] Helen Sun and Peter Heller. Oracle Information Architecture: An Architect's Guide to Big Data. August, Oracle, 2012.

[218] Nassim N. Taleb. Beware the Big Errors of 'Big Data'. Wired Opinion, August 2013. URL http://www.wired.com/opinion/2013/02/big-data-means-big-errors-people/. Accessed: 12-04-2013.

[219] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. In *Proceedings of the VLDB Endowment*, volume 2 of *PVLDB*, pages 1626–1629. VLDB Endowment, August 2009. URL http://dl.acm.org/citation.cfm?id=1687553.1687609.

[220] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive – A Petabyte Scale Data Warehouse Using Hadoop. In *Proceedings of the 29th IEEE International Conference on Data Engineering*,

ICDE ’10, pages 996–1005, Los Alamitos, CA, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-5445-7. doi: http://doi.ieeecomputersociety.org/10.1109/ICDE.2010.5447738.

[221] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’10, pages 1013–1020, New York, NY, USA, June 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807278.

[222] Oliver Vogel, Ingo Arnold, Arif Chughtai, and Timo Kehrer. *Software Architecture: A Comprehensive Framework and Guide for Practitioners.* Springer-Verlag Berlin Heidelberg, 2011.

[223] Werner Vogels. Eventually Consistent. *Communications of the ACM*, 52(1):40–44, January 2009.

[224] Michael Walker. Data Veracity. Blog Entry, November 2012. URL http://www.datasciencecentral.com/profiles/blogs/data-veracity. Accessed: 05-04-2013.

[225] Steven Euijong Whang, David Menestrina, Georgia Koutrika, Martin Theobald, and Hector Garcia-Molina. Entity Resolution with Iterative Blocking. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’09, pages 219–232, New York, NY, USA, June 2009. ACM. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559870.

[226] Tom White. *Hadoop: The Definitive Guide.* O’Reilly Media, Inc, 2009.

[227] Karl E. Wiegers. *Software Requirements, 2nd Edition.* Microsoft Press, 2003.

[228] Thorsten Winsemann, Veit Köppen, and Gunter Saake. A Layered Architecture for Enterprise Data Warehouse Systems. In Marko Bajec and Johann Eder, editors, *Advanced Information Systems Engineering Workshops*, volume 112 of *Lecture Notes in Business Information Processing*, pages 192–199. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-31068-3. doi: 10.1007/978-3-642-31069-0_17.

[229] Lili Wu, Roshan Sumbaly, Chris Riccomini, Gordon Koo, Hyung Jin Kim, Jay Kreps, and Sam Shah. Avatara: OLAP for Webscale Analytics Products. In *Proceedings of the VLDB Endowment*, volume 5 of *PVLDB*, pages 1874–1877. VLDB Endowment, August 2012. URL http://dl.acm.org/citation.cfm?id=2367502.2367525.

[230] Yuqing Zhu, Philip S. Yu, and Jianmin Wang. Latency Bounding by Trading off Consistency in NoSQL Store: A Staging and Stepwise Approach. *The Computing Research Repository*, abs/1212.1046, December 2012.

[231] Paul C. Zikopoulos, Dirk deRoos, Krishnan Parasuraman, Thomas Deutsch, David Corrigan, and James Giles. *Harness the Power of Big Data: The IBM Big Data Platform.* McGraw-Hill, 2013.