Big Data Systems: A Software Engineering Perspective

ALI DAVOUDIAN, Carleton University, Canada
MENGCHI LIU, South China Normal University, China

Big Data Systems (BDSs) are an emerging class of scalable software technologies whereby massive amounts of heterogeneous data are gathered from multiple sources, managed, analyzed (in batch, stream or hybrid fashion), and served to end-users and external applications. Such systems pose specific challenges in all phases of software development lifecycle and might become very complex by evolving data, technologies, and target value over time. Consequently, many organizations and enterprises have found it difficult to adopt BDSs. In this article, we provide insight into three major activities of software engineering in the context of BDSs as well as the choices made to tackle them regarding state-of-the-art research and industry efforts. These activities include the engineering of requirements, designing and constructing software to meet the specified requirements, and software/data quality assurance. We also disclose some open challenges of developing effective BDSs, which need attention from both researchers and practitioners.

CCS Concepts: • **Information systems → Information integration**;

Additional Key Words and Phrases: Big Data, Big Data systems, software engineering, requirements engineering, software reference architecture, quality assurance

ACM Reference format:
Ali Davoudian and Mengchi Liu. 2020. Big Data Systems: A Software Engineering Perspective. *ACM Comput. Surv.* 53, 5, Article 110 (September 2020), 39 pages.
https://doi.org/10.1145/3408314

1 INTRODUCTION

Since 2011, the ease of data collection and storage along with the growing number of Big Data technologies and affordable infrastructures have severely motivated many organizations to intensify their business value by launching Big Data (or data-intensive) systems [Trends 2020]. As Figure 1 shows, data processing pipeline in a typical BDS aims at extracting meaningful insights from Big Data through several stages and supporting functionalities, where data mainly flow from left to right [Hu et al. 2014]. Each stage is a collection of similar functionalities. In addition, these stages are not necessarily mandatory and sequential. For example, to satisfy the heavy timeliness

This work was partly supported by the National Natural Science Foundation of China (No. 61672389) and Guangzhou Key Laboratory of Big Data and Intelligent Education (No. 201905010009).
Authors' addresses: A. Davoudian, Advanced Database Laboratory, School of Computer Science, 1125 Colonel By Dr., Carleton University, Ottawa, ON, Canada, K1S 5B6; email: alidavoudian@cmail.carleton.ca; M. Liu (corresponding author), Guangzhou Key Laboratory of Big Data and Intelligent Education, School of Computer Science, South China Normal University, 55 Zhongshan Avenue West, Guangzhou, Guangdong, China 510631; email: liumengchi@scnu.edu.cn.

0360-0300/2020/09-ART110 $15.00
https://doi.org/10.1145/3408314

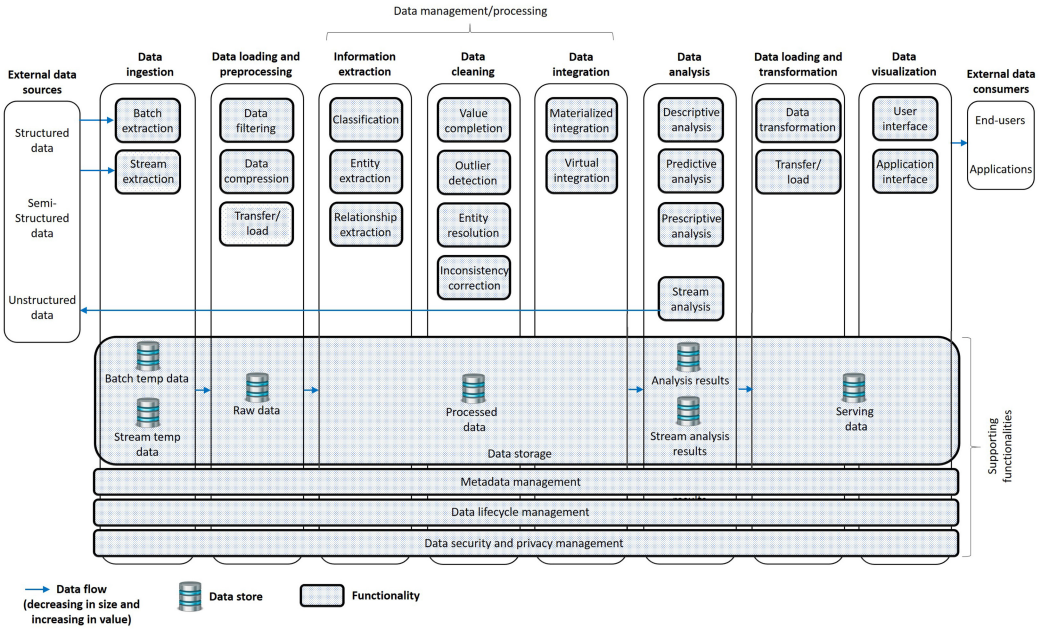

Fig. 1. Data processing pipeline in a typical BDS.

for stream processing, data cleansing can be left out, which means accepting more approximate results in exchange for low latency. We elaborate each stage in the following.

— *Data ingestion*, which extracts raw data either static (which do not move and require periodic synchronization) or streaming, from various data sources such as web servers and database management systems (DBMSs). The batch extraction functionality acquires static data (e.g., relational tables or Hadoop files) through some connectors such as massive data transfer protocols (e.g., Hadoop's copyFromLocal[1] or FTP), drivers for specific protocols (e.g., Apache Sqoop[2]), and APIs provided by the source application and web-crawlers. The collected static data may be temporarily stored in the Batch temp data store. The stream extraction functionality usually acquires streaming data (e.g., log data) by subscribing to a streaming API[3] (e.g., tweets in JSON format that are collected from Twitter Streaming API[4]). The collected streaming data may also be temporarily stored in the Stream temp data store such as Apache Flume, Apache Kafka, and Amazon Kinesis.

— *Data loading and preprocessing*, where parts of the extracted, unprocessed data that are either not suitable for further processing or not trustworthy might be filtered out by the *data filtering* functionality. This can help to decrease either the amount of data stored or the total uncertainty of the data. Compressing data, before transferring and loading activities, can also improve efficiency. Note that either the compression or filtering of streaming data is considered as stream processing. The above preprocessed data may be transferred and loaded from the temporary data stores into the Raw data store for further processing and

[1] https://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-common/FileSystemShell.html#copyFromLocal.
[2] http://sqoop.apache.org.
[3] It pushes data to the client, when they are available. This API significantly reduces the network latency by maintaining a persistent connection, whereby the client does not need to pull newer data by sending frequent requests to the server.
[4] https://developer.twitter.com/en/docs.

analysis. For example, Hadoop Distributed File System (HDFS) for the storage of batch data, as well as Apache Cassandra and Apache HBase for the storage of streaming data [Costa and Santos 2016; Santos et al. 2017].

— *Information extraction*, which aims at imposing a structured format on raw data, that is suitable for analysis. This includes methods for parsing and extracting information from semi-structured data (e.g., machine readable HTML or XML) [Sleiman and Corchuelo 2012], as well as more complex methods, such as natural language processing, text analytics, and ontology learning [Buitelaar et al. 2005] for classification and entity/relationship extraction over unstructured data [Balke 2012]. The classification component classifies unstructured data in various dimensions such as the extraction of a topic. The entity extraction component identifies real-world entities and categorizes them (e.g., as products or persons). The relationship extraction component extracts facts or relationships (e.g., represented by RDF triples) among identified entities. This is usually limited to a given context where possible entity types are known in advance and ontologies are used to specify possible relationships between entity types.

— *Data cleaning*, which refers to the handling of quality issues originated from a single dataset [Rahm and Do 2000]. This is the task of value completion, inconsistency correction, outlier detection and entity resolution components [Bauer and Günzel 2013]. The value completion component uses statistical and Machine Learning (ML) techniques along with the derivations of values from other attributes to fill incomplete and empty attributes' values. The inconsistency correction component prevents the violation of either dependencies between entities or the constraints on their attributes. The outlier detection component exploits distance-based, statistical model-based, ML-based, and context-aware techniques for the identification and correction of abnormal data. The entity resolution component identifies unique entities and merges the identical ones [Papadakis et al. 2020]. This resolution is easy when identical entities have the same value for the corresponding keys (or even other attributes). Otherwise, it needs to use entity matching techniques, whereby either identified entities with different names (or synonyms) are merged, or different entities with the same name (or homonyms) are distinguished [Fan et al. 2009].

— *Big data integration*, which aims at providing a unified view and query interface over heterogeneous datasets from many autonomous data sources [Doan et al. 2012; Dong and Srivastava 2015]. Data integration frameworks are mainly classified into materialized and virtual integration as the following:

- *Materialized integration*, which mainly refers to Data Warehouse (DW), physically integrates the extracted contents of OnLine Transaction Processing (OLTP) data sources using a common format, usually the relational model. Here, integration means defining a target database scheme together with a set of Extract-Transform-Load (ETL) jobs (or procedural mappings) that regularly collect data and populate the target DW. This leads to efficient query processing and physical independence among the DW and the sources of data; however, there are some issues such as out-of-date results, synchronization delay, costly evolution to adapt to changes in the business environment and the complexity of ETL development to deal with cleaning and transforming data as well as managing data refreshment.

  To transform and analyze massive amounts of heterogeneous data more quickly, the database community is making a transition from classical DW to the novel concept of Data Lake (DL) [Abadi et al. 2020]. Despite DWs that follow an approach of schema-on-write with a relational view of data, DL solutions adopt a schema-on-read approach [Suriarachchi and Plale 2016; Terrizzano et al. 2015]. In this regard, a DL collects the

ingested contents of heterogeneous data sources, in their native formats, in a common repository. This repository is a massive storage system that is mostly based on a distributed file system (e.g., HDFS); where data are processed in parallel, typically using the MapReduce model. DLs can be used for OnLine Analytical Processing (OLAP) and business analytics (similarly to DWs), as well as batch and real-time data analytics using Big Data technologies.

Unlike the extreme flexibility of current practical DL systems, there is an unknown semantics (or metadata) of the stored data, which prevents an efficient integrated query processing. This entails providing a functionality for the management of metadata (e.g., schema information) and a uniform query interface in some DL solutions [Hai et al. 2016; Quix et al. 2016; Walker and Alrehamy 2015]. This functionality should be able to extract implicit and explicit metadata from the imported data and the corresponding sources into a unified metamodel.

- *Virtual integration*, which can be regarded as an inheritor of the well-known mediated [Wiederhold 1992] or federated frameworks [Elmagarmid et al. 1999], defines a global (or mediated) schema providing a view over heterogeneous data sources [Langegger et al. 2008]. Here, queries expressed on the global schema are automatically translated to queries that are sent to and performed in the underlying sources of data. It is followed by collecting the results and assembling them into the answers of queries. The integration task here is to define declarative mappings to make relationships among the global schema and the underlying data sources. These mappings are performed through the definition of views based on two main paradigms: (1) Global-As-view (GAV) and (2) Local-As-View (LAV); that directly dictate how the queries are handled [Lenzerini 2002]. In GAV, the elements of the global level are defined as queries over source schemas. This provides simplicity in answering user queries as they just need to be unfolded to the sources. However, the mappings are invalidated by changes in the source schemata. In contrast, in LAV, local concepts are defined as queries over the global schema. This suites dynamic environments where source schemata change a lot. However, answering queries is a computationally complex task as they need to go through a more complex process of rewriting.

  A well-known method of virtual integration is Ontology-Based Data Access (OBDA) [Poggi et al. 2008], which follows the GAV paradigm. This is based on decoupling of intensional data in the sources and extensional data (i.e., schema) in an ontology. The most notable approaches to OBDA exploit generic reasoning in Description Logics (DLs) for query rewriting. In this regard, an OWL2 QL ontology, that is built upon DL-Lite family of DLs, is used to encode the global schema.

  The virtual integration of data results in no cost for freshness, synchronization and space. Furthermore, there are growing numbers of scenarios where the physical transfer of data is either illegal (due to the privacy of data) or prohibitive (due to the hidden data behind APIs). However, there are some issues such as showing poor performance of queries, figuring out the structure and quality of data sources, dynamic modification of a global schema by changing the availability of underlying sources of data, query optimization with respect to the global schema and being impacted by the unavailability of data sources. This architecture typically exploits the caching and reusing of query results to address the poor performance of queries [Gessert et al. 2017; Zakhary et al. 2017].

The techniques of conventional data integration aim to integrate a small number (in the order of tens or less) of structured data sources where metadata (e.g., views or schemata) for data integration are provided [Halevy et al. 2006]. However, *Big Data integration* have

brought new challenges as integrating a huge number of diverse (even unstructured) sources of data and building a global schema might be very hard or even unattainable [Bleifuß et al. 2018; Golshan et al. 2017]. The huge number of data sources can be tackled by integrating a smaller number of *related* ones. This entails building a searching system over the collection of data sources, which provides services such as locating all sources that are related to a specific query, providing useful information about each data source, finding similar data sources, making an integrated global view of all related data sources and monitoring all data sources [Golshan et al. 2017].

— *Data analysis*, which contains all functionalities that aim at deriving meaning and insights from data. Bertolucci [2013] classified data analysis into the following three levels, based on the analysis depth:
  - *Descriptive analysis*, where the demographics of the observed data are portrayed via techniques such as reporting and OLAP. In other words, it answers the questions: "What happened?" and "Why did it happen?"[5] Common examples include reports, querying, data visualization, and dashboards.
  - *Predictive analysis*, where the probability of an event to be happened in the future is forecasted. It exploits data mining, statistical analysis, ML and probabilistic models over observed data to detect patterns and recognize relationships in data. In other words, it answers the question, "What might happen?" in the future. A common example is customer behavior forecast.
  - *Prescriptive analysis*, which is closely coupled with optimization, uses high level modeling tools to improve decision making by predicting the possible consequence of future actions before they are taken. In other words, it answers the question, "What should I do?"A common example is identifying optimal solutions for resource allocation [IBM 2017]. Although the majority of efforts on data analysis is currently focused on descriptive and predictive analysis, prescriptive analysis is increasingly gathering research interest according to Gartner Inc. [Sapp et al. 2018]. A survey [Lepenioti et al. 2019] investigates state-of-the-art literature on prescriptive analysis, existing challenges and future directions.

  The stream analysis component analyzes data while they are flowing and accordingly reacts to the results (e.g., sending some alerts to users). The analytical results can be saved back to the initial data stores (to be seen by their users) or stored in a distinct Stream analysis results store. As streaming data are always incomplete and unknown before execution, their analysis (via techniques such as sketching and approximation [Babcock et al. 2002]) is different from the deep analysis of static data. However, the stream analysis task can be accelerated by exploiting partial results, models and rules precalculated by the analysis of static data.
— *Data loading and transformation*, which transform (i.e., cube generation) and load the results of the data analysis stage into the Serving data store.
— *Data visualization*, which is typically the access point (via the user and application interfaces) to the data stored in the Serving data store . The visualization of data can be performed via reports, dashboards, graphs, end-user applications (e.g., mobile applications), and so on.
— *Data storage*, which includes all data stores along the data processing pipelines, either long-term or temporary. Note that in Figure 1, each data store is shown along with the corresponding stage(s).

[5]This is related to diagnostics analysis, which is an extension of descriptive analysis, where the root causes of detected issues are investigated.

— *Metadata management*, which refers to the extraction and storage of metadata (i.e., data describing other data or systems) that are necessary for data management activities [Ceravolo et al. 2018]. Such metadata may describe (I) the structure of stored data, (II) the processing steps that either were conducted or still need to follow, (III) the provenance of each data item (when the extraction happened and from which source), and (IV) data status such as archived, purged, or active. Metadata extraction is performed by taking into account all data sources and storage areas within BDS (e.g., using the XML Metadata Interchange (XMI) format and extracting schema definitions form a relational database). Metadata about data sources (e.g., timeliness and completeness) might be manually prepared. This is discussed further in Michael and Miller [2013] and Soares [2013]. Metadata storage is simply providing a centralized repository dedicated to storing all metadata. This also requires establishing a metamodel of metadata (e.g., the Dublin Core standard [Miller and Mork 2013]).
— *Data lifecycle management*, which consists of data creation and discard activities throughout its lifecycle. This is performed by using rule-based methods, such as data compression, discarding, and archiving of stale data [Soares 2012].
— *Data security and privacy management*, which ensures protecting information against denial-of-service attacks and unauthorized access/modification of data by means of authentication and authorization, data anonymization, and access tracking strategies. Authorization and authentication require providing identification traditionally through usernames and passwords (assigned to each authorized user) and can limit the ability of the user for searching and extracting data. Access tracking, which is based on authorization and authentication necessitates logging information such as who request access to which datasets, how data are manipulated during each access, and whether the access request is valid or not. Data anonymization is the process of hiding identity or sensitive data from original data before getting presented to end-users. Note that cross-references that are emerged after merging several data source may deanonymize data[6] [Machanavajjhala and Reiter 2012].

The illustrated complexity of BDSs, which is due to the intrinsic characteristics of Big Data, has challenged the well-established development process of traditional, structured data systems [Chen et al. 2015; Gorton and Klein 2015]. However, as indicated in economic reports [Davenport 2019; Nadkarni 2020; Sharala 2019] and scientific literature [Kumar and Alencar 2016; Laigner et al. 2018], this complexity requires using novel or revised techniques of Software Engineering (SE)[7] in the development of BDSs. This is reflected in recent studies that state that many organizations are failing to derive business value from their data. After surveying 300 organizations, Kelly and Kaskade [Infochimps 2013] reported that 55% of BDSs had not been completed and many others had fallen short of their objectives. In addition, many companies are still stuck in the pilot phase and only 15% successfully developed and deployed their BDSs, according to a Gartner report [Van Der Meulen 2016]. Later, an Infoworld article [Patrizio 2019] noted that nothing has changed and failure rates are still high.

In fact, the rapid growth of BDSs is generating a SE paradigm shift from traditional, functionality-driven software development style toward modern, data-driven style. More intuitively, beyond functionality, the achievement and added value of developed systems is tightly coupled with the proper identification of data (and the corresponding quality) requirements with respect to Big Data characteristics. Subsequently, satisfaction of the extracted requirements entails

[6]Including all the aforementioned functionalities in a BDS may overburden the overall performance of system. For example, Interlandi et al. [2015] show a 30% overhead on performance by using automatic data provenance.
[7]It refers to the "*systematic application of scientific and technological knowledge, methods and experience to the design, implementation, testing and documentation of software systems*" [ISO 2017].

the selection and orchestration of (rapidly evolving) Big Data technologies and frameworks, which in turn pose an enormous challenge for software architects. They should ensure that developed BDSs can scale and evolve to meet long-term requirements. However, benchmarking or testing in the context of BDSs requires huge amounts of varying test datasets, where there is often a lack of known set of inputs and expected outputs.

Given such a data-centric environment, how the requirements should be specified for an end-user system utilizing Big Data? How the system should be structured, or which reference architecture should be selected for processing a voluminous amount of structured and unstructured data? Similarly, how the software/data quality assurance should be done differently? This situation has led to some existing research challenges (and also opportunities) that need to be tackled by the Software Engineering and Computer Science (CS) research areas to create new or improved BDSs within budget on time. However, the available literature mostly focuses on technical aspects of Big Data or specific platforms; and there is no comprehensive survey to the best of our knowledge on existing SE methodologies and tools for the development of BDSs. Hence, we take a software engineering perspective and investigate the existing research and industry efforts in three sub-disciplines of software engineering, i.e., requirements engineering, architecture design and quality assurance in the context of BDSs.

The rest of this survey is organized into five sections. In Sections 2, we present state-of-the-art of Requirements Engineering (RE) research in the context of BDSs. Section 3 introduces some architectural requirements regarding Big Data characteristics and reflects on state-of-the-art BDS reference architectures evaluated by the requirements. In Section 4, we present state of the research and hot topics in the area of ensuring BDS quality. Section 5 concludes the article and presents the exiting challenges of developing BDSs with respect to the three sub-disciplines of software engineering.

2 BDS REQUIREMENTS ENGINEERING

The term *software requirement* refers to either a function that must be performed by a system or system element (a.k.a. functional requirement) or a required constraint on the functional effects of the software (a.k.a. non-functional or quality requirement). Identification of requirements is one of the first essential steps in IT projects. Requirements Engineering (RE) refers the use of systematic and repeatable methods to ensure the completeness, relevance and consistency of requirements [Sommerville and Sawyer 1997]. It consists of such key activities as the elicitation of use-case scenarios, the analysis and specification of requirements, the validation of requirements, the prioritization among the requirements for implementation and the management of requirements throughout the system's lifecycle. These activities build a bridge among the actual needs of stakeholders (e.g., users, customers, developers and businesses) and the offered capabilities of the target software.

Although RE has long been recognized as a mature discipline in Software Development Life Cycle (SDLC), it is currently passive in the development of BDSs. In the following, we elaborate some major RE challenges in the context of BDSs.

— *Integrating Big Data characteristics in requirements.* As a cross-functional discipline, RE supports all SDLC phases. Accordingly, it supports the design of BDSs by facilitating the selection of Big Data technologies and frameworks. Currently, as such technologies are being developed and introduced into the market at a very rapid rate, they should be matched with the requirements of BDSs, by taking into account some engineering tradeoffs. This entails to properly incorporate Big Data characteristics in the definition, analysis, and specification of requirements, which in turn leads to an accurate selection of those technologies and

frameworks [Arruda 2018; Madhavji et al. 2015]. For instance, we can refer to the rationale behind selecting Cassandra (from a wide range of NoSQL stores [Cattell 2011; Davoudian et al. 2018]) in project EPIC,[8] to store a huge amount of tweets (Volume) with evolving schema (Variability). Cassandra follows a wide-column data model, whereby data are automatically partitioned, both horizontally and vertically, across the cluster (i.e., meeting Volume). In addition, its flexible schema allows the storage of JSON objects (i.e., meeting Variability). As another example, a requirements analyst may need to effectively use a newly arrived customer's video footage (Velocity) in a shopping store along with possible prior experiences (Volume) in the store. Hence, time-sensitive and personalized discounts are given in real time to the customer at particular display points along the path where s/he might buy products.

— *New requirements might be emerged from new data.* Existing RE strategies focus primarily on early object-oriented analysis of finite datasets with a known structure. They are therefore not prepared to handle situations where new analysis opportunities might be facilitated by new data and new desired outcomes (not known when analyzing the requirements). In other words, data observations can refine current requirements or create new ones. This means paving the way for some functionalities to be realized differently or even fresh functionalities. This also makes it necessary to exploit highly iterative and agile development approaches (as in Chen et al. [2015]), whereby the application domain of the system and user needs[9] are evolutionary recognized.

— *Linking requirements to business goals.* Software requirements, as a significant way of communication in IT projects, are strongly related to project management, business modeling, technical system design, and architectural resources [Hiisilä et al. 2016]. This requires linking the RE activities to business goals. Furthermore, while not all business goals relate to the system's quality requirements, researchers claim that such requirements should be generated from business goals and concerns [Clements and Bass 2010]. Accordingly, BDS developers must address the strict connection between software requirements and business goals.

— *Complexity.* Due to the aforementioned complexity of BDS, where several dynamic components interact in a distributed environment, traditional RE techniques, tools, templates, and gathering artefacts cannot be applied very well [Madhavji et al. 2015].

In this section, the state-of-the-art of RE research are investigated in the context of BDSs (see Table 1), where requirements engineers and business analysts need to know about the following relevant concepts.

— *Big Data scenarios*, which describe a specific use of the system with regard to Big Data characteristics. These scenarios, which are identified from business goals,[10] are used to derive Data-specific requirements, quality requirements, and constraints.[11]

[8] It is a research project investigating how people use social media in times of crisis [Palen et al. 2010].

[9] For example, Do the users require interactive data access or batch-generated reports?, Is the information presented by the reports on the entire dataset or on time-windows within the dataset?, Is it required to have 24/7 data collection?; Are the sources of data bursty or steady?, How many sources of data are there?, and Are the results of analysis stored?

[10] They describe what an organization expects to achieve over a particular period of time. They are linked to the organization's needs rather than customers' needs [Clements and Bass 2010].

[11] They restrict or dictate the project team's actions such as schedule, budget, quality, scope, resources, and software license restrictions.

Table 1. State-of-the-Art of RE Research in the Context of BDSs

Paper	RE activity	Focused requirement category	Contribution	Empirically validated
[Lau et al. 2014]	Elicitation	Functional	A conceptual architecture for requirements elicitation	Yes
[Chen et al. 2016, 2015]	Analysis	Quality	A method for BDS development	Yes
[Jutla et al. 2013]	Analysis	Quality	UML extension for privacy requirements analysis	No
[Sachdeva and Chung 2017]	Analysis	Quality	An approach for handling quality requirements for BDSs in Scrum	Yes
[Noorwali et al. 2016]	Specification	Quality	An approach for integrating Big Data characteristics & quality requirements	No
[Arruda and Madhavji 2019]	Specification	Quality	A tool for modeling quality requirements	No
[Al-Najran and Dahanayake 2015]	Specification	Data specific	A requirement specification framework for data collection	Yes
[Narayanan 2016]	Specification	Data specific	Two templates to specify data requirements	No
[Otero and Peter 2014]	Specification	Functional, concept-drift -aware	State-of-the-art	No
[Madhavji et al. 2015]	Specification	Functional	State-of-the-art	No
[Eridaputra et al. 2014]	Modeling	All	A requirement specification model based on I* & KAOS tools	Yes
[Arruda and Madhavji 2017]	Modeling	Quality	An RE artifact model	No
[Bersani et al. 2016]	Validation	Not specified	A requirement validation tool	Yes
[NIST 2018]	Not specified	All	NIST interoperability framework	No

—*Data-specific requirements*, which can be categorized as follows:

- *Data capability requirements*, which address the network and storage requirements. For example, system needs to support PostrgeSQL and MongoDB in a high-speed InfiniBand network.
- *Data source requirements*, which refer to the different characteristics of data sources, such as file formats, data size, growth rate, and being static or streaming, which in turn address the BDS data ingestion stage. For example, system must collect data from sensors and cloud data sources.
- *Data transformation requirements*, which address the BDS data processing and analysis stages. For example, system must support batch and rand data analysis.
- *Data consumer requirements*, which address the BDS visualization stage. For example, system must support processed results in text and table formats.
- *Data lifecycle management requirements*, which address the BDS data lifecycle management functionality. For example, system must support data quality curation consisting of format transformation, data reduction, classification, and clustering.

Lau et al. [2014] propose a model-driven methodology of requirements elicitation (deployed within a research project named Dicode[12]) in the context of BDSs. It includes three steps: first, eliciting requirement from scenarios; second, exploiting (individual and collaborative) sense-making

[12] http://dicode-project.cti.gr/site/.

models (e.g., iterative cognitive process conducted by people to create a representation of a knowledge space that is useful for achieving a goal); and, third, developing a conceptual generic architecture for analytics of Big Data to bring user and technology perspectives together. Sense-making models have been applied to understand Big Data analytics' cognitive complexity where components that exploit human and machine intelligence are included. Two instantiates (biomedical research and social media domains) of the proposed architecture are also presented by the authors.

Chen et al. [2016, 2015] propose a methodological framework for BDS development, whereby data modeling (conceptual, logical and physical) techniques, architecture analysis as well as technology selection are integrated in SDLC. Although this approach is specific for system design, it implements an RE phase for the analysis of requirements that consists of the following activities: first, business goals are captured; second, concerns, constraints, and drivers of the design are identified; third, quality attribute scenarios are defined; and fourth, based on the quality attributes scenarios, Big Data scenarios are identified. This method suggests Big Data templates to log data architecture elements, such as data volume, variety and velocity, data quality, frequency of read/write and time-to-live as well as queries, captured for each source of data. The obtained requirements affect the subsequent selection of architecture, data model, technologies, and data access pattern.

Agile methodologies, such as Scrum and Extreme Programming (XP), allow the rapid delivery of a working software that satisfies functional requirements and adapts to customer feedback and requirements changes. However, to reduce time to market, it is either probable that quality requirements will be ignored or introduced late, hence resulting in the failure of software projects. Accordingly, Sachdeva and Chung [2017] introduce a method to deal with the Performance and Security requirements of BDSs residing on the cloud and developed through the Scrum agile framework. This method treats Performance as spikes and acceptance criteria of user stories and Security as system functionalities (or collection of user stories) introduced at the start of the process of software development. The authors also present an industrial case-study in Scrum, where the quality requirements' issues are mitigated using the proposed method. Note that agile professionals have not yet agreed on the meaning and the way to handle quality requirements.

Jutla et al. [2013] extend UML use-case diagrams with components of Privacy to assist software engineers speed up the analysis phase of Privacy requirements in the context of BDSs. This is implemented as the UML extension ribbon of MS Visio that is automatically loaded in Visual Studio, whereby Big Data privacy services (e.g., anonymization) can be easily dragged and dropped by software engineers into their UML diagrams. In a health sector application, the author also demonstrates the usefulness of the extension by using an IBM Watson-like commercial use case on Big Data.

By taking into account the issues presented by the characteristics of Big Data on quality requirements, a specification approach is proposed by Noorwali et al. [2016], whereby the 4-V characteristics of Big Data (i.e., Volume, Variety, Velocity, and Veracity) are integrated with the specification of traditional quality requirements (e.g., Availability, Performance, Security, Scalability, and Privacy) in a unified requirement description. These integrated specifications would permit various permutations of Big Data characteristics along with quality requirements (see Table 2). The full requirement description must specify the desired permutations. This approach necessitates a requirement analyst (in the context of BDSs) to know about Big Data reference architectures, tools, libraries and technologies to address Big Data characteristics. According to this work, [Arruda and Madhavji 2019] introduce QualiBD, a tool for modeling the quality requirements of BDSs. However, the tool has not yet been empirically evaluated at the time of writing.

Al-Najran and Dahanayake [2015] aim to enhance data ingestion process via reducing irrelevant collected data. They provide a new requirements specification framework that identifies data

Table 2. Permutation Examples of Big Data Characteristics along with Quality Attributes

Characteristic of Big Data × Quality attribute	Quality requirement description	Rationale
Velocity × Performance	Real-time data generated by global earthquake sensors shall be processed by Apache Storm, Samza, or S4, with a latency of 0.5 – 1.5 seconds.	To meet Performance requirements, high Velocity streaming data need specialized processing engines, such as Apache Storm, Samza, or S4, to be routed, transformed and analyzed.
Variety × Security	The Security of structured, semi-structured and unstructured data, shall be ensured by exploiting VCAM, IPAC and FIM methods respectively.	Data Variety incurs exploiting different access control methods, such as VCAM, IPAC, or FIM, to ensure Security.
Veracity × Security × Performance	Using the CMD, or FHE method, 50K tweets shall be encrypted, queried and decrypted in 2.0 seconds.	Veracity is ensured by exploiting computationally inexpensive Security methods such as CMD or FHE.
Variety × Availability	N.R. (domain dependent)	Variety does not affect the system Availability.

collection scenarios through a backward analysis process, whereby the properties of input are defined based on the properties and context of the output. Elicited scenarios are characterized with respect to their domain, spatial and temporal factor, search patterns, such as named entities, phrases and keywords, as well as capturing and analyzing techniques. Accordingly, scenario-relevant data are captured from the sources that meet the scenarios. This framework has been empirically tested using quantitative experiments for measuring the relevance of Twitter feeds [Al-Najran 2015].

Otero and Peter [2014] refer to a critical challenge in specifying testable or verifiable requirements in the context of BDSs, as the predictive analysis of the incoming data is subject to concept drift. It refers to the unexpected changes overtime in statistical properties of the target variable, which is supposed to be predicted by the model such as an ML algorithm, and results in less accurate predictions.

Eridaputra et al. [2014] introduce a new requirement specification model for BDSs through the following three steps. First, eliciting generic requirements for BDSs (by taking into consideration the 4-V characteristics of Big Data) including (I) huge capacity of database, (II) fine performance of database, (III) structure and quality of data, and (IV) guaranteed privacy and security of data. Second, modeling the elicited requirements through the i* and KAOS tools as parts of Goal Oriented Requirement Engineering (GORE) method. Third, using the models extracted from the tools as references for the modeling of both functional and quality requirements. According to the authors, after applying the generic model to a BDS for a government agency, 10 non-functional and 26 functional requirements were obtained whose accuracy was further validated by stakeholders.

Bersani et al. [2016] propose the DICE Verification Tool (D-VerT), which allows verifying safety properties (e.g., whether the system topology reaches an undesired configuration) of a topology-based BDS. A topology is an abstract representation of the BDS, with two kinds of nodes: data sources and operators implementing the logic of system (e.g., Filter, Aggregate and Join). The verification is carried out on annotated UML models containing all the required data related to the system topology. The tool supports both the checking of bounded reachability and satisfiability checking, which are two distinct kinds of verification built on top of logical formalism. Through the checking of bounded satisfiability, an input topology property is checked for executions that violate the property. Through the reachability checking approach, an array-based system is used

to define the input topology, followed by verifying the system against a safety problem. This approach, whose result is either safe or unsafe, leverages an initial configuration, a formula defining the set of unsafe states and a set of system transitions.

Narayanan [2016] proposes two templates that can be used for supporting data-specific requirements: (I) a template for data acquisition and (II) a template for matching data to business problems. NIST Group [NIST 2018] provides a consensus list of generalized Big Data requirements extracted from 51 use cases of nine diversified BDS contexts. The requirements are divided into seven individual groups, namely data source, data consumer, capabilities, privacy and security, data transformation, lifecycle management, and other requirements. NIST Big Data Reference Architecture is built using these requirements.

Madhavji et al. [2015] introduce a new SE context model for BDSs, where various elements such as business and client scenarios as well as corporate decision-making are taken into account. Subsequently, as this model lacks RE process design, Arruda and Madhavji [2017] propose the creation of an RE artefact model, named BD-REAM, for BDSs. Based on this initial study, Arruda [2018] has developed a SE context model for BDSs. At the time of writing, this research is still in early stages of development.

According to the related works of RE, there is a lack of a requirements modeling language for BDSs. The formal definition of such a language should allow the verification of generated models by a corresponding tool. Furthermore, it is suggested to work on the automatic generation of functional and quality requirements from the corresponding textual description. However, in some works there is no empirical evaluation for the validation of the corresponding proposal. Therefore, it is important to carry out more empirical studies in industry to get a better understanding of RE activities in the development of BDSs. The definition of some functional requirements of a BDS depends on the right specification of data requirements such as the need for various output file formats for rendering, reporting and visualization, as well as advanced distributed data storage [NIST 2018]. However, only two works in our investigation [Al-Najran and Dahanayake 2015; Narayanan 2016] discuss the data properties (e.g., file formats, data types, data size, at rest or in motion and rate of growth) as well as the necessity of the selection of right type of data. Nonetheless, no concrete example of what a data requirement looks like is provided by the two papers.

As Table 1 depicts, most of the works address either the phase of requirements analysis or specification. Elicitation, modeling, and validation of requirements are discussed by only a few works [Arruda and Madhavji 2017; Bersani et al. 2016; Eridaputra et al. 2014; Lau et al. 2014]. In addition, no paper has been found addressing the phase of requirements negotiation, prioritization, or management. Overall, there is no significant amount of research on RE methods, processes, and tools. This is due to the fact that apart from Big Data characteristics, RE for BDSs has no difference at its core with RE for any type of commercial data processing and analytics system.

3 BDS ARCHITECTURE DESIGN

Big data systems are inherently distributed systems whose technologies and frameworks are being rapidly evolved. The architecture of such system is tightly coupled with data and deployment architectures [Gorton and Klein 2015]. Consequently, many challenges have arisen for BDS architects and designers. In the following, we elaborate some major design challenges.

— *Data architecture.* The distribution of data results in a fundamental design challenge defined by the CAP theorem over three quality attributes (availability, consistency, and partition tolerance). During a network Partition, a distributed data system must make a tradeoff between strong Consistency (i.e., every read request observes the most recent update) and Availability (i.e., every read/write request receives a successful response within a finite time). By

taking this theorem into account, NoSQL stores have relaxed many core principles of relational database systems to attain high performance and availability. As such, NoSQL stores adopt proprietary APIs, deliberately denormalized (and flexible) data models, and weak consistencies instead of the SQL standard, normalized (and schema-full) data models and the strong consistency, respectively.

Since in practice network partitions cannot be avoided by distributed systems, BDS architects need to focus on the tradeoff between (strong) consistency with availability. As a result, architects must diligently evaluate candidate database engines and select the ones that satisfy application requirements. This usually results in *polyglot persistence*—to solve a complicated problem by splitting it into fragments whose associated datasets are stored in various database technologies with different tradeoffs between the quality attribute. The results must then be integrated into a solution for hybrid data storage and analysis.

— *Deployment architecture.* As BDSs are inherently distributed systems, the architects of such systems have to explicitly deal with unpredictable communication latencies, partial failures, consistency, replication, and resource optimization (e.g., via elasticity and data compression). These challenges are also exacerbated at scale. In other words, the rapid growth of data (to and beyond petascale) has led to the growth of hardware resources to many thousands of processing nodes and disks and, possibly, the geographical distribution of data. This, in turn, increases the probability of failure in hardware components and requires a resilient software architecture whereby failures are gracefully handled with no interruption in the operations of applications. However, many commonly used software architectures are not resilient at this scale of deployment; where a resilient architecture preserves availability in case of disk failures or network partitions and keeps replicas consistent via multi-master or master-slave protocols. In addition, such architecture limits failures by providing replicated and stateless components that are also tolerant of dependent services' failures.

— *Architecture patterns and implementation stacks.* Despite using a proven technology stack by small data systems, there is a lack of well-established BDS architecture tactics and patterns.[13] In addition, there is a fast evolution and growth of (proprietary and open source) Big Data technologies that causes pressure on architects to learn and experiment with new technologies. As such, BDS developers are often have to (I) integrate frameworks in an ad hoc manner, (II) select the corresponding technologies, and (III) deploy them to scalable environments such as cloud computing.

— *Evolutionary and query-driven design.* In small data systems, where the typical Create, Retrieve, Update, and Delete (CRUD) operations are merely implemented, there are mature methods, such as Object/Relational (O/R) mapping and ordinary design solutions such as system's persistence layer. However, since BDSs usually approach technological boundaries, data handling has a great effect on their architecture design. Therefore, queries discovered after requirements analysis often drive the core of a BDS architecture. This requires novel SE methods that support the agile creation of query-driven architectures in a highly evolutionary manner.

In the early 1990s, OLAP was proposed by Codd [1993] to express a new type of queries that differ essentially from OnLine Transaction Processing (OLTP). By the end of 1990s, the reference architecture of DW—consisting of OLTP and OLAP databases that are linked by the pipelines of ETL—was widely known. However, there were two issues: First, developers realized the difficulty

[13] Although the community has been aiming for some time to collect Big Data patterns [Arcitura 2017], this research is still in a rather immature state and is often based on some anecdotal blog posts.

to build and maintain ETL pipelines [Boykin et al. 2014; Lee et al. 2012; Lin and Ryaboy 2013]; second, organizations had to use day-old data for decision making, owing to the delay introduced by ETL pipelines and using nightly analytical processing.

By accelerating the pace of business, organizations started to conduct business intelligence on fresher data. One solution was to increase the frequency of ETL (e.g., hourly); but it led to more stress on unsteady ETL pipelines and high probability of passing the breakpoints [Cuzzocrea et al. 2018; Mishne et al. 2013]. This shows the inability of traditional DWs, that are just optimized for bulk loading, to deal with real-time data[14] [Duggan et al. 2015]. This, in turn, has motivated to explore alternative architectures that support Hybrid Transactional/Analytical Processing (HTAP) integrating analytical and transactional processing. Accordingly, today's organizations aims at using BDS architectures that (tightly) integrate batch and real-time processing, enabling high performance and seamless querying over both historical and real-time data. In the following, we introduce some BDS architectural requirements regarding the characteristics of Big Data. We also reflect on state-of-the-art BDS architectures evaluated by the requirements. This facilitates the design of architecture and the selection of commercial solutions or technologies for the BDS development.

3.1 BDS Architectural Requirements

This subsection provides a list of requirements that a BDS architecture should fulfill. It takes into consideration the five "V's" of Big Data characteristics namely Volume, Velocity, Variety, Variability, and Veracity that cover Big Data's real nature. We left out the Value characteristic as in the tight collaboration between the IT and business sides (for developing a BDS), it falls on the side of business while the other characteristics mentioned above fall on the side of IT (see Table 3).

- **Volume** refers the ever-growing amount of data (up to 180 ZB by 2025). This requires providing a scalable storage and processing of massive datasets (**R1.1**). This is usually tackled by the distribution and parallel processing of data using cloud-based Big Data technologies. However, despite the natural adaptation between descriptive analysis (**R1.2**) and distributed data management solutions, it is still challenging to fit predictive and prescriptive analysis (**R1.3**) into such distributed solutions [Tsai et al. 2015]. Note that in small data systems, data scientists generally export datasets to specialized software (e.g., SAS[15] or R[16]) for running statistical methods outside the system [Ordonez 2010]. However, it is not feasible in BDSs due to the Volume characteristic. This requires to rethink about techniques of predictive and prescriptive analytics to run in the parallel and distributed infrastructure by taking into account the principle of data locality [Özsu and Valduriez 2011].
- **Velocity** refers to a high speed at which the data are produced, ingested, managed, analyzed and served for meaningful decision making. This results in two major challenges. First, providing mechanisms for the ingestion of streaming data (that can be based on a buffering model of sliding window whereby the irregularities of data are made smooth) (**R2.1**). Second, achieving a (near) real-time processing of streaming data by providing linear or sublinear algorithms (**R2.2**). This requires providing a storage for the management of runtime data.
- **Variety** refers to the heterogeneity of ingested data in unstructured (e.g., plain text and video), semi-structured (e.g., JSON and XML based), and structured (e.g., relational tables)

[14] Despite traditional DWs, Big DWs are a subset of BDSs. These modern DWs have encountered such progressive research that requires a separate survey on their underlying features.
[15] www.sas.com.
[16] https://cran.r-project.org/.

Table 3. Main Requirements for a BDS Architecture

Requirements	
R1.	**Volume**
R1.1	– A scalable storage and processing of massive datasets is provided.
R1.2	– Descriptive analysis is provided.
R1.3	– Predictive/prescriptive analysis is provided.
R2.	**Velocity**
R2.1	– Streaming data are extracted.
R2.2	– Streaming data are processed in a (near) real-time manner.
R3.	**Variety**
R3.1	– Heterogeneous data are ingested.
R3.2	– A machine-readable schema of the entire data is provided.
R3.3	– Semantic data interoperability conflicts are resolved.
R4.	**Variability**
R4.1	– Adaptation mechanisms for schema evolution are provided.
R4.2	– Adaptation mechanisms for data evolution are provided.
R4.3	– Adaptation mechanisms for the automatic inclusion of new data sources are provided.
R5.	**Veracity**
R5.1	– Mechanisms for data provenance are provided.
R5.2	– Mechanisms for the assessment of data quality are provided.
R5.3	– Mechanisms for tracing data liveliness are provided.
R5.4	– Mechanisms for data cleaning are provided.

formats (**R3.1**). Providing an efficient data analysis requires a query engine to understand what is exactly stored (**R3.2**) and to resolve semantic data interoperability conflicts [Pagano et al. 2013; Vidal et al. 2019] (**R3.3**), which entail the management of rich and appropriate metadata [Ceravolo et al. 2018]. The former (**R3.2**) can be tackled through the previously mentioned materialized (e.g., rule-based data transformation) or virtual (e.g., OBDA) methods of data integration, whereby schematic data conflicts (i.e., related to different schemata) are resolved. However, the latter (**R3.3**) requires an effective integration of data to resolve other data interoperability conflicts: (1) domain conflicts that are related to the different interpretations of the same domain, including homonyms, synonyms, acronyms, and integrity constraints; (2) granularity conflicts that are related the different units of measurement and aggregation of data; and (3) completeness conflicts that are related to different pieces of data belonging to the same entity.

— **Variability** refers to the evolutionary nature of ingested data [Duncan 2014]. Despite existing a lot of mechanisms to handle schema evolution[17] (**R4.1**) and data evolution[18] (**R4.2**) in the relational model [Curino et al. 2013], achieving so in the context of BDSs is more challenging due to the schema-flexible nature of NoSQL stores [Lu and Holubová 2019; Meurice and Cleve 2017]. In addition, data sources may evolve during the lifecycle of a BDS (e.g., due to an outage is a sensor grid, or adding a new social network). This requires handling the evolution of data sources (**R4.3**).

[17]It refers to continuous modifications in the structure or schema of ingested data, which incurs the continuous adaptation of storage strategies, indices, queries, and data instances to such changes.
[18]It refers to the transformation of data with respect to the existing schema.

— **Veracity** refers to the quality and trustworthiness of historical or even real-time streaming data. It is accomplished through automated data governance frameworks consisting of the following components [Alhassan et al. 2016; Khatri and Brown 2010]:
  - Provenance of data (**R5.1**), which means tracing the lineage (or history of performed transformation steps) of any data piece to the sources, whereby the corresponding computation is reproduced. [McClatchey et al. 2015]. This requires the storage of metadata for all transformations carried out in a common data model, such as the Open Provenance Model [Moreau et al. 2011], for further study and exchange.
  - The assessment of data quality (**R5.2**), whereby all data are tagged with quality characteristics such as timeliness, completeness and accuracy among others (see Subsection 4.2). This prevents using low quality data (e.g., those with missing values) and generating poor analysis outcome.
  - Liveliness of data (**R5.3**), which determines when data are used. This feature can be used in data lifecycle management (see Figure 1).
  - Cleaning of data (**R5.4**), which exploits a set of strategies, such as deduplication, to enhance the quality of data (see Figure 1).

3.2 BDS Software Reference Architectures

The aforementioned architectural requirements prove the inability of exploiting traditional Business Intelligence (BI) architectures (relying on relational databases) in the context of BDSs. Such systems exploit NoSQL stores [Cattell 2011; Davoudian et al. 2018] to store loosely structured data. NoSQL stores, however, cannot meet the respective requirements as stated for Variability and Veracity. BDSs are presently being developed using architectural solutions that are ad hoc and complicated. Accordingly, BDS architects need to have an extremely high degree of expertise to select and orchestrate some software components among lots of available and overlapping ones according to the system requirements. This motivates exploiting some Software Reference Architectures (SRAs)[19] that facilitate the development of concrete architectures, as the software architect knows in advance the type of components and their corresponding interconnections. Hence, s/he is primarily responsible to design a concrete architecture through technology selection for those components with respect to the organizational needs and goals. In the following, we investigate six relevant SRAs exploited in the context of BDSs and discuss the strengths and limitations of these solutions.

3.2.1 The Lambda Architecture. Lambda tackles the *Volume* and the *Velocity* complexity dimensions of Big Data by complementing a high-throughput and high-accuracy batch processing component with a low-latency real-time processing one. It[20] is built on three major principles: data immutability, fault-tolerance, and recomputation [Marz 2011]. Immutability is assured by replacing the CRUD update and delete tasks with the append functionality whereby incoming data units are stored by timestamps and appended to an immutable, constantly growing Master dataset. This immutable data model has three advantages: (I) By taking into account the entire history, analytical applications can perform time-series analysis (e.g., a social media application can analyze old friendship relations that are not valid anymore); (II) fault tolerance, since lost or corrupted data (due to hardware failures and human errors) can be replaced by earlier data; and (III) the simplicity of the system, since there is no need for either indexing or locking data in the append-only Master dataset. Assuming a query is a function that computes its result over the whole Master dataset, a

[19] A reference architecture combines the "*general architecture knowledge and general experience with specific requirements for a coherent architectural solution for a specific problem domain.*"
[20] http://lambda-architecture.net/.

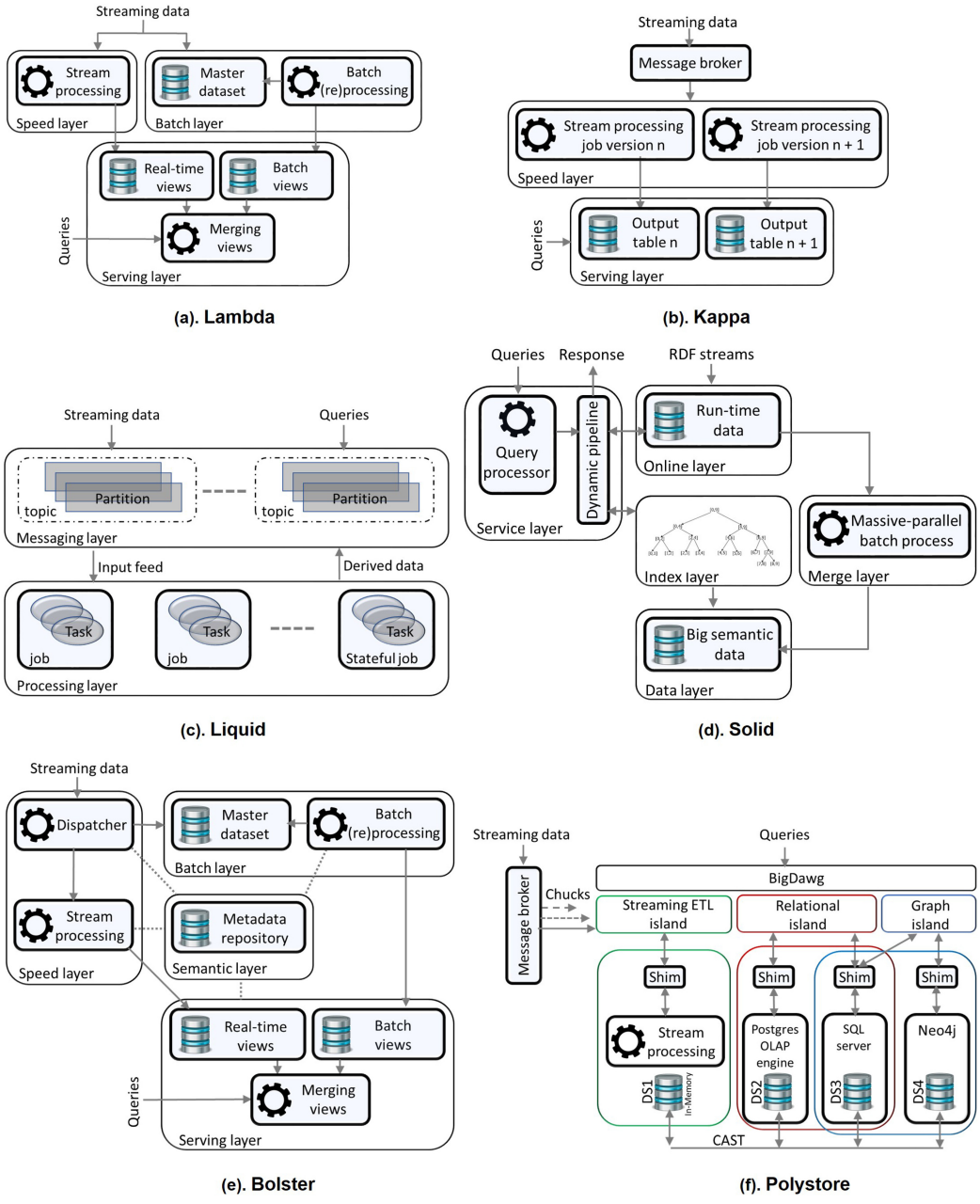

Fig. 2. Six representative SRAs in the context of BDSs.

recomputing algorithm is required to obtain precise results with respect to the evolved/fixed data or processing logic. Since recomputation comes at the cost of high latency, Lambda uses precomputed results (batch views) to respond with low latency [Marz and Warren 2015]. Figure 2(a) shows an overview of the Lambda architecture composed of a Batch layer, Speed (streaming) layer, and a Serving layer. As illustrated, incoming data split into two separate streams captured and fed into

the Speed and Batch layers in parallel. The Batch layer has essentially two functions: (I) It stores the Master dataset (e.g., via HDFS), and (II) it periodically recomputes batch views (e.g., a DW) via a batch processing framework (e.g., Apache Hadoop) to combine new data into the earlier existing views. This processing of historical data from the ground up also ensures fault tolerance. Batch views are written to a database at the Serving layer. This database is optimized for serving batch writes and random reads.

Since recomputation takes a long time, the latest generated batch views might not contain the latest data available to the system. This problem is tackled by the Speed layer, which processes only recent data. Unlike the Batch layer, the retrieved data are not written to a database, but they are forwarded directly to a stream processing framework (e.g., Apache Spark Streaming[21]) to incrementally compute real-time views. These views are written to a database at the Serving layer, which is optimized for serving random writes and random reads. Merging the batch and real-time views at query-time ensures a complete answer with regard to the latest information. Although the Lambda architecture allows the fast processing of requests, there are some drawbacks.

—Developing, deploying, and maintaining the same processing logic (evolving over time) for two layers augments the architecture complexity.
—Using two processing frameworks increases the hardware footprint. Both layers must be kept synchronous as by changing a particular view in one layer, the corresponding view must be adapted in the other layer as well.

  The above issues can be alleviated by using a unified ecosystem (e.g., Apache Spark[22]) to perform both batch and stream processing using the same code base. So the Speed layer can be implemented with minimal overhead via the corresponding API, such as Spark Streaming, which makes use of existing processing code and deployment. However, if the Batch and Speed layers use different ecosystems (e.g., Apache Hadoop and Storm,[23] respectively), then the processing code can be written by an abstract language (e.g., Summingbird [Boykin et al. 2014]) and automatically compiled for both the layers. However, developers should be able to manipulate complex programming abstractions. In addition, deployment and maintenance overhead still remain.
—Merging in the Serving layer involves a certain complexity. The data must be structured in a way that efficient merging is possible. Thus, designing the database schemes to be compatible with each other is essential. We can tackle this issue by using the same database (e.g., Apache Cassandra or Apache HBase) for storing both the real-time and batch views in the Serving layer.
—There is a redundant processing of data (in the Speed layer) for use-cases where low latency is not continuously needed. Kroß et al. [2015] tackle this issue by using the Batch layer exclusively, however, when batch processes are probable to exceed response time limitations, the Speed layer is switched on.
—Lambda is unable to set up an incremental processing job as it periodically recomputes batch views from the ground up. This increases the processing latency.
—There is no "tight" integration of the Batch and Speed layers, since there is no possibility to exchange intermediate results (e.g., data mining models) between the layers. However, this

[21] http://spark.apache.org/streaming/.
[22] https://spark.apache.org/.
[23] https://storm.apache.org/.

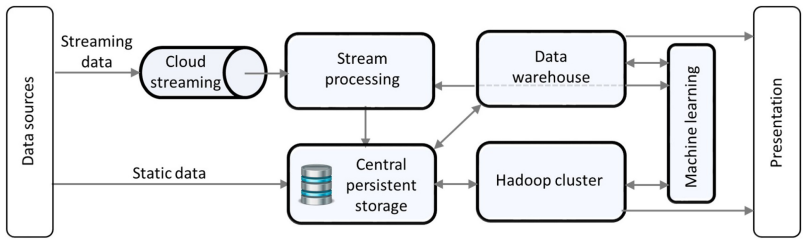

Fig. 3. The cloud-based Lambda architecture.

is required for a comprehensive analysis.[24] Giebler et al. [2018] tackle this issue by intertwining the Batch and Speed layers through communication channels among the layers.

Heilig and Voß [2017] combine the philosophy of the Lambda architecture with the cloud platform. This amalgam can be used for the implementation of BDSs in cloud environments, where BDSs usually consist of multiple managed cloud services. As illustrated by the diagram in Figure 3, the static data are initially stored in a central persistent storage (e.g., HDFS, or a cloud storage). The analysis of such data in a DW requires data processing and loading into the tables of the corresponding relational databases via ETL jobs performed by a Hadoop cluster. In more details, the Hadoop cluster processes and transforms semi-structured and unstructured data into structured data to be processed more in DWs and databases. This is viable through different solutions provided by the Hadoop ecosystem for the processing and storage of data in NoSQL stores (e.g., HBase), DWs (e.g., Hive), and ML (e.g., Mahout) and statistical algorithms.

Note that it is often unreasonable to use a Hadoop cluster for the permanent storage of data as it incurs enormous expenses. This is due to the need for more virtual machines (VMs) for extra cluster nodes. Therefore, large cloud consumers (e.g., Netflix) demonstrate the benefit of using low-cost cloud storage services rather than the local storage of HDFS. As such, by taking into account various data analysis tasks, the same data may be accessed and processed by multiple clusters for distinct workloads. However, to compromise between the low-latency access to the local storage of HDFS and the low-cost access to the central cloud storage, all intermediate data of MapReduce jobs are stored in the local storage of HDFS.

However, the streaming data are sent to a stream processing component to be immediately utilized (e.g., by creating alerts or making suggestions based on ML algorithms). The streaming data may also be transmitted to the central cloud storage, just if such data are useful for future processing. This may require ETL jobs to reliably export the data (see, for example, Spotify's latest streaming solution using Google Cloud). Table 4 specifies the cloud-based Lambda architecture for the services provided by the top three cloud providers, Google Cloud, Amazon Web Service, and Microsoft Azure.

As a real case study, Nurminen and Mfula [2018] propose a Lambda-based architecture for 5G network management software tools. This architecture aims at addressing two main concerns in the well-known three-tier architecture in current tools: first, inability to handle network management operations that depend on long-term and real-time patterns of data and, second, the lack of support for the strict latency requirements of 5G networks. They extend Lambda by adding a data ingestion layer where the Kafka messaging platform is utilized to make a high-throughput connection between heterogeneous data producers and consumers. They use Apache Spark for

[24] In such an analysis, for instance, data mining models produced in the Batch layer are loaded as an initial configuration into the Speed layer. Furthermore, the data mining models exploited in the Batch layer may be adjusted by real-time views. This can lead to a maximum business benefit [Gröger et al. 2014].

Table 4. The Applicability of the Cloud-based Lambda Architecture for the Services of Major Cloud Providers

Cloud provider	Streaming service	Streaming analytics service	Central storage service	Data warehouse service	Hadoop cluster service	Machine learning service
Google Cloud https://cloud.google.com	Cloud Pub/Sub	Cloud Dataflow	Cloud Storage	BigQuery	Cloud Dataproc VMs	Prediction API
Amazon Web Services (AWS) https://aws.amazon.com	Kinesis Data Streams	Kinesis Data Analytics	S3	Redshift	EMR VMs	Amazon ML
Microsoft Azure https://azure.microsoft.com/	Even hub	Stream analytics	Azure Storage	SQL DW	HDInsight VMs	Azure ML

Table 5. Exemplar Use-cases of the Lambda Architecture

BDS	Domain	Data ingestion	Batch layer	Speed layer	Serving layer
AllJoyn Lambda [2014]	Smart cities	IoT-broker	MongoDB	Apache Storm	MongoDB
CiDAP [2015]	Smart cities	IoT-broker	Apache Spark + HDFS	Apache Samza	CouchDB
ELA [2018]	Building 5G network management software tools such as cell outage management	Apache Kafka	Apache Spark + HDFS	Spark Streaming	Apache Cassandra
RADStack [2017]	Interactive analytics	Apache Kafka	Apache Hadoop	Apache Samza	Druid
[2018]	Disease surveillance	N.A.	Apache Hadoop	Apache Hive streaming	Apache HBase
[2017]	Cardiovascular disease prediction	via API calls & via Map-only jobs	Apache Hadoop	Spark Streaming	HDFS
[2019]	Aviation manufacturing	Apache Kafka	Apache Hive + HDFS	Apache Storm	Apache HBase

the batch layer where Spark SQL API is used for the precalculation of batch views with regard to the long-term expected traffic patterns of network cells. However, Spark Streaming is used for the speed layer, whereby the sudden change in traffic patterns is captured from streaming data. Apache Cassandra, with an in-built support from Apache Spark, is used in the serving layer to persist the batch views. Table 5 depicts some existing BDSs based on the Lambda architecture and the corresponding technology stacks.

3.2.2 The Kappa Architecture. The ability to recompute data streams, the scalability feature of new message brokers (e.g., Apache Kafka) and the existing powerful stream processors (e.g., Apache Samza) resulted in the emergence of the Kappa architecture [Kreps 2014] as an alternative (not replacement) to Lambda. On the one hand, Kafka is a distributed, high-throughput and fault-tolerant message broker based on a publisher-subscriber model, where data publishers persist streaming events in a write-ahead log and data subscribers read data at their own pace. It ingests data streams as a set of replicated, append-only and immutable sequence of records ordered by time. Message persistence for a defined period facilitates reprocessing of streams. On the other hand, Samza is able to cope with data at a far greater rate than it is incoming.

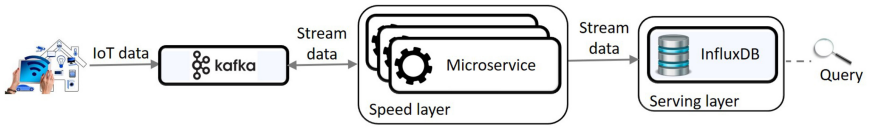

Fig. 4. An architecture based on the concepts of Kappa and microservices.

Figure 2(b) depicts the basic outline of the Kappa architecture. As illustrated, Kappa simplifies Lambda by performing all computation in a streaming layer alone. The streaming layer supports both batch and real-time processing by the buffering of historical data in a logging system for long enough. When recomputation is required, a new stream processing job is launched alongside an old one. It recomputes the historical data and outputs different results into the Serving layer. After the new job has caught up the old one, back-end systems read from the new result. In addition, the old job is terminated and the corresponding output result is deleted.

Note that when the underlying processing logic changes, a new stream processing job (version n+1) is launched alongside an old one (version n). Therefore, in comparison to the Lambda Architecture, evolving processing and analytics requirements are adapted more flexible; as Kappa needs to only reflect the changes at the first processing pipeline (job version n). Due to this flexibility, Kappa can cope with the diversity of analytics scenarios in the domain of consumer-centric Internet of Things (IoT), such as smart home. In this domain, analytics scenarios are handled on a much smaller scale than prevalent scenarios of Big Data. However, there exist a significant number of different small scenarios (per consumer) that require Big Data processing capabilities. Hence, instead of using a heavy full-fledged framework of stream processing, we can exploit a flexible lightweight microservice-based stream processing. Microservices use stream processing libraries, whereby each analytics scenario can be performed by a single, lightweight microservice.

Although Kappa provides a single processing path, the effort to recompute the entire history is linearly increased with the growing volume of data. In addition, Kappa does not fit some data analytics, such as ML algorithms, where there are different outputs of streaming and batch algorithms. This means considering the Kappa architecture as an alternative to Lambda in applications whose requirements allow for processing a sufficiently large segment of current streaming data (instead of the entire data volume). For example, cell outage compensation in 5G networks requires a long-term pattern matching [Nurminen and Mfula 2018].

As a real case study, Zschörnig et al. [2020] propose an analytics platform architecture based on the concepts of Kappa and microservices, to predict electricity consumption of households over a period. Smart meters produce consumption data, while various types of meters are used by various households. Accordingly, data with various structure and semantics are sent to smart home platform providers that are responsible to predict the consumers' energy consumption over a requested time period. A prediction model is created by feeding data into an ML training process. A growing number of households, with different consumption profiles, is equal to the need to train and implement different ML models that vary from one another. Figure 4 depicts the proposed analytics platform architecture. The speed layer is implemented using Apache Kafka, whereby data from different IoT data sources are ingested and distributed within the architecture. In addition, the serving layer is implemented using a time-series database named InfluxDB that stores processed data and answers ad hoc queries on the data. Table 6 depicts some existing BDSs based on the Kappa architecture and the corresponding technology stacks.

3.2.3 The Liquid Architecture. It avoids the drawback of the Lambda and Kappa architectures as they are unable to set up an incremental processing job [Fernandez et al. 2015]. Incremental processing makes data processing faster and more efficient by avoiding recomputation from

Table 6. Exemplar Use-cases of the Kappa Architecture

BDS	Domain	Data ingestion	Speed layer	Serving layer
[2020]	Personal analytics in IoT environments	Apache Kafka	Microservice stream processing	Time-series databases such as Graphite or InfluxDB
Cyclic [2017]	Load forecasting using event stream processing	Apache Kafka	Spark Streaming	Redis

scratch (when the input data changes). This requires offering fine-grained data access and handling transient computation states by the architecture. As such, Liquid annotates incoming streams with metadata, such as timestamps (at which data were read). As illustrated in Figure 2(c), Liquid includes Processing and Messaging layers.

The Messaging layer is a distributed, topic-based, publish/subscribe platform that is highly available and scalable. This is implemented by Apache Kafka whereby input messages are stored in topics consisting of one or more partitions. Kafka maintains metadata on data and enables metadata-based access to partitions. Producers can publish messages on topics and consumers can subscribe on a topic and wait for published messages. It is the responsibility of the Messaging layer to store both the primary data from sources and the derived data generated by the Processing layer. The derived data include metadata, such as lineage information about how the data were computed. The Processing layer is able to access data according to different metadata, such as the timestamp at which data were read. This allows reading data from particular points in time by the back-end data systems. As there is an offset manager that allows to checkpoint offsets, it is possible for the Processing layer to reprocess the last consumed data after failure.

The Processing layer uses a stateful stream processing model to perform jobs for distinct back-end data systems. These jobs perform various transformations to prepare raw input streams for applying more sophisticated querying and storage in the back-end systems. For example, the available data can be enriched with existing metadata, formatted, standardized, cleaned, ordered, or merged. This layer is implemented by Apache Samza that explicitly represents states as part of the computation. In addition, the Processing layer stores annotated data as metadata in the Messaging layer, enabling jobs to dynamically select input data streams. Each job executes data processing on messages from an input topic and publishes the results as messages to an output topic. The job results are named derived data feeds. They can be source for another processing job or being directly queried by the back-end systems. Each job is split into some tasks that process a topic's distinct partitions to provide parallel processing.

However, the scalability of the Processing layer is limited to the Messaging layer. In more details, the number of partitions of a topic restricts the maximum number of tasks in a job, which in turn limits the scalability of the jobs. Mirvakili et al. [2019] tackle this drawback by separating the Processing layer from the Messaging layer through an intermediate virtual messaging layer. This layer contains some virtual topics where each one corresponds to a topic in the Messaging layer. A virtual topic catches messages (about the corresponding topic) from the Messaging layer and distributes them among the tasks of the jobs that have subscribed to the topic. This makes independent the number of the tasks in a job from the number of the partitions in the topic on which the job has subscribed.

3.2.4 The Solid Architecture. It aims at integrating heterogeneous data under a single logical data model, namely RDF [Cuesta et al. 2013; Martínez-Prieto et al. 2015]. Using RDF, in conjunction with OWL, gives meaning to individual schemas and facilitates their efficient integration. Solid adapts the Lambda architecture to separate the complexities of real-time data acquisition

Table 7. Exemplar Use-cases of the Bolster Architecture

BDS	Domain	Data ingestion	Batch layer	Stream processing	Dispatcher	Semantice layer	Serving layer
BDAL [2017]	Batch analytics	FTP, Apache Sqoop	Apache Spark + HDFS	N.R.	N.R.	Virtuoso	SAS
SUPERSEDE [2017]	Feedback-driven SDLC	Apache Kafka	Apache Spark + HDFS	Spark Streaming	Apache Flume	Virtuoso	Apache HBase
WISCC [2017]	The Chagas disease	Data extraction drivers	MangoDB + Quarry [2015]	N.R.	N.R.	Virtuoso	Microsoft SQL Server

and consumption from managing big RDF datasets (i.e., big semantic data). The Data layer follows the Lambda's Batch layer principles. It can be viewed as a simple triplestore for storing big immutable RDF datasets, where raw triples are stored using a compact binary RDF representation, namely RDF/HDT. The Index layer, constructed above the Data layer, provides an efficient querying of the RDF datasets typically through SPARQL, that is the de-facto standard for querying RDF. However, the Online layer follows the Lambda's Speed layer principles. This layer captures new RDF streams and temporarily stores them into a runtime triplestore where a SPARQL interface facilitates communication with the Service layer. The online layer preserves the efficiency of insertions and SPARQL resolution through storing a small amount of data (as compared to the Data layer). This layer also triggers integrating runtime data into the historical one in the Data layer. When the quantity of temporary data stored reaches a certain limit, the Online layer will dump its data to be used as the Merge layer's input. The Merge layer (which is a massive-parallel batch process) integrates the earlier historical data with the runtime data at the Online layer request. After the integration, the dumped data are removed from the Speed layer as they are available via the Index layer. In spite of Lambda, where every new data piece is duplicated and maintained in the Serving and Batch layers, every new triple in SOLID is only collected by the Online layer. The integration is similar to Lambda's recomputation of views. Figure 2(d) provides the basic outline of the SOLID architecture. However, resource consumption is far less than Lambda as the single RDF/HDT serialization is recomputed by SOLID occasionally.

3.2.5 The Bolster Architecture. This architecture improves Lambda by adding a new Semantic layer containing a MetaData Management system (MDM) [Nadal et al. 2017], as shown in Figure 2(e). MDM is in charge of providing information needed to deal with data governance and the description and modeling of raw data. It includes a metadata repository where all the relevant machine-readable semantic annotations are represented in an RDF ontology. This ontology contains the input data characteristics such as what attributes they should have and where they come from. An RDF triplestores (e.g., Virtuoso[25]) can be used to store all the metadata artifacts. Bolster exploits a Dispatcher component, such as Apache Flume, that is in charge of ensuring stream data routing and stream data quality. Unlike the Lambda architecture where all input streams are fed to both the Batch and the Speed layers in parallel, Dispatcher decides where the streaming data are shipped (to either the Batch or the Speed layer). This decision can be influenced by analyzing the system workload or evaluating QoS cost models, as performed in Kroß et al. [2015]. Dispatcher also ensures if all ingested data follow a schema specified in MDM for the corresponding data sources. Table 7 depicts some existing BDSs based on the Bolster architecture and the corresponding technology stacks.

[25] http://virtuoso.openlinksw.com.

3.2.6 The Architecture of a Polystore. This architecture, whose first implementation is BigDawg [Duggan et al. 2015; Elmore et al. 2015], is a thorough solution for unified querying over multiple heterogeneous storage engines with different data and query models. Polystore is based on the "no one size fits all" observation for data management solutions. It organizes data, that span multiple data models, into information islands. An island represents a category of storage engines and provides a single data model and query language suitable to access the corresponding engines. For example, a relational island may be a collection of traditional relational DBMSs, such as MySQL or Postgres. It is also possible for an individual database engine, that belongs to multiple categories, to be included in multiple islands. A user can query an island through the corresponding query language. As such, for each storage engine of an island, there is a software module, called *shim*, whereby the island language is mapped into the native language of the engine.

Polystore establishes an integration framework based on the aforementioned *virtual* and *materialized* methods. In more details, underlying storage engines are virtually integrated by islands. Furthermore, a user can request the physical movement of data items (using the CAST operator) between storage engines. This happens when more than one storage engines are accessed by a single- or cross-island query. This migration implies transforming the required data from the source system to a serialized binary format that the target engine can then interpret. To handle the distinctive requirements of streaming data, all streaming systems (e.g., S-Store [Meehan et al. 2015], Apache Spark Streaming and Storm) are included in a streaming island that is push-based[26] (unlike the other non-streaming islands that are pull based). This is implemented by a scalable publish-subscribe messaging module, such as Apache Kafka, which also selects the proper stream processing system to be fed by the input streaming data. This architecture is comparable to Kappa and Liquid, because a streaming system handles all new input data. However, instead of just tackling the *Volume* and *Velocity* challenges of Big data, it focuses on *Variety* as well, and the corresponding serving layer is made up of heterogeneous storage engines.

As an illustration, Figure 2(f) depicts an example architecture of a polystore. Data are organized in three information islands: a streaming, a relational and a graph island. The first island contains S-Store [Meehan et al. 2015], streaming system with its own in-memory data source DS1. It facilitates streaming ETL by ingesting raw data (in configurable chucks of) steams from multiple sources, performing various transformations such as data cleaning and storing the data into DS1. Despite traditional ETL where the intermediate results are written to files, the results in streaming ETL are pushed into the next element in the ETL pipeline. The transformed streams can then be used for real-time analytics. They can also be incrementally pushed in batches, via the CAST operator, into appropriate underlying storage engines for batch analytics. However, the relational island contains two relational data sources DS2 and DS3 with their own dedicated shims, whereby a relational interface is exposed to the island. The third island includes a single graph data source DS4 whose shim exposes a graph interface to the island. Note that the two latter islands share in DS3 via two shims; therefore, DS3 can be queried from both the islands.

3.3 Evaluation of BDS Architectures

In the following, we analyze the satisfaction of the architectural requirements shown in Table 3 by each of the previously mentioned BDS architectures. Table 8 summarized the results of evaluation, where SRAs and custom architectures are distinguished.

3.3.1 Requirements on Volume. DL-based architectures (as in Lambda and Bolster) meet all the requirements on Volume. They offer an scalable storage and processing of stored data (R1.1), where

[26] It means that new data are automatically fed into the destination engines.

Table 8. Fulfilment of the Requirements in State-of-the-Art BDS Architectures

SRAs	DL based	1. Volume			2. Velocity		3. Variety			4. Variability			5. Veracity			
		R1	R2	R3	R1	R2	R1	R2	R3	R1	R2	R3	R1	R2	R3	R4
Lambda [Marz and Warren 2015]	Yes	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Kappa [Kreps 2014]	No	✗	✓	✤	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Liquid [Fernandez et al. 2015]	No	✗	✓	✤	✓	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗
Bolster [Nadal 2019; Nadal et al. 2017]	Yes	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✗	✗	✗
Solid [Martínez-Prieto et al. 2015]	No	✗	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
Polystore [Meehan et al. 2016]	No	✓	✓	✤	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗

Full support (✓), Limited support (✤), No support (✗).

MapReduce (or similar interfaces) support the analytical capabilities (R1.2 and R1.3). However, the scalable storage and processing of massive datasets (R1.1) is not met by Kappa and Liquid as no data are stored. Note that there may also be a bottleneck due to the exchange of data between a message broker, such as Apache Kafka and the streaming system [Karimov et al. 2018]). This requirement is not also enforced by Solid. This stems from the processing capabilities of triplestores (even though there are some efforts [Davoudian 2019; Shi et al. 2016; Wang et al. 2018] on improving these capabilities, the W3C recommendations does not include any mature scalable solution). With regard to the analytical capabilities, the descriptive level (R1.2) is satisfied by all the architectures via the stream or batch processing of data (as in Lambda, Kappa, Liquid, Bolster and Polystore), via SPARQL (as in Solid) or via an abstract language corresponding to each information island (as in Polystore). In addition, although the complex predictive/prescriptive level (R1.3) is fully satisfied by the Lambda and Bolster architectures, it is partially satisfied by Kappa, Liquid and Polystore due to the stream processing of a segment of data instead of the entire data volume.

3.3.2 Requirements on Velocity. Stream extraction (R2.1) and processing (R2.2) requirements are met by all architectures.

3.3.3 Requirements on Variety. The ingestion of raw data (R3.1) is met by all the architectures except Solid, as it ingests only RDF data. However, schema management (R3.2) is met by Bolster, Solid, and Polystore. In this regard, Bolster virtually integrates data via an OBDA method, where the LAV paradigm is used for the declarative mappings between the global ontology and the underlying data sources [Nadal 2019]. In Solid, a unified model (i.e., RDF) of raw data is provided. Polystore meets this requirement by providing the same data model to access the engines of an island of information. However, none of the architectures resolve the semantic data interoperability conflicts (R3.3). Regarding the Big Data integration requirements (R3.2 and R3.3), to provide an efficient data analysis, a BDS should fully exploit metadata in all data management activities, including information extraction, data cleaning and data integration [Ceravolo et al. 2018]. However, metadata management is not fully established by the architectures.

3.3.4 Requirements on Variability. Bolster is the only architecture that meets the requirements on Variability. In this regard, it stores the schema information of ingested elements (R4.1), descriptive statistics to access data evolution (R4.2) and the information of input data sources (R4.3) in MDM.

3.3.5 Requirements on Veracity. Veracity is satisfied by almost none of the architectures. Data provenance (R5.1) is only met by Liquid as it supports logging the performed transformation steps on derived data.

As Table 8 shows, Volume, Velocity, and partly Variety (i.e., R3.1) are more fulfilled with respect to the other ones. In some degree, this is because of the fact that the corresponding requirements are the main functionalities of Big Data technologies, such as batch and stream processing engines and NoSQL stores. However, satisfying the other requirements depends on the utilization of data semantics in data management activities. Traditional relational databases and data warehouses offer metadata repositories and metadata management as a builtin feature [Poole et al. 2002]. However, a metadata standard has not yet been developed for the current landscape of Big Data technologies. In summary, BDS architectures still lack a comprehensive, sound approach to metadata management.

However, regarding the complexity of the aforementioned SRAs, it still remains challenging to select the most suitable architecture for a specific use case scenario. This selection requires considering the above architectural requirements. In this respect, [Volk et al. 2019] introduce a decision maker by developing the well-known Analytical Hierarchy Process (AHP) for multi criteria decision making. After determining the candidate SRAs, they are contrasted using SACAM [Stoermer et al. 2003] that is a scenario-based method for the comparison of software architectures. The comparison results are used afterward for developing an AHP for SRAs.

4 BDS QUALITY ASSURANCE

Traditional activities and concerns of quality assurance do not take into account the characteristics of Big Data. This imposes new challenges on quality assurance as elaborated in the following categories.

- *The verification of test results.* The data-driven behavior of a ML model (i.e. supervised and unsupervised learning), where a training algorithm determines the evolving decision logic from the evolving training data, makes it often impossible to verify the expected test results of a given input. This, in turn, makes its quality assurance challenging[27] [Amershi et al. 2019]. For example, the movement of a robot to assemble a part of a car can only be verified with uncertain ML algorithms such as Deep Neural Networks (DNNs) [Tian et al. 2018]. Note that the formal verification of a ML model does not guarantee its proper usage or implementation by an application [Otero and Peter 2014].
- *Resource-intensive testing environments.* The comprehensive testing of a BDS needs a workload that is comparable to the one used in the production system. In addition, the repetition and comparison of tests leads to the storage of different test datasets and the corresponding results. These entail a testing environment with many processing cores (maybe geographically distributed) similar to the real system, along with high storage capacity [Madhavji et al. 2015]. However, this may not be feasible due to the operating costs or efforts to test involved with such environments. This, in turn, makes it impossible to comprehensively test every aspect of BDSs at scale before deployment to production. Note that proper functionality and behavior of a BDS at one scale might not be preserved at bigger scales.
- *The generation of an optimal set of realistic test datasets.* A sophisticated testing of BDSs requires providing realistic and application-specific test datasets that cover all characteristics of Big Data. This makes the generation of such datasets a challenging task, particularly when deciding on an optimal coverage.
- *Error-tracing, logging and debugging.* The distributed nature of BDSs necessitates testing of such systems to be in a distributed environment. However, due to the current limitations of distributed development and debugging tools, BDS developers usually have to

[27] A software system with no reliable test oracle is sometimes referred to as a "non-testable" software [Weyuker 1982].

use distributed log files. Therefore, understanding BDS behaviors and tracing back the observed errors to their origins rely on distributed log files that is inherently difficult. This may require mining of the log files [Pettinato et al. 2019] and eventually needs to develop methods for sophisticated distributed debugging.

— *Verification methods.* Due to a combinatorial state explosion, the use of distributed computing to process Big Data leads to complicated verification methods. Despite a breakthrough in verification approaches made by partial reduction techniques and symbolic model checking, the verification capabilities of explicit state models are still better [Camilli 2014].

— *The confusing notion of consistency.* With respect to the CAP theorem, the notion of data consistency is weakened in BDSs when trading consistency for availability. However, these inconsistencies may confuse users and quality technicians, since the latest updates to the data may not be visible in subsequent read requests.

— *The assessment of data quality.* A prerequisite for Big Data analysis is that the data are of adequate quality for the context of use. However, for the following main reasons, this is a non-trivial task [Auer and Felderer 2019]:

- *Volume*, as the quality assurance of voluminous data with reasonable resources and within reasonable time is challenging, especially since the data are mostly unstructured and initially require analysis (e.g, text analysis) or transformation. This can be tackled by the assessment of a representative sample of the collected data [Kläs et al. 2016; Taleb et al. 2016]. However, there is an issue of selecting a reasonable sample size and an appropriate sampling strategy, especially in large datasets.
- *Velocity*, as the data of heavy timeliness (e.g., sensor measurements of the environment) require real-time analysis, otherwise the data become obsolete. Here the challenge is that data quality assessment may hinder the real-time analysis of data. As another challenge, frequent update of data might change the corresponding quality. One solution is the complete reassessment of data quality, which may cause performance problems for voluminous data.
- *Variety*, as the integration of the data of various types results in far more inconsistencies and conflicts, which in turn makes data quality assurance more challenging.

The above challenges necessitate applying new approaches for *software testing* and *data quality assurance* whose corresponding state of the research and hot topics are presented in this section.

4.1 Software Testing

The software testing process is concerned with the evaluation of a System Under Test (SUT) and related development artefacts by checking both the satisfaction of all specified requirements (i.e., verification) and meeting all user expectations (i.e., validation) followed by detecting faults (or defects[28] [ISTQB 2018; Young 2008]. By taking BDSs into account, software testing can be classified with respect to three dimensions: *test objective*, *granularity level*, and *test execution level*, as the following:

— *Test objective*, whereby there are three categories of software testing: *functional testing*, *non-functional testing*, and *data quality testing*. Functional testing is intended to identify errors in SUT's functionality requirements. For example, testing the generation of right outputs for right inputs. In contrast, non-functional testing aims at assessing the quality requirements

[28] A fault is usually caused by human errors either in specification, design, or coding phase and results in a failure (or undesired system behavior).

of SUT, such as reliability, performance, security, or safety. Data quality testing is explained in Section 4.2.

— *Test granularity level*, whereby software testing is classified into four categories: *algorithm testing*, *unit (or component) testing*, *subsystem (or integration) testing*, and *system testing*. (I) Algorithm testing evaluates ML algorithms, especially the ones used in Deep Learning systems with thousands of parameters and neurons. In more details, unlike traditional software where developers write the program logic manually, in Deep Learning systems, program logic is automatically learned from a massive amount of data while human guidance is minimal. After detecting the erroneous behavior of such systems, such behavior can be fixed by altering the parameters of the model/structure or adding inputs that induce errors to the training dataset. (II) Unit (or component) testing evaluates every class in an object-oriented implementation. (III) Subsystem (or integration) testing checks the integration of components as a subsystem. (IV) System testing checks the complete system.

— *Test execution level*, whereby there are three categories of software testing: *static testing*, *dynamic testing*, and *runtime monitoring* to observe the erroneous behaviour of the system.

- *Static testing*, such as a review or static analysis, which checks software development artefacts (e.g., requirement, design, or code documents) with no execution.
- *Dynamic testing*, which executes a test suite[29] to evaluate whether a SUT behaves as expected or not. After performing a test-case, the SUT's expected and actual behaviors are compared with each other. This is performed by a *test oracle* (or simply an *oracle*) mechanism and results in a *verdict*, which can be either *pass* (conforming behaviors), *fail* (non-conforming behaviors due to some failures), or *inconclusive* (not knowing whether behaviors conform).

  However, this comparison is not always feasible, especially in BDSs with ML components, where it is impossible or too expensive to specify the intended (or correct) output of a given input. This refers to one of the fundamental software testing issues, called the *oracle problem* [Barr et al. 2014; Weyuker 1982], which can be alleviated by a technique called *Metamorphic Testing (MT)* [Chen et al. 1998]. This approach is based on an idea saying that reasoning about relations between the outputs of two or more related inputs is easier than assessing a single actual output of it. More precisely, in the absence of an ideal oracle to verify each individual output, MT verifies the functional correctness of software and reveals failures through checking expected relations (a.k.a. MT-relations) between multiple outputs of the SUT.

  Violating a MT-relation reveals a defect in the implementation of the selected algorithm. This entails constructing MT-relations based on the necessary properties of the algorithm, which in turn means using MT for the verification as in Tian et al. [2018].[30] This violation can also reveal the algorithm's deficiency in meeting user's expectations. This requires constructing MT-relations based on the user expectations on the algorithm, which in turn means using MT for the validation as in Zhou et al. [2015]. Note that MT's

[29] Assuming a test-case as an input selected for the evaluation of SUT, a test suite is a finite set of properly chosen test-cases from the usually infinite execution domain.

[30] As an instance, in a program P that computes the shortest path s between two nodes of an undirected graph, assume for a given non-trivial graph G and two nodes (x, y) in G, the expected output $s(G, x, y)$ cannot be correctly and precisely specified. However, we can apply MT by taking into account the following MT-relation: *If* G_2 *is a permutation of* G_1 *(i.e.,* G_1 *and* G_2 *are isomorphic) and* (x_1, y_1) *in* G_1 *correspond to* (x_2, y_2) *in* G_2*, then* $s(G_1, x_1, y_1) = s(G_2, x_2, y_2)$. Now suppose the results of two executions (one with input (x_1, y_1) and the other with input (x_2, y_2) are different. Therefore, the above relation is violated, which in turn concludes that P is faulty. In addition, we could consider many more MT-relations such as $s(G, x, y) = s(G, y, x)$ [Chen et al. 2004].

Table 9. State-of-the-Art of Research on Testing BDSs

Paper	Test execution level	Test objective	Test granularity level	BDS context	Contribution
DeepTest [Tian et al. 2018]	Dynamic via MT	Functional	Algorithm	Autonomous driving	Tool
DeepRoad [Zhang et al. 2018]	Dynamic via MT	Functional	Algorithm	Autonomous driving	Tool
[Ding et al. 2017]	Dynamic via MT	Non-Functional	Algorithm	Image classification	Tool
[Zhou et al. 2015]	Dynamic via MT	Non-functional	Algorithm	Search engines	Tool
[Auer and Felderer 2018]	Runtime	Non-functional	Algorithm	General	Method

effectiveness depends on the quality of the MT-relations recognized and the relevant test suites. However, the identification of proper and effective MT-relations is still challenging and requires deep understanding of the domain in question, software testing and ML algorithms. The readers are referred to recent surveys on MT testing [Chen et al. 2018; Segura et al. 2016] and generally on ML testing [Zhang et al. 2020].

- *Runtime monitoring*, whereby a SUT has to be instrumented for the collection of runtime data and analyzing the effect. This allows evaluating software quality attributes such as efficiency (e.g., execution time), usability (e.g., user feedback), reliability (e.g., frequency of failures), and functionality (e.g., unsatisfying results).

Recent works demonstrate the feasibility of MT to ensure the quality of ML-based BDSs (see Table 9). Deeptest [Tian et al. 2018] exploits a MT-based method for the verification of DNN-based software, such as those used to drive autonomous cars.[31] Suppose that the software takes a road image as input and then outputs the steering angle. Therefore, as a MT-relation, the same image under any lighting/weather conditions should not significantly change the autonomous car's steering angle. Accordingly, DeepTest applies various filters such as rain/fog/blurring and simple transformations to training driving scenes, and then checks the aforementioned MT-relation. DeepTest can cheaply and rapidly detect multiple inconsistent and erroneous driving behaviors for some real-world models of autonomous driving by taking into consideration large quantities of initial and transformed driving scenes. However, Zhang et al. [2018] claim that transformed driving scenes generated by DeepTest are distorted and cannot correctly represent the driving scenes of the real-world. As such, they develop DeepRoad, which similarly tests DNN-based autonomous driving systems. However, it synthesizes driving scenes with various weather conditions through a Generative Adversarial Network– (GAN) based technique [Goodfellow et al. 2014].

Ding et al. [2017] propose an MT-based method for rigorously validating a ML framework that classifies biology cell images. This method takes into account the whole framework including a neural network, a massive image dataset and an execution environment. Accordingly, the authors construct MT-relations on three distinct levels: (1) system level, where MT-relations are defined based on the classification accuracy relation of alternative ML algorithms; (2) dataset level, where MT-relations are defined based on the classification accuracy relation of reorganized training datasets; and (3) data item level, where MT-relations are defined based on the classification accuracy relation of reproduced individual images. Similar works in testing ML systems can be found in Du et al. [2018], Kim et al. [2019], and Pei et al. [2017].

[31] https://deeplearningtest.github.io/deepTest/.

Zhou et al. [2015] introduce a user-oriented MT for the validation of search engines whose specifications and algorithms are usually unknown to the users. Despite the conventional MT, where MT-relations are designed based on the target algorithms, the relations in the user-oriented MT are defined based on the expectations of users to reflect what is really important to them. In more details, MT-relations are determined from available online specifications (which are accessible via the online help pages of search engine), and the set of functionalities offered by search engines to users, and are related to certain software quality characteristics such as usability and reliability.

The aforementioned challenges of testing BDSs are still relevant to the above related works. In more details, regarding the verification of test results, future research could explore more reliable oracles for testing DNN-based systems. For instance, Stocco et al. [2019] introduce anticipatory testing as an alternative to MT, where a new type of self-assessment oracles detect future system failures at runtime.[32] In addition, there is no suitable testing approach focusing on Velocity as a Big Data characteristic. Instead, the current effort of research community is mostly on Volume and Variety of data [Pettinato et al. 2019]. However, Auer and Felderer [2018] reveal a fundamental weakness of existing ML testing solutions as the data dependent behavior of an ML algorithm results in a limited reasoning about its later quality. This requires shifting from costly test environment for simulation to available live system where the algorithm is constantly executed.

4.2 Big Data Assessment for Quality Assurance

Although valuable insights can be extracted via Big Data analysis, the results of such analysis are barely reliable unless proper quality assurance activities are applied before using the data. Data quality assurance is a process whereby the quality of data is initially assessed or measured, with respect to a Data Quality Model (DQM), and then improved through data cleansing activities.[33] A DQM is a set of measurable dimensions or characteristics[34] of data quality. In addition, each dimension is quantified by one or more associated quality metrics or formulas yielding numerical values. The standard ISO/IEC 25012 is a well-known DQM containing the most desirable data quality dimensions, categorized as *intrinsic* and *system-dependent* (see Table 10). An intrinsic data quality dimension is inherently fulfilled by the data, with no dependency on computer systems' capabilities, whereas a system-dependent one is achieved by a technological domain where the data are used. Currently, the research on the effective quality assessment of Big Data is in its initial stage. Table 11 depicts the state-of-the-art of research on this context.

Through the existing well-known DQMs for regular data, such as ISO/ETC 25012 and ISO/TS 8000-1, data quality is assessed with no respect to the context of use. Accordingly, Merino et al. [2016] propose a DQM derived from ISO/ETC 25012, that enables evaluating the quality-in-use of big datasets regarding the intended analysis. In more details, this 3A model reclassifies the data quality characteristics of ISO/ETC 25012 into three categories of Adequacy: (1) *contextual adequacy*, as the fact that the input data is capable of being used within the domain of the intended analysis; (2) *temporal adequacy*, as the fact that the age of input data is acceptable for the intended analysis; and (3) *operational adequacy*, as the fact that the input data can be fully analyzed via sufficient and appropriate resources. For each category, they select suitable data quality characteristics (of ISO/ETC 25012) to assess the quality of Big Data regarding their Volume, Velocity and Variety (see Table 12).

[32] http://www.pre-crime.eu/.
[33] A comprehensive survey on Big Data cleansing activities is provided in [Mirzaie et al. 2019].
[34] Note that there is no general consensus in literature on either all data quality dimensions or the exact meaning of each one [Ehrlinger et al. 2019].

Table 10. A Summary of Data Quality Dimensions Defined in ISO-25012

Data quality characteristics	Description	Inherent	System-dependent
Accuracy/ Correctness	It refers to the closeness of data values to the corresponding reference values. The accuracy of some values are simply measured through their known reference values, obtained by specific business rules such as "a gender value can be *Male* or *Female*". The measurement of others is usually context-aware and requires extra information. As an example of IoT scenario, a sensed temperature value is not accurate if it deviates the average of temperature values in the past two hours by more that 15%.	✓	
Timeliness/ Currentness	it refers to the degree to which the collected are temporally valid for the intended analysis.	✓	
Completeness	It refers to the degree to which the collected data covers the data desired for the analysis.	✓	
Consistency	It refers to the coherence among a set of data items which are semantically related. For example, base on some consistency rules, the consistency of a sensed value of temperature may be measured regarding the sensed values of humidity and relative pressure.	✓	
Credibility	The degree to which the data are believable by users.	✓	
Accessibility	It refers to the easiness of accessing the data by public users and enterprises.	✓	✓
Compliance	The degree to which the data are complied with conventions or standards.	✓	✓
Confidentiality	It refers to providing an authorized access to the data.	✓	✓
Efficiency	The degree to which the data are accessed efficiently.	✓	✓
Precision	In an IoT scenario, it is the degree to which successive sensed values are identical or similar (i.e., smaller the standard deviation, higher the precision).	✓	✓
Traceability	It refers to providing an audit trail of accesses (and changes made) to the data.	✓	✓
Understandability	The degree to which the data are clear and easy to understand for users.	✓	✓
Portability	It refers to preserving the quality of data after moving the data from one system to another.		✓
Availability	The degree to which the data can be retrieved by users.		✓
Recoverability	It refer to preserving the quality of data after a system failure.		✓

In a context-aware data quality assessment, the set of quality dimensions and related metrics are adapted with the context of assessment, determined by data source, data type and intended analysis. For example, the assessment of accuracy of a value in a batch dataset needs a metric different from the one in a sensor data stream. In this regard, Merino et al. [2016] present an architecture for context-aware data quality assessment. Initially, the user specifies the requirements such as consistency rules, the granularity level of assessment, required quality dimensions and list of attributes to be considered in the assessment. Subsequently, a context-aware assessment module measures the specified dimensions at the requested granularity. However, this architecture does not cover the assessment of integrated data collected form multiple sources.

To reduce the computing costs of data quality evaluation, Taleb et al. [2016] reduce the size of big datasets through sampling techniques. This evaluation scheme contains the following modules: (1) sampling of data via the Bootstrap strategy; (2) profiling of data samples via the corresponding metadata and data parsing tools, whereby data characteristics, such as the description of data

Table 11. State-of-the-Art of Research on Data Quality Assessment in the Context of BDSs

Paper	Contribution	Real-time assessment	Assessed quality dimensions	Unstructured data assessment	Empirically validated
[Auer and Felderer 2019]	Method	No	Intrinsic dimensions in ISO/IEC 25012	No	Yes
[Cappiello et al. 2019]	Method	Yes	Accuracy, completeness, consistency and precision	No	Yes
[Cappiello et al. 2018]	Method	No	Accuracy, completeness, consistency, distinctness, precision, timeliness and volume	No	Yes
[Taleb et al. 2016]	Method	No	Completeness and consistency	No	Yes
[Kläs et al. 2016]	Method	No	Dimensions in ISO/IEC 25012	No	Yes
[Immonen et al. 2015]	Architecture	Not relevant	Accuracy, completeness and consistency	Yes	Yes
[Merino et al. 2016]	Model	Not relevant	Dimensions in ISO/IEC 25012	Not relevant	Yes
[Ehrlinger et al. 2019]	A Survey on Tools	Not relevant	Accuracy, completeness, consistency and timeliness	Not relevant	Yes

Table 12. The Assessment of 3As Affected by 3Vs

	Volume	Velocity	Variety
Contextual adequacy	—Completeness —Consistency —Confidentiality	—Completeness	—Accuracy —Consistency —Credibility —Compliance —Confidentiality —Understandability
Temporal adequacy	—Timeliness	—Timeliness —Accuracy	—Timeliness —Consistency
Operational adequacy	—Efficiency	—Efficiency —Confidentiality	—Efficiency —Confidentiality —Accessibility

formats, various attributes and their related types, values, ranges, constraints (if any), as well as data generation speed and data quality dimensions are discovered from data sources; (3) selection of data attributes and quality dimensions; (4) selection of quality metrics; and (5) evaluation of samples data.

Kläs et al. [2016] propose similar assessment method based on sampling. This method tackles the reassessment of frequently updated datasets by only evaluating of the changed part rather than the entire data. Due to the complex calculation of the changed part, authors assume no update or delete of existing data. Their assumption is based on the common usage of immutable data stores in BDSs. Ehrlinger et al. [2019] evaluate the capabilities of state-of-the-art data quality assessment tools on measuring accuracy, timeliness, consistency, and completeness that are relevant in many studies. Regarding the investigation results, none of the tools implement a context-aware assessment of data quality.

According to the above related works on Big Data quality assessment, many methods [Auer and Felderer 2019; Cappiello et al. 2019, 2018; Taleb et al. 2016] only assess the quality of data when

Table 13. Some Open Research Challenges in Developing BDSs

Research aspect	Open research challenges
Architecture design	—BDS architectures lack a comprehensive, sound approach to metadata management. —It is still challenging to select the most suitable architecture for a specific use case scenario. —The need for designing a semantics-based BDS reference architecture aiming to support all the architectural requirements (see Table 3).
Software quality assurance	—How to shift from costly test environments for simulation to available live systems where algorithms are constantly executed. —Scaling down the test environment or datasets while maintaining their characteristics. —The need for more reliable oracles for testing DNN-based systems. —The need for automatic construction of test oracles. —There is no suitable testing approach focusing on Velocity as a Big Data characteristic. —The relatively immature tool support of testing in the context of BDS.
Data quality assessment	—Existing assessment tools have limitations to implement data quality metrics (e.g., timeliness metrics). —New data quality characteristics and metrics are required for semi-structured and unstructured data, and the changing expectations of users. —Providing a fast context-aware strategy for the quality assessment of streaming data. —There is no standard quality model for Big Data, addressing quality dimensions and related metrics. —There is no tool providing a context-aware assessment of data.

they are collected; however, we believe that it should be handled at all phases of data lifecycle, when data are processed and analyzed. That is why we considered the architectural requirements on Veracity in Section 3. However, there is no method for the assessment of unstructured data (e.g., textual or image data), which is due to the challenging task in analyzing their correlation and semantics. This motivates using new assessment strategies that focus on the available metadata rather than assessed data itself. It, in turn, requires a metadata management functionality as explained in Immonen et al. [2015] and Kläs et al. [2016]. Furthermore, none of the existing methods focuses on context-aware strategies for speeding up the online quality assessment of streaming data.

5 CONCLUSION AND OPEN RESEARCH CHALLENGES

Despite existing a wide range of SE methods to develop robust software systems, engineering of BDSs is still in its infancy. This survey provides a wide overview of specific challenges that the development of BDS imposes on software engineers. It also addresses these issues by providing a comprehensive overview of state-of-the-art research and industry efforts in the engineering of requirements, designing and constructing software to meet the specified requirements, and software/data quality assurance in the context of BDSs. Furthermore, we have disclosed some existing BDS software engineering challenges that have been confronted by BDS developers and designers. Table 13 summarizes some of these open research challenges. The article should be beneficial to increase practitioners' awareness of prevalent challenges and to offer researchers with a strong basis for new research directions in software engineering.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their critical reading of the article and their valuable feedback, which has substantially helped to improve the quality and accuracy of this article.

REFERENCES

Daniel Abadi, Anastasia Ailamaki, David Andersen, Peter Bailis, Magdalena Balazinska, Philip Bernstein, Peter Boncz, Surajit Chaudhuri, Alvin Cheung, AnHai Doan, et al. 2020. The seattle report on database research. *ACM SIGMOD Rec.* 48, 4 (2020), 44–53.

Noufa Al-Najran. 2015. *A Requirements Specification Framework for Big Data Collection and Capture*. Master's thesis. Prince Sultan University, Riyadh.

Noufa Al-Najran and Ajantha Dahanayake. 2015. A requirements specification framework for Big Data collection and capture. In *East European Conference on Advances in Databases and Information Systems*. Springer, 12–19.

Ibrahim Alhassan, David Sammon, and Mary Daly. 2016. Data governance activities: An analysis of the literature. *J. Dec. Syst.* 25, sup1 (2016), 64–75.

Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: A case study. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. IEEE Press, 291–300.

Arcitura. 2017. Big Data Patterns and Mechanisms. Retrieved July 23, 2019 from http://www.bigdatapatterns.org/.

Darlan Arruda. 2018. Requirements engineering in the context of Big Data applications. *ACM SIGSOFT Softw. Eng. Not.* 43, 1 (2018), 1–6.

Darlan Arruda and Nazim H. Madhavji. 2017. Towards a requirements engineering artefact model in the context of Big Data software development projects. In *Proceedings of the International Conference on Big Data*. IEEE, 2314–2319.

Darlan Arruda and Nazim H. Madhavji. 2019. QualiBD: A tool for modelling quality requirements for Big Data applications. In *Proceedings of the International Conference on Big Data (Big Data'19)*. IEEE, 5977–5979.

Florian Auer and Michael Felderer. 2018. Shifting quality assurance of machine learning algorithms to live systems.

Florian Auer and Michael Felderer. 2019. Addressing data quality problems with metamorphic data relations. In *Proceedings of the 4th International Workshop on Metamorphic Testing*. IEEE Press, 76–83.

Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and issues in data stream systems. In *Proceedings of the 21th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM, 1–18.

Wolf-Tilo Balke. 2012. Introduction to information extraction: Basic notions and current trends. *Datenb.-Spektr.* 12, 2 (2012), 81–88.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE Trans. Softw. Eng.* 41, 5 (2014), 507–525.

Andreas Bauer and Holger Günzel. 2013. *Data-Warehouse-Systeme: Architektur, Entwicklung, Anwendung*. dpunkt.verlag.

Marcello M. Bersani, Francesco Marconi, Matteo Rossi, and Madalina Erascu. 2016. A tool for verification of Big-Data applications. In *Proceedings of the 2nd International Workshop on Quality-Aware DevOps*. ACM, 44–45.

Jeff Bertolucci. 2013. Big Data analytics: Descriptive vs. predictive vs. prescriptive. Retrieved from https://www.informationweek.com/big-data/big-data-analytics/big-data-analytics-descriptive-vs-predictive-vs-prescriptive/d/d-id/1113279.

Tobias Bleifuß, Leon Bornemann, Theodore Johnson, Dmitri V. Kalashnikov, Felix Naumann, and Divesh Srivastava. 2018. Exploring change: A new dimension of data analytics. *Proc. VLDB Endow.* 12, 2 (2018), 85–98.

Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. 2014. Summingbird: A framework for integrating batch and online mapreduce computations. *Proc. VLDB Endow.* 7, 13 (2014), 1441–1451.

Paul Buitelaar, Philipp Cimiano, and Bernardo Magnini. 2005. *Ontology Learning from Text: Methods, Evaluation and Applications*. Vol. 123. IOS Press.

Matteo Camilli. 2014. Formal verification problems in a Big Data world: Towards a mighty synergy. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 638–641.

Cinzia Cappiello, Marco Comuzzi, Florian Daniel, and Giovanni Meroni. 2019. Data quality control in blockchain applications. In *Proceedings of the International Conference on Business Process Management*. Springer, 166–181.

Cinzia Cappiello, Walter Samá, and Monica Vitali. 2018. Quality awareness for a successful Big Data exploitation. In *Proceedings of the 22nd International Database Engineering & Applications Symposium*. 37–44.

Otávio Carvalho, Eduardo Roloff, and Philippe O. A. Navaux. 2017. A distributed stream processing based architecture for IoT smart grids monitoring. In *Proceedings of the 10th International Conference on Utility and Cloud Computing*. ACM, 9–14.

Rick Cattell. 2011. Scalable SQL and NoSQL data stores. *ACM SIGMOD Rec.* 39, 4 (2011), 12–27.

Paolo Ceravolo, Antonia Azzini, Marco Angelini, Tiziana Catarci, Philippe Cudré-Mauroux, Ernesto Damiani, Alexandra Mazak, Maurice Van Keulen, Mustafa Jarrar, Giuseppe Santucci, et al. 2018. Big Data semantics. *J. Data Semant.* 7, 2 (2018), 65–85.

Hong-Mei Chen, Rick Kazman, and Serge Haziyev. 2016. Agile Big Data analytics development: An architecture-centric approach. In *Proceedings of the 49th Hawaii International Conference on System Sciences*. IEEE, 5378–5387.

Hong-Mei Chen, Rick Kazman, Serge Haziyev, and Olha Hrytsay. 2015. Big Data system development: An embedded case study with a global outsourcing firm. In *Proceedings of the 1st International Workshop on Big Data Software Engineering*. IEEE, 44–50.

Tsong Y. Chen, Shing C. Cheung, and Shiu Ming Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases*. Technical Report HKUST-CS98-01. Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong.

Tsong Yueh Chen, D. H. Huang, T. H. Tse, and Zhi Quan Zhou. 2004. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*. Polytechnic University of Madrid, 569–583.

Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.* 51, 1 (2018), 4.

Bin Cheng, Salvatore Longo, Flavio Cirillo, Martin Bauer, and Ernoe Kovacs. 2015. Building a Big Data platform for smart cities: Experience and lessons from santander. In *Proceedings of the International Congress on Big Data*. IEEE, 592–599.

Paul Clements and Len Bass. 2010. *Relating Business Goals to Architecturally Significant Requirements for Software Systems*. Technical Report. Carnegie-Mellon University, Software Eengineering Institute.

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

Carlos Costa and Maribel Yasmina Santos. 2016. Reinventing the energy bill in smart cities with NoSQL technologies. In *Transactions on Engineering Technologies*. Springer, 383–396.

Carlos E. Cuesta, Miguel A. Martínez-Prieto, and Javier D. Fernández. 2013. Towards an architecture for managing Big Semantic Data in Real-Time. In *Software Architecture*. Vol. 7957. Springer, Berlin, 45–53.

Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. 2013. Automating the database schema evolution process. *Int. J. VLDB* 22, 1 (2013), 73–98.

Alfredo Cuzzocrea, Rim Moussa, and Gianni Vercelli. 2018. An innovative lambda-architecture-based data warehouse maintenance framework for effective and efficient near-real-time OLAP over Big Data. In *Proceedings of the International Conference on Big Data*. Springer, 149–165.

R. Davenport. 2019. Big Companies Are Embracing Analytics, But Most Still Don't Have a Data-Driven Culture. Retrieved March 20, 2020 from https://hbr.org/2018/02/big-companies-are-embracing-analytics-but-most-still-dont-have-a-data-driven-culture.

Ali Davoudian. 2019. Helios: An adaptive and query workload-driven partitioning framework for distributed graph stores. In *Proceedings of the SIGMOD International Conference on Management of Data*. ACM, 1820–1822.

Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. A survey on NoSQL stores. *ACM Comput. Surv.* 51, 2 (2018), 40.

Junhua Ding, Xiaojun Kang, and Xin-Hua Hu. 2017. Validating a deep learning framework by metamorphic testing. In *Proceedings of the 2nd International Workshop on Metamorphic Testing (MET'17)*. IEEE, 28–34.

AnHai Doan, Alon Halevy, and Zachary Ives. 2012. *Principles of Data Integration*. Morgan Kaufmann.

Xin Luna Dong and Divesh Srivastava. 2015. *Big Data Integration. Synthesis Lectures on Data Management*. Morgan & Claypool.

Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Jianjun Zhao, and Yang Liu. 2018. DeepCruiser: Automated guided testing for stateful deep learning systems. *arXiv preprint arXiv:1812.05339* (2018).

Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. 2015. The BigDAWG polystore system. *ACM SIGMOD Rec.* 44, 2 (2015), 11–16.

A. D. Duncan. 2014. Focus on the 'Three Vs' of Big Data Analytics: Ariability, Veracity and Value. no. 25-11-2015, pp. To drive better analytic outcomes, business leader, 2016. [Online]. Available at https://www.gartner.com/doc/2921417/focus-vs-big-data-analytics.

Lisa Ehrlinger, Elisa Rusz, and Wolfram Wöß. 2019. A survey of data quality measurement and monitoring tools. *arXiv preprint arXiv:1907.08138* (2019).

Ahmed K. Elmagarmid, Marek Rusinkiewicz, Amit Sheth, and Amit Sheth. 1999. *Management of Heterogeneous and Autonomous Database Systems*. Morgan Kaufmann.

Aaron Elmore, Jennie Duggan, Mike Stonebraker, Magdalena Balazinska, Ugur Cetintemel, Vijay Gadepally, Jeffrey Heer, Bill Howe, Jeremy Kepner, Tim Kraska, et al. 2015. A demonstration of the BigDAWG polystore system. *Proc. VLDB Endow.* 8, 12 (2015), 1908–1911.

Hanif Eridaputra, Bayu Hendradjaya, and Wikan Danar Sunindyo. 2014. Modeling the requirements for Big Data application using goal oriented approach. In *Proceedings of the International Conference on Data and Software Engineering*. IEEE, 1–6.

Wenfei Fan, Xibei Jia, Jianzhong Li, and Shuai Ma. 2009. Reasoning about record matching rules. *Proc. VLDB Endow.* 2, 1 (2009), 407–418.

Raul Castro Fernandez, Peter R. Pietzuch, Jay Kreps, Neha Narkhede, Jun Rao, Joel Koshy, Dong Lin, Chris Riccomini, and Guozhang Wang. 2015. Liquid: Unifying nearline and offline Big Data integration. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'15)*.

Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath, Erik Witt, Eiko Yoneki, and Norbert Ritter. 2017. Quaestor: Query web caching for database-as-a-service providers. *Proc. VLDB Endow.* 10, 12 (2017), 1670–1681.

Corinna Giebler, Christoph Stach, Holger Schwarz, and Bernhard Mitschang. 2018. BRAID-A hybrid processing architecture for Big Data. In *Proceedings of the International Conference on Data Science, Technology, and Applications (DATA'18)*. 294–301.

Behzad Golshan, Alon Halevy, George Mihaila, and Wang-Chiew Tan. 2017. Data integration: After the teenage years. In *Proceedings of the 36th SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 101–106.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in Neural Information Processing Systems*. 2672–2680.

Ian Gorton and John Klein. 2015. Distribution, data, deployment: Software architecture convergence in Big Data systems. *IEEE Softw.* 32, 3 (2015), 78–85.

Christoph Gröger, Holger Schwarz, and Bernhard Mitschang. 2014. Prescriptive analytics for recommendation-based business process optimization. In *Proceedings of the International Conference on Business Information Systems*. Springer, 25–37.

Rihan Hai, Sandra Geisler, and Christoph Quix. 2016. Constance: An intelligent data lake system. In *Proceedings of the SIGMOD International Conference on Management of Data*. 2097–2100.

Alon Halevy, Anand Rajaraman, and Joann Ordille. 2006. Data integration: The teenage years. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB Endowment, 9–16.

Sang Hun Han, Kyoung Ok Kim, Eun Jong Cha, Kyung Ah Kim, and Ho Sun Shon. 2017. System framework for cardiovascular disease prediction based on Big Data technology. *Symmetry* 9, 12 (2017), 293.

Leonard Heilig and Stefan Voß. 2017. Managing cloud-based Big Data platforms: A reference architecture and cost perspective. In *Big Data Management*. Springer, 29–45.

Heli Hiisilä, Marjo Kauppinen, and Sari Kujala. 2016. An iterative process to connect business and IT development: Lessons learned. In *Proceedings of the 18th Conference on Business Informatics (CBI'16)*, Vol. 1. IEEE, 94–103.

H. Hu, Y. G. Wen, T.-S. Chua, and X. L. Li. 2014. Towards scalable systems for Big Data analytics: A technology tutorial. *IEEE Access* 2 (2014), 652–687.

IBM. 2017. How to leverage the power of prescriptive analytics to maximize the ROI. Retrieved May 6, 2019 from https://www.ibmbigdatahub.com/blog/how-leverage-power-prescriptive-analytics-maximize-roi.

Anne Immonen, Pekka Pääkkönen, and Eila Ovaska. 2015. Evaluating the quality of social media data in Big Data architecture. *IEEE Access* 3 (2015), 2028–2043.

Infochimps. 2013. CIOs & Big Data: What your IT team wants you to know. Retrieved Dec 21, 2018 from http://www.infochimps.com/resources/report-cios-big-data-what-your-it-team-wants-you-to-know-6/.

Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2015. Titian: Data provenance support in Spark. *Proc. VLDB Endow.* 9, 3 (2015), 216–227.

ISO. 2017. *Systems and Software Engineering-vocabulary*. Technical Report. ISO/IEC/IEEE 24765.

ISTQB. 2018. Standard Glossary of Terms used in Software Testing Version 3.2. Retrieved Jun 14, 2019 from https://www.istqb.org/downloads/category/20-istqb-glossary.html.

Petar Jovanovic, Óscar Romero Moral, Alkis Simitsis, Alberto Abelló Gamazo, Héctor Candón Arenas, and Sergi Nadal Francesch. 2015. Quarry: Digging up the gems of your data treasury. In *Proceedings of the 18th International Conference on Extending Database Technology*. 549–552.

Dawn N. Jutla, Peter Bodorik, and Sohail Ali. 2013. Engineering privacy for Big Data apps with the unified modeling language. In *Proceedings of the International Congress on Big Data*. IEEE, 38–45.

Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking distributed stream data processing systems. In *Proceedings of the 34th International Conference on Data Engineering (ICDE'18)*. IEEE, 1507–1518.

Vijay Khatri and Carol V. Brown. 2010. Designing data governance. *Commun. ACM* 53, 1 (2010), 148–152.

Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 1039–1049.

Michael Kläs, Wolfgang Putz, and Tobias Lutz. 2016. Quality evaluation for Big Data: A scalable assessment approach and first evaluation results. In *Proceedings of the Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA'16)*. IEEE, 115–124.

Jay Kreps. 2014. Questioning the Lambda architecture. Retrieved April 1, 2019 from https://www.oreilly.com/ideas/questioning-the-lambda-architecture.

Johannes Kroß, Andreas Brunnert, Christian Prehofer, Thomas A. Runkler, and Helmut Krcmar. 2015. Stream processing on demand for lambda architectures. In *Proceedings of the European Workshop on Performance Engineering*. Springer, 243–257.

Vijay Dipti Kumar and Paulo Alencar. 2016. Software engineering for Big Data projects: Domains, methodologies and gaps. In *Proceedings of the International Conference on Big Data (Big Data'16)*. IEEE, 2886–2895.

Rodrigo Laigner, Marcos Kalinowski, Sérgio Lifschitz, Rodrigo Salvador Monteiro, and Daniel de Oliveira. 2018. A systematic mapping of software engineering approaches to develop Big Data systems. In *Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'18)*. IEEE, 446–453.

Andreas Langegger, Wolfram Wöß, and Martin Blöchl. 2008. A semantic web middleware for virtual data integration on the web. In *Proceedings of the European Semantic Web Conference*. Springer, 493–507.

Lydia Lau, Fan Yang-Turner, and Nikos Karacapilidis. 2014. Requirements for Big Data analytics supporting decision making: A sensemaking perspective. In *Mastering Data-intensive Collaboration and Decision Making*. Springer, 49–70.

George Lee, Jimmy Lin, Chuang Liu, Andrew Lorek, and Dmitriy Ryaboy. 2012. The unified logging infrastructure for data analytics at Twitter. *Proc. VLDB Endow.* 5, 12 (2012), 1771–1780.

Maurizio Lenzerini. 2002. Data integration: A theoretical perspective. In *Proceedings of the 21st SIGMOD Symposium on Principles of Database Systems*. ACM, 233–246.

Katerina Lepenioti, Alexandros Bousdekis, Dimitris Apostolou, and Gregoris Mentzas. 2020. Prescriptive analytics: Literature review and research challenges. *Int. J. Inf. Manage.* 50 (2020), 57–70.

Jimmy Lin and Dmitriy Ryaboy. 2013. Scaling Big Data mining infrastructure: The Twitter experience. *ACM SIGKDD Explor. Newslett.* 14, 2 (2013), 6–19.

Jiaheng Lu and Irena Holubová. 2019. Multi-model databases: A new journey to handle the variety of data. *ACM Comput. Surv.* 52, 3 (2019), 55.

Ashwin Machanavajjhala and Jerome P. Reiter. 2012. Big privacy: Protecting confidentiality in Big Data. *XRDS* 19, 1 (2012), 20–23.

Nazim H. Madhavji, Andriy Miranskyy, and Kostas Kontogiannis. 2015. Big picture of Big Data software engineering: With example research challenges. In *Proceedings of the 1st International Workshop on Big Data Software Engineering*. IEEE Press, 11–14.

Gunasekaran Manogaran and Daphne Lopez. 2018. Disease surveillance system for big climate data processing and dengue transmission. In *Climate Change and Environmental Concerns: Breakthroughs in Research and Practice*. IGI Global, 427–446.

Miguel A. Martínez-Prieto, Carlos E. Cuesta, Mario Arias, and Javier D. Fernández. 2015. The solid architecture for real-time management of big semantic data. *Fut. Gener. Comput. Syst.* 47 (2015), 62–79.

Nathan Marz. 2011. How to beat the CAP theorem. Retrieved April 2, 2019 from http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.

Nathan Marz and James Warren. 2015. *Big Data: Principles and Best Practices of Scalable Real-time Data Systems*. Manning, New York, NY.

Richard McClatchey, Andrew Branson, Jetendr Shamdasani, Zsolt Kovacs, et al. 2015. Designing traceability into Big Data systems. *arXiv preprint arXiv:1502.01545* (2015).

John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, Michael Stonebraker, Kristin Tufte, and Hao Wang. 2015. S-Store: Streaming meets transaction processing. arXiv:1503.01143.

John Meehan, Stan Zdonik, Shaobo Tian, Yulong Tian, Nesime Tatbul, Adam Dziedzic, and Aaron Elmore. 2016. Integrating real-time and batch processing in a polystore. In *Proceedings of the High Performance Extreme Computing Conference (HPEC'16)*. IEEE, 1–7.

Jorge Merino, Ismael Caballero, Bibiano Rivas, Manuel Serrano, and Mario Piattini. 2016. A data quality in use model for Big Data. *Fut. Gener. Comput. Syst.* 63 (2016), 123–130.

Loup Meurice and Anthony Cleve. 2017. Supporting schema evolution in schema-less NoSQL data stores. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER'17)*. IEEE, 457–461.

Katina Michael and Keith W. Miller. 2013. Big Data: New opportunities and new challenges. *Computer* 46, 6 (2013), 22–24.

H. Gilbert Miller and Peter Mork. 2013. From data to decisions: A value chain for Big Data. *It Profess.* 15, 1 (2013), 57–59.

Seyed Esmaeil Mirvakili, MohammadAmin Fazli, and Jafar Habibi. 2019. Reactive liquid: Optimized liquid architecture for elastic and resilient distributed data processing. *arXiv preprint arXiv:1902.05968* (2019).

Mostafa Mirzaie, Behshid Behkamal, and Samad Paydar. 2019. Big Data quality: A systematic literature review and future research directions. *arXiv preprint arXiv:1904.05353* (2019).

Gilad Mishne, Jeff Dalton, Zhenghua Li, Aneesh Sharma, and Jimmy Lin. 2013. Fast data in the era of Big Data: Twitter's real-time related query suggestion architecture. In *Proceedings of the SIGMOD International Conference on Management of Data*. ACM, 1147–1158.

Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, et al. 2011. The open provenance model core specification (v1.1). *Fut. Gener. Comput. Syst.* 27, 6 (2011), 743–756.

Sergi Nadal. 2019. *Metadata-driven Data Integration*. Ph.D dissertation. The Polytechnic University of Catalonia.

Sergi Nadal, Victor Herrero, Oscar Romero, Alberto Abelló, Xavier Franch, Stijn Vansummeren, and Danilo Valerio. 2017. A software reference architecture for semantic-aware Big Data systems. *Inf. Softw. Technol.* 90 (2017), 75–92.

V. D. Nadkarni. 2020. Worldwide Big Data Technology and Services Forecast, 2016–2020. Retrieved March 20, 2020 from https://www.marketresearch.com/IDC-v2477/Worldwide-Big-Data-Technology-Services-10510864/.

Ravishankar Narayanan. 2016. Evolving and Improving the Requirements approach to Big Data Projects. https://re-magazine.ireb.org/articles/a-roadmap-to-implementing-big-data-projects/.

NIST. 2018. *NIST Big Data Interoperability Framework: Volume 3, Use Cases and General Requirements*. U.S. Department of Commerce, National Institute of Standards and Technology.

Ibtehal Noorwali, Darlan Arruda, and Nazim H. Madhavji. 2016. Understanding quality requirements in the context of Big Data systems. In *Proceedings of the 2nd International Workshop on Big Data Software Engineering*. ACM, 76–79.

Jukka K. Nurminen and Harrison Mfula. 2018. A unified framework for 5G network management tools. In *Proceedings of the 11th Conference on Service-Oriented Computing and Applications*. IEEE, 41–48.

Carlos Ordonez. 2010. Statistical model computation with UDFs. *IEEE Trans. Knowl. Data Eng.* 22, 12 (2010), 1752–1765.

Carlos E. Otero and Adrian Peter. 2014. Research directions for engineering Big Data analytics software. *IEEE Intell. Syst.* 30, 1 (2014), 13–19.

M. Tamer Özsu and Patrick Valduriez. 2011. *Principles of Distributed Database Systems*. Springer Science & Business Media.

Pasquale Pagano, Leonardo Candela, and Donatella Castelli. 2013. Data interoperability. *Data Sci. J.* 12 (2013), 119–125.

Leysia Palen, Kenneth M. Anderson, Gloria Mark, James Martin, Douglas Sicker, Martha Palmer, and Dirk Grunwald. 2010. A vision for technology-mediated support for public participation & assistance in mass emergencies & disasters. In *ACM-BCS Visions of Computer Science Conference*. British Computer Society, 8.

George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. 2020. Blocking and filtering techniques for entity resolution: A survey. *ACM Comput. Surv.* 53, 2 (2020), 1–42.

A. Patrizio. 2019. 4 reasons Big Data projects failand 4 ways to succeed: Nearly all Big Data projects end up in failure, despite all the mature technology available. Here's how to make Big Data efforts actually succeed. Retrieved March 21, 2020 from https://www.infoworld.com/article/3393467/4-reasons-big-data-projects-failand-4-ways-to-succeed.html.

Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 1–18.

Michele Pettinato, Juan Pablo Gil, Patricio Galeas, and Barbara Russo. 2019. Log mining to re-construct system behavior: An exploratory study on a large telescope system. *Inf. Softw. Technol.* 114 (2019), 121–136.

Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. 2008. Linking data to ontologies. *J. Data Semantics* 10 (2008), 133–173.

John Poole, Dan Chang, Douglas Tolbert, and David Mellor. 2002. *Common Warehouse Metamodel*. Vol. 20. John Wiley & Sons.

Christoph Quix, Rihan Hai, and Ivan Vatov. 2016. GEMMS: A generic and extensible metadata management system for data lakes. In *Proceedings of the CAiSE Forum*. 129–136.

Erhard Rahm and Hong Hai Do. 2000. Data cleaning: Problems and current approaches. *IEEE Data Eng.* 23, 4 (2000), 3–13.

Vaibhav Sachdeva and Lawrence Chung. 2017. Handling non-functional requirements for Big Data and IOT projects in scrum. In *Proceedings of the 7th International Conference on Cloud Computing, Data Science & Engineering-Confluence*. IEEE, 216–221.

Maribel Yasmina Santos, Jorge Oliveira e Sá, Carina Andrade, Francisca Vale Lima, Eduarda Costa, Carlos Costa, Bruno Martinho, and João Galvão. 2017. A Big Data system supporting Bosch Braga industry 4.0 strategy. *Int. J. Inf. Manage.* 37, 6 (2017), 750–760.

Carlton Sapp, Daren Brabham, Joseph Antelmi, Henry Cook, Thornton Craig, Soyeb Barot, Doreen Galli, Sumit Pal, Sanjeev Mohan, George Gilbert. 2018. Planning guide for data and analytics. [Online]. Available at https://www.gartner.com/en/doc/361501-2019-planning-guide-for-data-and-analytics [Accessed 29 May 2020].

Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Trans. Softw. Eng.* 42, 9 (2016), 805–824.

A. Sharala. 2019. Why 85% of Big Data projects fail. Retrieved March 21, 2020 from https://www.digitalnewsasia.com/insights/why-85-big-data-projects-fail.

Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and concurrent {RDF} queries with RDMA-based distributed graph exploration. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 317–332.

Hassan A. Sleiman and Rafael Corchuelo. 2012. Information extraction framework. In *Trends in Practical Applications of Agents and Multiagent Systems*. Springer, 149–156.

Sunil Soares. 2012. *Big Data Governance: An Emerging Imperative*. Mc Press.

Sunil Soares. 2013. *IBM InfoSphere: A Platform for Big Data Governance and Process Data Governance*. Mc Press.

Ian Sommerville and Pete Sawyer. 1997. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, Inc.

Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. 2019. *PRECRIME: Self-assessment Oracles for Anticipatory Testing*. Technical Report TR-Precrime-2019-02. USI Universita della Svizzera Italiana.

Christopher Stoermer, Felix Bachmann, and Chris Verhoef. 2003. *SACAM: The Software Architecture Comparison Analysis Method*. Technical Report. Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Isuru Suriarachchi and Beth Plale. 2016. Provenance as essential infrastructure for data lakes. In *Proceedings of the International Provenance and Annotation Workshop*. Springer, 178–182.

Ikbal Taleb, Hadeel T. El Kassabi, Mohamed Adel Serhani, Rachida Dssouli, and Chafik Bouhaddioui. 2016. Big data quality: A quality dimensions evaluation. In *Proceedings of the Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld'16)*. IEEE, 759–765.

Ignacio G. Terrizzano, Peter M. Schwarz, Mary Roth, and John E. Colino. 2015. Data wrangling: The challenging journey from the wild to the lake. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'15)*.

Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 303–314.

Trends. 2020. Interest in "Big Data Analytics'' over time. Retrieved May 2, 2020 from https://trends.google.pt/trends/explore?cat=12&date=2011-01-01%202020-02-04&q=big%20data%20analytics.

Chun-Wei Tsai, Chin-Feng Lai, Han-Chieh Chao, and Athanasios V. Vasilakos. 2015. Big Data analytics: A survey. *J. Big Data* 2, 1 (2015), 21.

R. Van Der Meulen. 2016. Gartner survey reveals investment in Big Data is up but fewer organizations plan to invest. [Online]. Available at http://www.gartner.com/newsroom/id/3466117 [Accessed 29 May 2020].

Maria-Esther Vidal, Kemele M. Endris, Samaneh Jozashoori, Farah Karim, and Guillermo Palma. 2019. Semantic data integration of big biomedical data for supporting personalised medicine. In *Current Trends in Semantic Web Technologies: Theory and Practice*. Springer, 25–56.

M. Villari, A. Celesti, M. Fazio, and A. Puliafito. 2014. AllJoyn Lambda: An architecture for the management of smart environments in IoT. In *Proceedings of the International Conference on Smart Computing Workshops*. 9–14.

Matthias Volk, Sascha Bosse, Dennis Bischoff, and Klaus Turowski. 2019. Decision-support for selecting Big Data reference architectures. In *Proceedings of the International Conference on Business Information Systems*. Springer, 3–17.

Coral Walker and Hassan Alrehamy. 2015. Personal data lake with data gravity pull. In *Proceedings of the 5th International Conference on Big Data and Cloud Computing*. IEEE, 160–167.

Siyuan Wang, Chang Lou, Rong Chen, and Haibo Chen. 2018. Fast and concurrent {RDF} queries using RDMA-assisted {GPU} graph exploration. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC'18)*. 651–664.

Wei Wang, Lei Fan, Pu Huang, and Hai Li. 2019. A new data processing architecture for multi-scenario applications in aviation manufacturing. *IEEE Access* 7 (2019), 83637–83650.

Elaine J. Weyuker. 1982. On testing non-testable programs. *Comput. J.* 25, 4 (1982), 465–470.

Gio Wiederhold. 1992. Mediators in the architecture of future information systems. *Computer* 25, 3 (1992), 38–49.

Fangjin Yang, Gian Merlino, Nelson Ray, Xavier Léauté, Himanshu Gupta, and Eric Tschetter. 2017. The RADStack: Open source Lambda architecture for interactive analytics. In *Proceedings of the 50th Hawaii International Conference on System Sciences*. 1703–1712.

Michal Young. 2008. *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons.

Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. 2017. Caching at the web scale. In *Proceedings of the 26th International Conference on World Wide Web Companion*. 909–912.

Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* (2020).

Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd International Conference on Automated Software Engineering*. ACM, 132–142.

Zhi Quan Zhou, Shaowen Xiang, and Tsong Yueh Chen. 2015. Metamorphic testing for software quality assessment: A study of search engines. *Trans. Softw. Eng.* 42, 3 (2015), 264–284.

Theo Zschörnig, Jonah Windolph, Robert Wehlitz, and Bogdan Franczyk. 2020. A cloud-based analytics-platform for user-centric Internet of Things domains—Prototype and performance evaluation. In *Proceedings of the 53rd Hawaii International Conference on System Sciences*.

Received July 2019; revised May 2020; accepted June 2020