On-demand Data Integration

Ph.D. Dissertation

Sergi Nadal Francesch

Dissertation submitted on XXX

Thesis submitted: XXX, 2019

Ph.D. Supervisors: Prof. Alberto Abelló Gamazo
Universitat Politècnica de Catalunya, BarcelonaTech, Spain
Prof. Oscar Romero Moral
Universitat Politècnica de Catalunya, BarcelonaTech, Spain
Prof. Stijn Vansummeren
Université Libre de Bruxelles, Brussels, Belgium

PhD Committee: Prof. George H. L. Fletcher, Eindhoven University of Technology, Eindhoven, Netherlands
Prof. Eduardo Mena, Universidad de Zaragoza, Zaragoza, Spain
Dr. Mahmoud Sakr, Université Libre de Bruxelles, Brussels, Belgium
Prof. Robert Wrembel, Poznan University of Technology, Poznan, Poland
Prof. Esteban Zimányi, Université Libre de Bruxelles, Brussels, Belgium

PhD Series: Barcelona School of Informatics, Universitat Politècnica de Catalunya, BarcelonaTech

Printed in Belgium, 2019

Curriculum Vitae

Sergi Nadal

Sergi Nadal graduated in Computer Science Engineering in June 2013 at Barcelona School of Informatics, Universitat Politècnica de Catalunya (UPC). In September 2013 he continued his education enrolling the Erasmus Mundus Joint Master Degree Programme in Information Technologies for Business Intelligence (IT4BI), which brought him to internationally study at Université Libre de Bruxelles (ULB) in Belgium, Université François Rabelais Tours (UFRT) in France, and back to UPC. In 2014 he obtained a scholarship from the Erasmus+ programme.

In September 2015, he decided to pursue his PhD studies as part of the Erasmus Mundus Joint Doctorate in Information Technologies for Business Intelligence – Doctoral College (IT4BI-DC), under the supervision of professors Alberto Abelló, Oscar Romero and Stijn Vansummeren. As part of his PhD studies, Sergi performed three short research stays at Université Libre de Bruxelles, his host university.

During his master and PhD period, he has been working on the problems related to data integration in data-intensive ecosystems. His research interests mainly fall in the broad area of data management. Some specific topics being data integration, data warehousing, distributed data processing, data stream processing and complex event processing.

During his PhD studies, he has participated in the H2020 SUPERSEDE project (SUpporting evolution and adaptation of PERsonalized Software by Exploiting contextual Data and End-user feedback). There, he has been member of the feedback and data analysis work package performing technical and research tasks.

Sergi, has also participated in teaching and advisory work as teaching assistant at UPC during his PhD studies. He was involved in database-related courses taught by the Department of Service and Information System Engineering, precisely: *BI Project* (master level, winter 2017), *Big Data Management* (master level, winter 2017 and spring 2018), *Databases* (bachelor level, winter 2018), and *Concepts for Specialized Databases* (bachelor level, winter 2018). Since 2016, he has also been teaching lab sessions in the *Big Data Management and Analytics* (BDMA) postgraduate and master programmes, at UPC School of Professional and Executive Development. He also participated in the hands-on sessions for the *Big Data Management in Brief* course in the Summer School of the Master in Statistics and Operations Research (MESIO UPC-UB). Besides that, he has also co-advised one master thesis in the topic of complex event processing.

While doing his PhD he has co-authored 9 peer-reviewed publications, including 3 journal papers, 2 research track full conference papers, 2 workshop papers, 1 tool demonstration, and 1 reference work.

Abstract

Data has an undoubtedly impact on society. Storing and processing large amounts of available data is currently one of the key success factors for an organization. In order to carry on these data exploitation tasks, organizations first perform data integration combining data from multiple sources to yield a unified view over them. Nonetheless, we are recently witnessing a change represented by huge and heterogeneous amounts of data. Indeed, 90% of the data in the world has been generated in the last two years. This requires revisiting the traditional integration assumptions to cope with new requirements posed by such data-intensive settings.

This PhD thesis aims to provide a novel framework for data integration in the context of data-intensive ecosystems, which entails dealing with vast amounts of heterogeneous data, from multiple sources and in their original format. To this end, we advocate for an integration process consisting of sequential activities governed by a shared repository of metadata. From an stewardship perspective, this activities are the deployment of a data integration architecture, followed by the population of such shared metadata. From a data consumption perspective, the activities are virtual and materialized data integration, the former an exploratory task and the latter a consolidation one. Following the proposed framework, we focus on providing contributions to each of the four activities. We begin proposing a software reference architecture for semantic-aware data-intensive systems. Such architecture is as a blueprint to deploy a stack of systems, with metadata a first-class citizen. Next, we propose a graph-based metadata model as formalism for metadata management. We put the focus on supporting schema and data source evolution, a predominant factor the heterogeneous sources at hand. For virtual integration, we propose query rewriting algorithms that rely on the previously proposed metadata model. We additionally consider semantic heterogeneities in the data sources, which the proposed algorithms are capable of automatically resolving. Finally, the thesis focuses on the materialized integration activity, and to this end, proposes a method to select intermediate results to materialize in data-intensive flows. Overall, the results of this thesis serve as contribution to the field of data integration in current data-intensive ecosystems.

Resum

The final submission will contain here the abstract in catalan

Résumé

The final submission will contain here the abstract in french

Acknowledgements

The final submission will contain here the acknowledgements

Contents

List of Figures

List of Tables

Thesis Details

Thesis Title:	On-demand Data Integration
Ph.D. Student:	Sergi Nadal Francesch
Supervisors:	Prof. Alberto Abelló Gamazo, Universitat Politècnica de Catalunya, BarcelonaTech, Spain (UPC co-supervisor)
	Prof. Oscar Romero Moral, Universitat Politècnica de Catalunya, BarcelonaTech, Spain (UPC co-supervisor)
	Prof. Stijn Vansummeren, Université Libre de Bruxelles, Brussels, Belgium (ULB supervisor)

The main body of this thesis consists of the following papers:

[1] A software reference architecture for semantic-aware Big Data systems. Sergi Nadal, Victor Herrero, Oscar Romero, Alberto Abelló, Xavier Franch, Stijn Vansummeren, Danilo Valerio. Information & Software Technology 90: 75-92 (2017).

[2] An Integration-Oriented Ontology to Govern Evolution in Big Data Ecosystems. Sergi Nadal, Oscar Romero, Alberto Abelló, Panos Vassiliadis, Stijn Vansummeren. International Workshop On Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP) (2017).

[3] An Integration-Oriented Ontology to Govern Evolution in Big Data Ecosystems. Sergi Nadal, Oscar Romero, Alberto Abelló, Panos Vassiliadis, Stijn Vansummeren. Information Systems 79: 3-19 (2019).

[4] On-demand Integration of Heterogeneous and Evolving Data Sources. Sergi Nadal, Alberto Abelló, Oscar Romero, Stijn Vansummeren, Panos Vassiliadis. Under submission for SIGMOD 2019.

[5] Intermediate Results Materialization Selection and Format for Data-Intensive Flows. Rana Faisal Munir, Sergi Nadal, Oscar Romero, Alberto

Abelló, Petar Jovanovic, Maik Thiele, Wolfgang Lehner. Fundamenta Informaticae (163, 2): 111-138 (2018).

In addition to the main papers, the following peer-reviewed publications have also been made or the student has participated in its development.

Conference articles.

[6] A Situational Approach for the Definition and Tailoring of a Data-Driven Software Evolution Method. Xavier Franch, Jolita Ralyté, Anna Perini, Alberto Abelló, David Ameller, Jesús Gorroñogoitia, Sergi Nadal, Marc Oriol, Norbert Seyff, Alberto Siena, Angelo Susi. CAiSE 2018: 603-618.

[7] FAME: Supporting Continuous Requirements Elicitation by Combining User Feedback and Monitoring. Marc Oriol, Melanie J. C. Stade, Farnaz Fotrousi, Sergi Nadal, Jovan Varga, Norbert Seyff, Alberto Abelló, Xavier Franch, Jordi Marco, Oleg Schmidt. RE 2018: 217-227.

Workshop papers.

[8] Big Data Management Challenges in SUPERSEDE. Sergi Nadal, Alberto Abelló, Oscar Romero, Jovan Varga. EuroPro 2017.

Tool demonstrations.

[9] MDM: Governing Evolution in Big Data Ecosystems. Sergi Nadal, Alberto Abelló, Oscar Romero, Stijn Vansummeren, Panos Vassiliadis. EDBT 2018: 682-685.

Reference works.

[10] Integration-Oriented Ontology. Sergi Nadal, Alberto Abelló. In: Sakr S., Zomaya A. (eds) Encyclopedia of Big Data Technologies. Springer, Cham.

This thesis has been submitted for assessment in partial fulfillment of the PhD degree. The thesis is based on the submitted or published scientific papers which are listed above. Parts of the papers are used directly or indirectly in the extended summary of the thesis.

Chapter 1

Introduction

1 Background and Motivation

The importance of data in today's society is unquestionable. A large portion of companies - those known as digital companies - base their business model on the collection, storage and analysis of any data relevant to their business. This philosophy implies a paradigm shift in the management of organisations' operations, and requires the digitalisation of all their business processes (e.g., creating information systems to interact with customers and suppliers such as websites, mobile applications or GPS systems, adding sensors to mechanical processes to monitor them, etc). While the digitalisation of an organisation is an arduous task, the data generated and collected can be analysed in order to yield important information for making business decisions. This has now been identified as a determinant and differentiating success factor that increases organisations' competitiveness [87].

Nowadays, a new kind of data-intensive systems that gather and analyse all kinds of data has emerged bringing new challenges for data management and analytics[1] [147, 81]. The most popular characterization of such systems is based on the three Vs: **volume** (digitalisation of some processes can generate large volumes of data), **variety** (from heterogeneous and evolving data sources) and **velocity** (in terms of potential arrival time and data processing in real time). To address them, these systems are based on two pillars: new architectures (mainly based on cloud computing and distributed data management), and new data models (such as documents, graphs, key-value and streams). While abundant research has yield mature tools to handle volume (e.g., distributed data storage and processing [125]) and velocity (e.g., data stream and complex event processing [18]), variety has been mostly overlooked.

[1]These are today referred as Big Data systems.

Indeed, the data variety challenge refers to the complexity of providing an on-demand integrated view over an heterogeneous and evolving set of data sources such that it conceptualizes the domain at hand. For example, consider a company organizing events. External data such as weather data or a calendar with public holidays may help to predict the attendance to events. Crossing data from diverse sources has been identified as a key success factor in data-intensive projects [21]. Ultimately, the data variety challenge aims to democratize the access to relevant sources of data so that data analysts can conduct richer (i.e., better contextualized) analysis without needing to be proficient in data management tasks. However, current solutions to tackle the data variety challenge require data analysts to perform complex IT tasks, making the access to such systems nowadays restricted to highly specialized technical profiles (i.e., the so called *data scientists*).

2 Data Integration

Information integration, or data integration, has been an active research area for decades. Succinctly, it consists of given a single query involving several data sources get a single answer. Data integration is a pervasive area in data management, with applications spanning the domain of business (e.g., to enable access to legacy systems or external services), science (e.g., to combine information from the hundreds of biomedical databases available) or the Web (e.g., to build a platform analysing and comparing prices for products) [39]. All such examples entail building a system capable of modeling multiple autonomous data sources, and provide a uniform query interface over them. Since its outset, data integration has been traditionally tackled in two independent forms: *materialized integration* or *virtual integration*.

The materialized approach, *data warehousing* being the most popular alternative, consists of extracting the content of the sources and physically materializing it in a structured repository [92]. Here, the integration task consists of defining a target database schema and a set of *procedural* mappings (i.e., *extract-transform-load* -ETL- processes) that periodically fetch and populate the target warehouse. This approach creates physical independence between the warehouse and the data sources, at the expense of freshness (i.e., out-of-date results), storage space and synchronization cost. Online analytical processing (OLAP) tools are the most well known representatives of materialized integration systems. Relying on a lattice structure (i.e., the data cube), data analysts obtain the required information dismissing the access to the sources. Such structures are also referred, in general, as materialized views.

Conversely, the virtualized approach defines a global schema, also known as *mediated* schema, such that queries posed over it are automatically translated to queries over the sources. Now, the integration task consists of defining

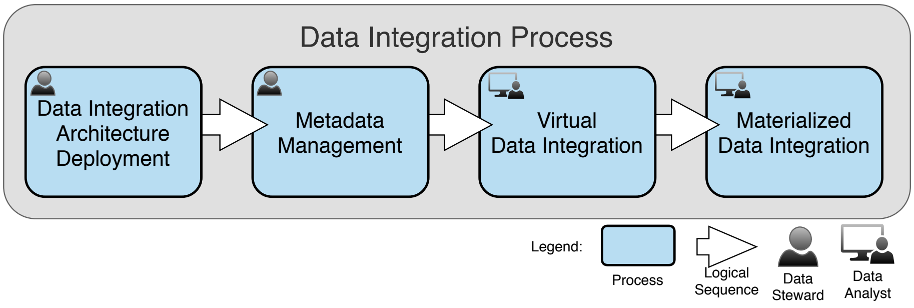


Fig. 1.1: The data integration process

declarative mappings that define relationships between the mediated schema and the sources. Such schema mappings are categorized as *global-as-view* (GAV), *local-as-view* (LAV), or the more general *tuple-generating dependencies* (GLAV); which directly determine how queries are processed. Rewriting queries in GAV, where concepts of the global schema are characterized in terms of queries over the sources, can be reduced to a simple unfolding process. Conversely, in the LAV setting, where the sources are characterized in terms of queries over the global schema, query rewriting generally becomes a complex reasoning task (likewise for GLAV). Thus, query answering consists of resolving such mapping assertions generating queries over the sources. This yields benefits for freshness, however at the expense of creating a dependence upon the availability of the sources.

Nonetheless, current data-intensive systems (i.e., those characterized with the three Vs) bring new challenges for data integration that require carefully rethinking how to deal with the traditional approaches [56]. Precisely, traditional data integration has focused on modeling well-defined domains with few structured data sources [71]. However, as characterized by the data variety challenge, we are now dealing with a scale of data sources growing to hundreds or thousands, each providing humongous amount of data in a variety of forms [26, 56, 146].

On the one hand, in this settings, the virtual approach can aid to easily navigate and explore the content of the sources with no cost for freshness, space and synchronization. However, efficiency when executing queries is now compromised on dealing with massive datasets. On the other hand, the materialized approach, which fails at the exploratory phase, can provide benefits on consolidating the materialization of relevant data and optimizing the execution of complex queries aimed at gaining deep insights into data. Thus, clearly combining both integration approaches (i.e., virtual and materialized) can bring benefits to tackle data integration in the context of nowadays data-intensive systems. This PhD thesis is motivated by this premise, thus we envision an end-to-end integration system where, after a metadata definition phase, data exploitation is performed first via virtual integration (i.e., exploration) followed by materialized integration (i.e., consolidation). We depict such process in Figure 1.1, which begins with the activities that define an architecture capable of interacting with metadata artifacts allowing to deal with the complexity entailed by variety. Note we distinguish two main actors that participate in the integration life-cycle (a) the *data steward*, who, similarly to the database administrator, is in charge of managing the integration system; and (b) the *data analyst*, who is the consumer of data. Next, we detail each of the phases composing the data integration life-cycle, where we also additionally highlight related open research problems that will drive the proposed contributions in this thesis.

2.1 Supporting end-to-end data integration

The cornerstone to perform data integration are metadata (i.e., data describing data). Relevant examples of metadata for data integration are: description of the sources, their schemata, or queries that will be posed over them (i.e., workload). The amount and kind of metadata available will have a direct impact on the degree of automation that can be achieved in further data transformation tasks. Thus, a general objective is the adoption of architectures that have means to represent metadata and the flows where those metadata are generated and consumed. Traditional data integration systems, depicted as *enterprise information integration* systems in industry [70], were commonly built on top of database management systems. Taking the relational data model as foundation, such architectures provides the core constructs to represent metadata for data integration (e.g., schemata or views).

Conversely, current technological stacks for the management and processing of data-intensive tasks are composed of independent components (commonly those in the NOSQL family [113]) that generally work in isolation and are orchestrated together to map to what would be equivalent to different functionalities of a database management system. A well-known example of this case is the Hadoop ecosystem [63]. This scenario requires manual orchestration of components, yielding ad hoc solutions that do not benefit from more general best practices in data integration (i.e., sharing metadata across components). This is a well-known problem of NOSQL repositories, which lack relevant semantics (i.e., metadata) due to their schemaless properties. This lack of metadata prevents the system from knowing which data are stored and how they interrelate. Thus, data analysts are hindered with data management tasks, like understanding the specific structure and parsing it, before writing their queries.

The previous discussion motivates the need to define a data-intensive integration architecture where metadata are accessible and shared throughout all its functional components. Such envisioned architecture is depicted in

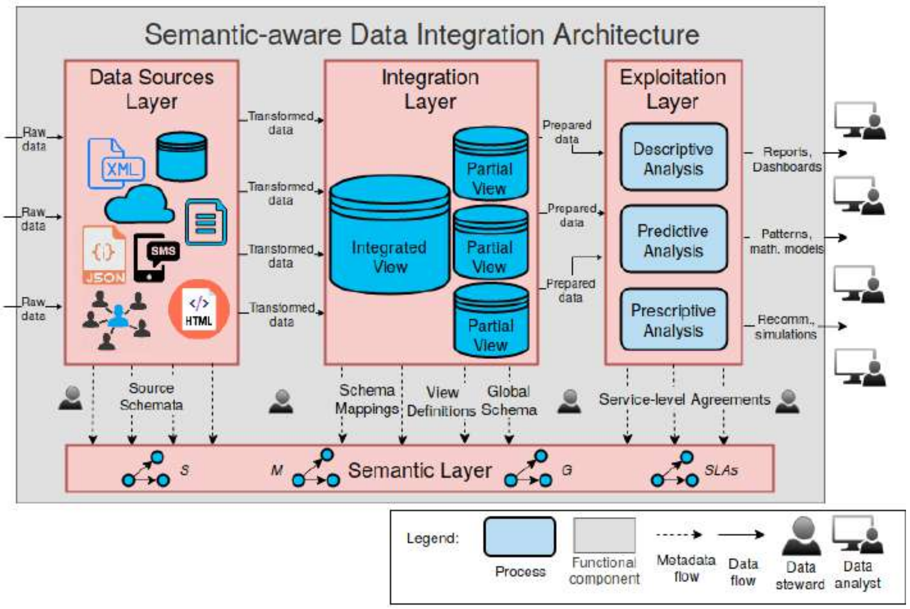


Fig. 1.2: High-level representation of a semantic-aware integration architecture

Figure 1.2 at a high abstraction level. Precisely, it is divided in three sequential layers where data are *(a)* ingested in its natural form from the sources; *(b)* consolidated in an integrated view and partial views (i.e., subsets of the integrated view targeted to groups of users); and *(c)* prepared for the specific data exploitation task. Processes in each layer generate and use metadata from such *Semantic Layer*, precisely in the figure we depict all metadata-generating flows. This, gives the means for data governance and automation of the following activities (i.e., virtual and materialized integration). The population and management of metadata in the semantic layer is a task supported by data stewards.

2.2 Virtual integration

Virtual data integration enables data analysts to perform exploratory tasks searching for particular insights of interest, a process enabled by the consumption of metadata from the semantic layer. As depicted in Figure 1.3, virtual integration consists of rewriting a query Q_G posed over a mediated (or global) schema (i.e., the domain metadata previously discussed) into an equivalent set of queries over the sources [27]. This amounts to the problem of answering queries using views [69]. As this activity directly accesses the sources, all tasks are carried out in the exploitation layer. Such activities precisely consist of rewriting Q_G using the global schema and mappings to a rewritten set of

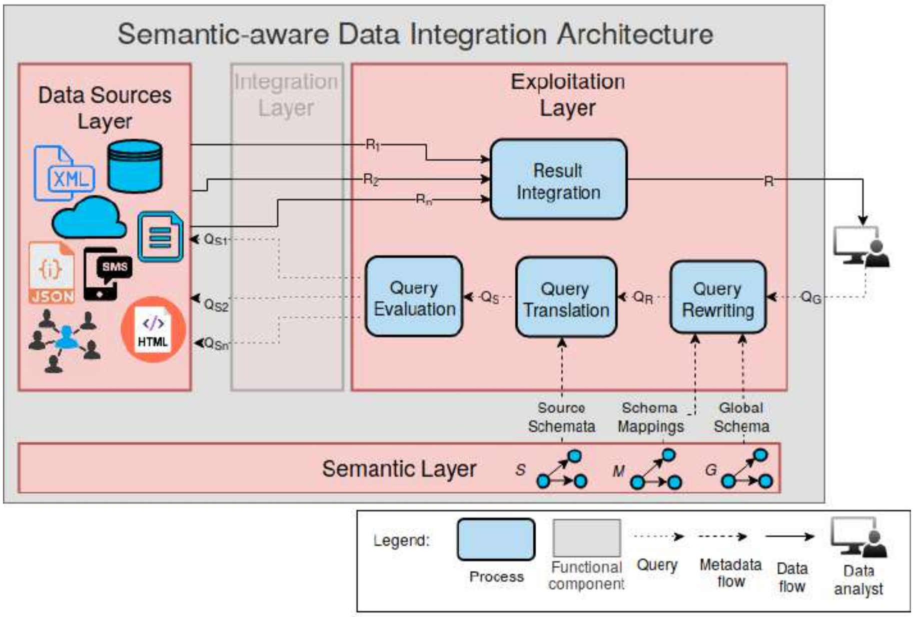


Fig. 1.3: Virtual integration within a semantic-aware data integration architecture

queries Q_R. Then, those queries are further translated, each to their native source language Q_S, using the source schemata, and evaluated. The returned results $R_1, \ldots, R_n$ are merged and integrated into a common structure R that the analyst receives.

2.3 Materialized integration

Once sources have been explored and the data analyst has identified the insights of interest, as virtual integration compromises computational complexity, it is time to materialize the subset of data used to compute such insights. Here, as for the virtual case, we also leverage metadata consumed from the semantic layer. Thus, this last activity consists of the consolidation of such exploratory queries into procedural mappings (i.e., ETL processes). These processes periodically extract, transform and load the desired information from the sources while meeting the specified quality requirements. Thus, here operations go beyond the capabilities of virtual data integration queries, now performing complex tasks such as data cleaning, computing user defined functions or running predictive tasks. While in classic data integration the vast majority of analysed data was transactional, the newly emerged data-intensive settings has replaced traditional ETL processes with much richer *data-intensive flows* (DIFs) [82]. MapReduce [38], Spark [170] or Flink [32] are exemplary

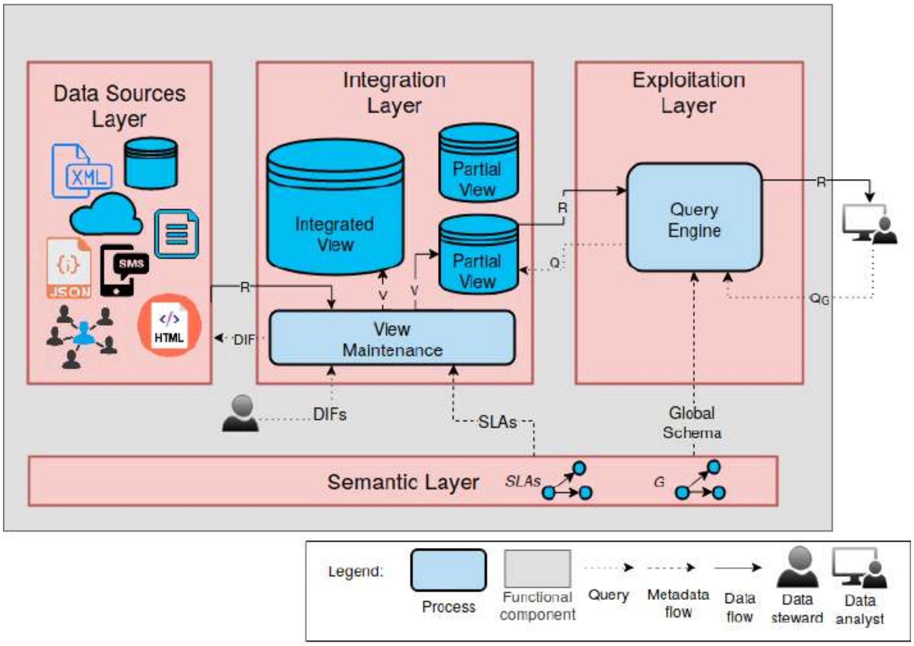


Fig. 1.4: Materialized integration within a semantic-aware data integration architecture

frameworks to implement such large scale DIFs.

Figure 1.4 depicts the materialized integration process, which is divided in two independent activities. *View maintenance* consists of periodically executing DIFs defined by data stewards that adhere to some service level agreements (SLAs). The execution of such DIFs yields resulting data R, which are incrementally included into the integrated and partial repository of views V. The second activity, which is triggered by the data analyst, consists of querying such integrated repositories containing transformed data. Thus, given a query over the global schema Q_G, the query engine transforms it and accesses the integration layer to yield the user the resulting data R.

2.4 Activities in data integration: state of the art and challenges

In this subsection we review the state of the art related to the data integration activities depicted in Figure 1.1. Here, we also highlight related open research problems that will drive the proposed contributions.

Data integration architecture deployment

The first activity composing the data integration process consists of deploying the software architecture encompassing the metadata artifacts for data integration. Such architecture must provide predefined flows to populate such metadata in order to automate the next activities. As previously discussed, in traditional data integration such architectures were based on relational database management systems. Prominent examples of such database architectures are the mediator/wrapper architecture [165], federated databases [141], peer-to-peer [37] or multi-databases [102]. All this examples contain the database catalog, which stores all relevant metadata for the system [80]. However, in the current landscape of tools to perform data-intensive tasks there is no such artifact as a metadata catalog, which hinders the automation of further integration activities. Furthermore, data-intensive architectures are complex, commonly spanning more than one product, and harnessing the collection, manipulation and expolitation of metadata as a whole. This is worsened by the vast number of available off-the-shelf tools for data-intensive architectures, as there are no existing architectural guidelines for their engineering considering the systematic management of semantic metadata [45, 106].

Hence, the first problem of interest in this thesis concerns the definition of a semantic-aware data-intensive integration architecture including predefined flows of metadata to support the automation of data exploitation.

Metadata management

Semantic Web technologies are nowadays the most popular approach to exchange self-describing linked data. Thus, they are well-suited to represent metadata for data integration. Given the simplicity and flexibility of semantic graphs (i.e., ontologies), they constitute an ideal tool to define a unified interface that models heterogeneous and autonomous sources. Besides *operational* metadata (e.g., schema), equally relevant are *domain* metadata which formalize the domain of interest. Indeed, the goal of an ontology is precisely to conceptualize the knowledge of a domain [64]. Such knowledge is commonly represented in terms of the Resource Description Framework (RDF) [166], which enables to automate its processing, and thus opens the door to exchange such information on the Web as Linked Data [25]. Therefore, a vast number of ontologies, or vocabularies, have been proposed to achieve common consensus when sharing data, such as the RDF Data Cube Vocabulary or the Data Catalog Vocabulary. Specific languages have been proposed in the RDF realm to define schema mappings, R2RML being a notorious example [36].

Nonetheless, such approaches fall short to represent complex relationships between the ontology and the data sources (e.g., LAV) in a virtual integration context. This is a necessary aspect to define the constructs to automate virtual data integration. Hence, alternative logical formalisms (e.g., datalog) must be adopted to define such mappings. However, the new data integration settings, where variety is a predominant factor, bring new challenges and account for novel techniques. Precisely, in this settings where *event data* generated by sensors, monitors or logs are highly predominant [65], it is common that different sources report data at different levels of generalization/specialization as well as aggregation/decomposition [125]. Additionally, due to the unprecedented growth in the number of data providers and how often they change, it is also necessary to reflect all such coexisting schema versions in the adopted metadata model. Thus, making more desirable to adopt LAV approaches that deal better with evolution. In this cases, the definition of specific models on top of semantic graphs (i.e., constraining the vocabulary) can aid on defining metadata models and mappings to support managing semantic heterogeneities and evolution.

Thus, the second problem of interest in this thesis concerns on providing new metadata artifacts that allow to represent variety and variability in the sources, while maintaining simplicity in schema mappings leveraging on semantic graphs and their formalisms.

Virtual data integration

A well-known approach is that of *ontology-based data access* (OBDA), based on the decoupling of extensional data (i.e., schema) in an ontology and intensional data in the sources. The most prominent OBDA approaches are based on generic reasoning in description logics (DLs) for query rewriting (see [130]). In this settings, the global schema is encoded in an *OWL2 QL* ontology [59], which is built upon the *DL-Lite* family of DLs. Those allow to represent conceptual models with polynomial cost for reasoning in the ontology [30]. This rewritings remain tractable as schema mappings follow the GAV approach. However, despite the flexibility on querying, the management of the sources is still a problem (magnified in such highly heterogeneous settings), as the variability in their content and structure (i.e., schema) would potentially entail reconsidering all existing mappings (a well known drawback in GAV).

Besides OBDA, there exist a variety of approaches that perform virtual integration based on LAV mappings. Precisely, some of the most prominent LAV mediation algorithms are the *bucket algorithm* [100], the *inverse rules algorithm* [42] and the *MiniCon algorithm* [131]. All these algorithms are datalog-based to yield sets of maximally-contained query rewritings. To this end, conjuncts in the body of datalog rules are considered subgoals that need to be isolately processed and further combined. How subgoals are resolved, and how rewritings are combined differs among each of them. Nonetheless, none of the discussed approaches (i.e., OBDA or LAV mediation algorithms) addresses the seamless management of semantic heterogeneities.

In a variety-centric integration environment, where hundreds of sources might provide data at multiple combinations of granularity levels, it is paramount to automatically aggregate all available data to this specific granularity.

Hence, the third problem of interest in this thesis is the definition of a new approach to the problem of answering queries using views under semantic heterogeneities as well as data source and schema evolution.

Materialized data integration

Complex DIFs may span throughout several areas of an organization and involve multiple sources. Indeed, a recent survey on large scale analytical workloads shows that user workloads have high temporal locality, as 80% of them will be reused by different stakeholders on the range of minutes to hours [35]. Clearly, reusing some of the intermediate results that are computed in such DIFs can highly increase the reusability and hence the performance of user workloads. This problem boils down to the classic problem of materialized view selection [72], a well known NP-hard problem [67]. The classic approaches to select materialized views in relational databases [69] have the single goal of improving the performance of executing queries, dismissing other relevant SLAs that are of interest for data-intensive applications. Examples of such SLAs are freshness, reliability or scalability [143]. There exists other, more recent, approaches to find the optimal partial materialization in DIFs [123, 44, 161]. Nonetheless, they are restricted to specific processing frameworks (i.e., MapReduce) and they generally focus on optimizing the system performance-wise ignoring other SLAs.

To this end, the fourth, and last, problem of interest in this thesis is that of selecting the optimal set of intermediate results to be reused from DIFs driven by metadata and SLAs.

3 Structure of the Thesis

The results of this PhD thesis are reported in the four main chapters of the document (i.e., Chapter 2 – Chapter 5). Each chapter is self-contained, corresponding to an individual or a collection of research papers. Thus, they can be read in isolation as each chapter adheres to the same structure providing related work for the topic, as well as concluding remarks. There might exist overlapping in concepts and examples given they were formulated in similar settings. This is specially the case of Chapters 3 and 4, which share an initial motivation but delve into different aspects of the integration process. Importantly, note that we refer to the same concept that denotes the proposed metadata structure using different terms. Specifically, in Chapter 3 we refer to it as *ontology* (where we put an emphasis on semantic graphs), while in

Chapter 4 we refer to it as graph (where we put an emphasis on the topology). Additionally, Appendix A refers to a published tool demonstration of our approach to virtual data integration.

The papers included in this thesis are listed below. Chapter 2 is based on Paper 1; Chapter 3 is based on Papers 2 and 3; Chapter 4 is based on Paper 4; Chapter 5 is based on Paper 5, and Appendix A is based on Paper 6.

1. A software reference architecture for semantic-aware Big Data systems. Sergi Nadal, Victor Herrero, Oscar Romero, Alberto Abelló, Xavier Franch, Stijn Vansummeren, Danilo Valerio. Information & Software Technology 90: 75-92 (2017).

2. An Integration-Oriented Ontology to Govern Evolution in Big Data Ecosystems. Sergi Nadal, Oscar Romero, Alberto Abelló, Panos Vassiliadis, Stijn Vansummeren. International Workshop On Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP) (2017).

3. An Integration-Oriented Ontology to Govern Evolution in Big Data Ecosystems. Sergi Nadal, Oscar Romero, Alberto Abelló, Panos Vassiliadis, Stijn Vansummeren. Information Systems 79: 3-19 (2019).

4. On-demand Integration of Heterogeneous and Evolving Data Sources. Sergi Nadal, Alberto Abelló, Oscar Romero, Stijn Vansummeren, Panos Vassiliadis. Under submission for ACM SIGMOD International Conference on Management of Data (2019).

5. Intermediate Results Materialization Selection and Format for Data-Intensive Flows. Rana Faisal Munir, Sergi Nadal, Oscar Romero, Alberto Abelló, Petar Jovanovic, Maik Thiele, Wolfgang Lehner. Fundamenta Informaticae (163, 2): 111-138 (2018).

6. MDM: Governing Evolution in Big Data Ecosystems. Sergi Nadal, Alberto Abelló, Oscar Romero, Stijn Vansummeren, Panos Vassiliadis. International Conference on Extending Database Technology (EDBT) 2018: 682-685.

4 Thesis Overview

This PhD thesis focuses on studying the current field of data integration, where some of the considered assumptions in the traditional setting [98] are no longer valid. In this section, we provide a brief overview of the results of this PhD thesis by discussing the contributions presented in each chapter.

4.1 Chapter 2: A software reference architecture for semantic-aware data-intensive systems

In Chapter 2, we study the problem of enabling metadata management and exploitation in current data-intensive software architectures. This chapter begins with a definition of sought requirements for such architecture. These requirements were obtained twofold, first we thoroughly reviewed the state of the art on software architectures with a focus on data-intensive architectures; second we obtained feedback from different industrial stakeholders in the domain. With that, a set of 15 requirements was defined, scattered across the 5 dimensions of volume, velocity, variety, variability and veracity. Next, we study the related work on data-intensive software architectures, which is performed via a systematic literature review checking whether the reviewed solutions fulfill the defined requirements. Precisely, we distinguish between custom architectures, which are software solutions tailored to a specific use case, and software reference architectures (SRAs), which are architectural blueprints tailored to a domain. As a result of this study we conclude that there nowadays exists two main families of architectures that cover part of the desired requirements. First, we encounter those presented as an evolution of the λ-architecture [111] (see Figure 1.5), succeeding to cover most of the volume and velocity requirements. Next, we find architectures based on Semantic Web principles, which analogously succeed to cover most of the variety, variability and veracity requirements. In general, we conclude that there is no existing architectural solution covering all desired requirements.

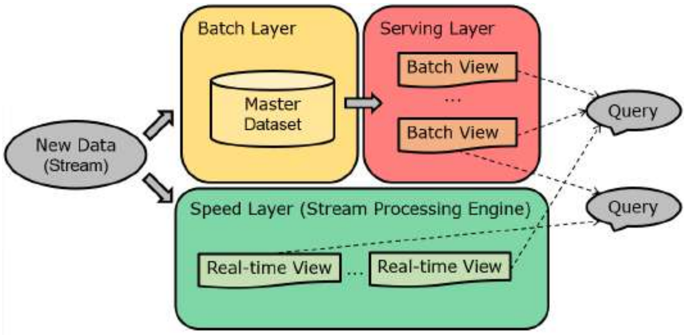


Fig. 1.5: λ-architecture

From the previous conclusion, a natural course of action is to propose a new architecture with the goal of covering all desired requirements. To this end, we propose *Bolster* (see Figure 1.6) a software reference architecture for semantic-aware data-intensive systems. *Bolster* adopts the best out of the two families of architectures. To this end, it extends the λ-architecture with a metadata management component that contains the necessary formalisms to represent metadata in a machine-readable format (e.g., RDF). As a second innovation, *Bolster* refines the λ-architecture by giving a precise definition of its components and their interconnections. Thus, the data steward's task is not the production of a new architecture from a set of independent components that need to be assembled. Instead, the data steward knows beforehand what type of components are needed and how they are interconnected. Therefore, his/her main responsibility is the selection of technologies for those components given the organizational requirements and structure.

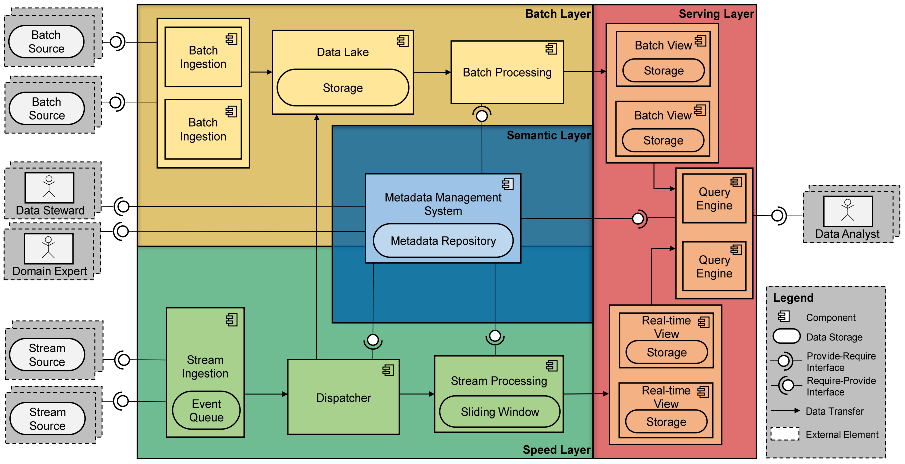


Fig. 1.6: *Bolster* SRA

Next, we provide an end-to-end illustrative example based on an online social network benchmark [173], where all components in *Bolster* interact. To aid the instantiation of *Bolster* we present a framework to select the right tool based on quality requirements from a set of candidate (here open source) tools. We base this (C)OTS (comercial off-the-shelf) selection problem [95] on the ISO/IEC 25000 SQuaRE standard (*Software Product Quality Requirements and Evaluation*) [79] as reference quality model. Finally, our work is concluded with the description of a set of industrial experiences where *Bolster* was successfully adopted. Precisely, we depict each specific instantiation as well as the results of a validation with the goal to assess to which extent *Bolster* lead to a perceived quality improvement in the software or service targeted in each use case.

4.2 Chapter 3: An integration-oriented ontology to govern evolution in data-intensive ecosystems

In Chapter 3, we tackle the second data integration activity. We first propose a flexible metadata model to deal with evolution. The ultimate goal is to build a metadata model that allows to represent how data sources and their schemata change, with the goal of enabling seamless virtual integration over it. We advocate for the adoption of graph formalisms (here semantic graphs) to represent an integration system $\mathcal{I}$ (i.e., the global schema, source schemata and mappings) [98]. Unlike other approaches that combine data structures with logical rules for mappings, using graphs we are capable of encoding all constructs composing $\mathcal{I}$ in a single data structure. This yields significant advantages on simplicity, as the algorithms using the structure need only to be concerned with a unique formalism to exploit metadata. Thus, $\mathcal{I}$ is composed of the three following elements:

1. *Global graph ($\mathcal{G}$).* Representing the domain of interest of analysts using relevant conceptual modeling constructs (i.e., association, specialization and aggregation). To this end we distinguish among *concepts* (i.e., classes) and *features* (i.e., class attributes). Furthermore we distinguish between those features that act as identifiers and those that do not.

2. *Source graph ($\mathcal{S}$).* Representing the schema of wrappers and their attributes. Adopted from the mediator-wrapper architecture, a wrapper consists of a view encoding a program or query hiding the complexity of accessing a data source [134].

3. *Mappings graph ($\mathcal{M}$).* Representing LAV schema mappings linked the global and source graphs. In logical terms, a LAV schema mapping is represented by a first-order formula of the form $\forall \overline{x}(R(\overline{x}) \rightarrow \exists y \psi(\overline{x}, \overline{y}))$, where R is an element (relation) in the source and ψ a query over the global schema. In the proposed graph-based representation LAV mappings are represented via subgraphs at the wrapper level. Furthermore, a surjective function $\mathcal{F} : \overline{a} \rightarrow \overline{f}$ links attributes in the wrappers to features. This is particularly relevant in heterogeneous integration settings where attribute names (externally defined in the wrappers) might differ from feature names (defined in the global graph).

Figure 1.7 depicts an overview of the proposed approach including the previously described components. To support source evolution, we present a method to semi-automatically include new wrappers in the source and mappings graphs.

Next, we present a method to semi-automatically include new wrappers in the source and mappings graphs. Finally, this chapter is complemented with

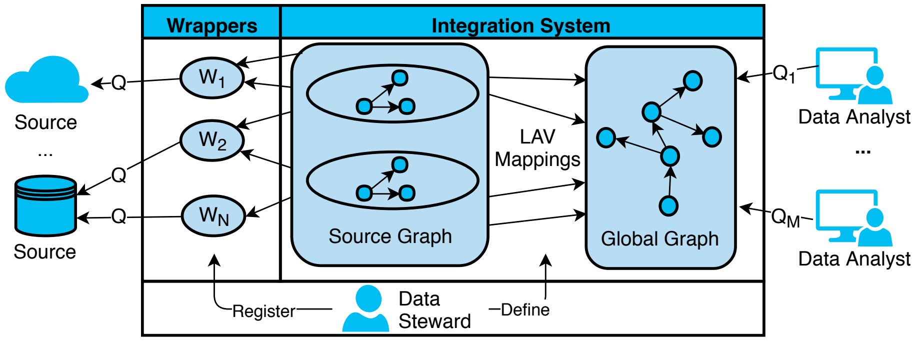


Fig. 1.7: High-level overview of the proposed integration system

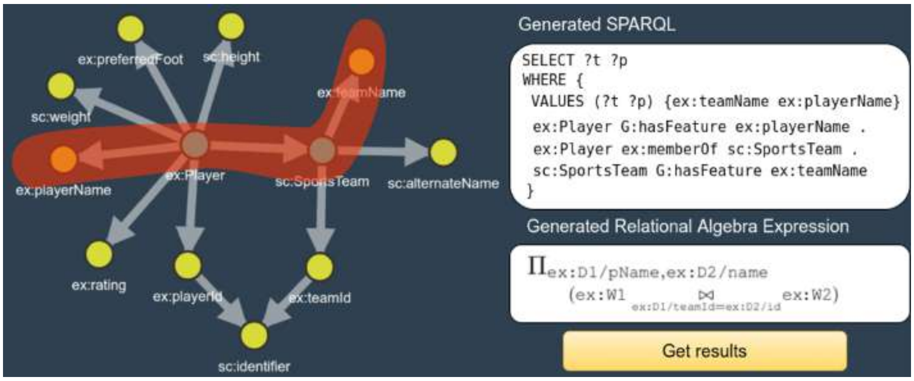
an evaluation of the proposed approach. Precisely, we perform a functional evaluation on the applicability of our approach w.r.t. the results of RESTful API evolution studies. Our evaluation results reveal that we are capable of semi-automatically accommodating all structural changes concerning data ingestion, which on average makes up 71.62% of the changes occurring on widely used APIs.

4.3 Chapter 4: Answering queries using views under semantic heterogeneities and evolution

This chapter presents our approach to query answering under semantic heterogeneities, focusing on specialization and aggregation (i.e., different granularity levels), as well as data source and schema evolution of the provided data. Precisely, leveraging on the previously introduced metadata model we present a query rewriting algorithm (i.e., REWRITECQ) that transforms a given query over $\mathcal{G}$ into a set of equivalent queries over the wrappers that include or discard semantically heterogeneous data sources, as well as perform implicit aggregations of data. Figure 1.8 depicts a high-level overview of the querying process (here implemented in SPARQL).

Given a query over the global graph $Q_{\mathcal{G}}$, REWRITECQ deals with the semantic heterogeneities of specialization and aggregation by generating sets of queries that request data at all available lower levels of granularity. This process, which is inspired by the bucket algorithm [100], leverages the semantic annotations in $\mathcal{G}$ to unambiguously resolve LAV mappings. This is achieved in two phases:

1. *Intra-concept generation.* Which receives as input a query (represented as a pattern in $\mathcal{G}$) and generates a graph of conjunctive queries. This graph, contains as nodes all concepts included in the pattern together with sets of conjunctive queries. Those indicate how to access the wrappers to



```
SELECT ?t ?p
WHERE {
 VALUES (?t ?p) {ex:teamName ex:playerName}
 ex:Player G:hasFeature ex:playerName .
 ex:Player ex:memberOf sc:SportsTeam .
 sc:SportsTeam G:hasFeature ex:teamName
}
```

$\Pi_{ex:D1/pName,ex:D2/name}(ex:W1 \bowtie_{ex:D1/teamId=ex:D2/id} ex:W2)$

Fig. 1.8: Example of a graph-based query rewriting

fetch the required features.

2. *Inter-concept generation.* Given the concept-centric graph of conjunctive queries, this phase deals with the discovery of equi join conditions among wrappers. To this end, we look for those intersecting LAV mappings (represented as subgraphs) to find shared identifier features. By systematically compacting the input graph, we compute a final set of rewritings (to be interpreted as a union of conjunctive queries) that represent all legal combinations of queries equivalent to $Q_{\mathcal{G}}$.

We theoretically show that REWRITECQ provides *minimally-sound* and *minimally-complete* rewritings.

To deal with semantic heterogeneities, we propose the aggregation graph $\mathcal{G}_{agg}$ as an analogy to an OLAP multidimensional lattice. $\mathcal{G}_{agg}$ is defined as a copy of $\mathcal{G}$ where all granularity levels at which wrappers provide data are explicitly materialized in hierarchies. Features are associated with their semantically valid aggregation functions, which shall be used to perform implicit aggregations in the querying answering process. To exploit such structure, we present REWRITECAQ an algorithm performing implicit aggregations of data to yield results at the requested granularity level by the analyst. For instance, if the data are required at the hourly level, but current wrappers provide it at the second or minute level, implicit aggregations should automatically be performed to produce the desired granularity. Intuitively, we define a virtual graph (i.e., $\mathcal{G}_{virtual}$), as a copy of $\mathcal{G}$, where implicit aggregate queries will be considered as new (virtual) wrappers. Once all virtual wrappers have been defined, we can evaluate $Q_{\mathcal{G}}$ over $\mathcal{G}_{virtual}$ to obtain a resulting union of conjunctive queries, where all data have been aggregated at the requested granularity. Next, we theoretically show that REWRITECAQ is also sound and complete.

This chapter is complemented with an extensive set of experiments where we positively show the practical behavior of the algorithms.

4.4 Chapter 5: SLA-driven selection of intermediate results to materialize

In this final chapter, we present our contribution to the last phase of the data integration life-cycle (i.e., materialized integration). As previously discussed, reusing intermediate results of a DIF's execution can potentially yield great benefits. To this end, in this chapter we revisit the traditional framework for materialized view selection [153] and analyse its applicability and extensions in DIFs. We present a method to select the optimal partial materialization of data from a DIF, driven by multiple quality objectives represented as SLAs. We apply well-known multi-objective optimization techniques, proven to efficiently tackle multiple and conflicting objectives. To assess different objectives, we introduce efficient cost estimation techniques leveraging on different data flow statistics gathered from the data sources and propagated over the DIF. Precisely, our method considers a set of design goals $\mathbb{DG}$, characterizing SLAs (computed by means of cost functions $\mathbb{CF}$) to minimize/maximize or constraints that should not be exceeded.

Due to the non-monotonicity of cost functions, purely greedy algorithms will not provide near-optimal results. Thus, we propose to adopt local search algorithms, consisting on the systematical modification of a given state by means of action functions in order to derive an improved solution state. Many complex techniques do exist for such approach (e.g., *simulated annealing* or *genetic algorithms*). The intricacy of these algorithms consists of their parametrization, which is at the same time their key performance aspect. Our proposal adopts *hill-climbing*, a non-parametrized search algorithm which can be seen as a greedy local search, always following the path that yields higher heuristic values. Since cost functions in DIFs are highly variable, due to their non-monotonicity, hill-climbing might provide different outputs depending on the initial state. In order to overcome such problem, we adopt a variant named *Shotgun hill-climbing* which consists of a hill-climbing with restarts. After certain number of iterations, we can obtain the most converging solution. Such approach of hill-climbing with restarts is surprisingly effective, specially when considering random initial states. We scrutinize the performance and quality of the algorithm with the proposed components.

5 Contributions

Figure 1.9 depicts a holistic view of the contributions of this PhD thesis within the data integration process.

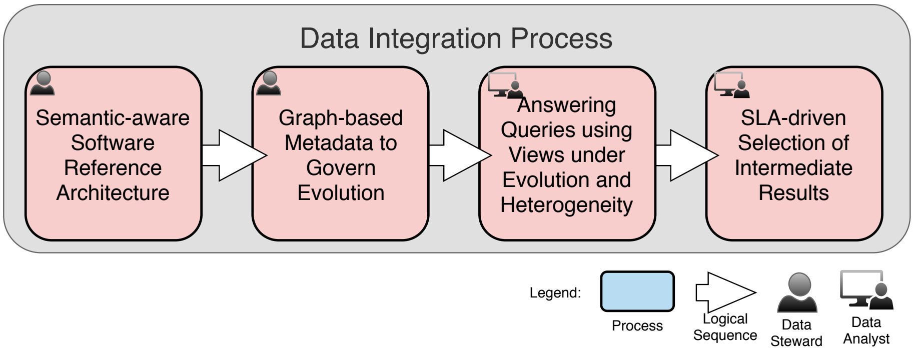

Data Integration Process
Semantic-aware Software Reference Architecture
Graph-based Metadata to Govern Evolution
Answering Queries using Views under Evolution and Heterogeneity
SLA-driven Selection of Intermediate Results
Legend:
Process
Logical Sequence
Data Steward
Data Analyst

Fig. 1.9: Contributions in the data integration process

The contributions of this PhD thesis are summarized as follows:

- *Data integration architecture.* We propose a software reference architecture for data-intensive systems. The architecture considers metadata as a first-class citizen by defining the *Metadata Management System* component. A detailed description of each component and their interaction allows data stewards to instantiate them with existing off-the-shelf tools. Engineering software architectures for data-intensive systems is nowadays an ad hoc task, thus our contribution brings novelty as it tackles the complex function of harmonizing and interconnecting different heterogeneous components towards a generic set of requirements.

- *Metadata management.* We propose a graph-based metadata model to support virtual data integration. Precisely, we encode in a graph all elements of an integration system (global schema, source schema and LAV mappings). The novelty of our contribution lies in the fact that we encode all integration constructs in a single data structure, thus simplifying the definition and exploitaiton of metadata. To exemplify this fact, we focus on the management of schema evolution presenting an algorithm to update the graph based on source changes, an aspect non commonly dealt with in the data integration literature.

- *Virtual integration.* We present a query rewriting algorithm that, given a query posed over the graph-based model, transforms it to an equivalent set of queries over the sources exploiting semantic annotations. The proposed algorithm automatically considers specialization relationships, which allow to prune the number of sources involved in a query, as well as navigations through featureless concepts. As an extension, we propose semantic annotations to deal with semantic heterogeneities and include data at lower granularity levels. To this end, we present a novel algorithm inspired by OLAP approaches, which rely on multidimensional lattices, that automatically generates sets of queries that perform implicit aggregations. This is a particularly relevant extension of the proposed integration graph, as currently such kind of semantic heterogeneities must be manually dealt with.

- *Materialized integration.* Lastly, we provide support to the reuse of intermediate results in DIFs. We propose a method that, given a set of SLAs, selects the optimal nodes to materialize. Our approach is novel with respect to traditional materialized view selection, where we select to materialize intermediate results of a DIF for a given set of weighted conflicting objectives.

Chapter 2

A Software Reference Architecture for Semantic-Aware Data-Intensive Systems

This chapter has been published as a paper in Information and Software Technology, 90: 75-92 (2017).
The layout of the papers has been revised
DOI: https://doi.org/10.1016/j.infsof.2017.06.001

Abstract

Context: *Big Data systems are a class of software systems that ingest, store, process and serve massive amounts of heterogeneous data, from multiple sources. Despite their undisputed impact in current society, their engineering is still in its infancy and companies find it difficult to adopt them due to their inherent complexity. Existing attempts to provide architectural guidelines for their engineering fail to take into account important Big Data characteristics, such as the management, evolution and quality of the data.*

Objective: *In this chapter, we follow software engineering principles to refine the λ-architecture, a reference model for Big Data systems, and use it as seed to create Bolster, a software reference architecture (SRA) for semantic-aware Big Data systems.*

Method: *By including a new layer into the λ-architecture, the Semantic Layer, Bolster is capable of handling the most representative Big Data characteristics (i.e., Volume, Velocity, Variety, Variability and Veracity).*

Results: *We present the successful implementation of Bolster in three industrial projects, involving five organizations. The validation results show high level of agreement among practitioners from all organizations with respect to standard quality factors.*

Conclusion: *As an SRA, Bolster allows organizations to design concrete architectures tailored to their specific needs. A distinguishing feature is that it provides* semantic-awareness *in Big Data Systems. These are Big Data system implementations that have components to simplify data definition and exploitation. In particular, they leverage metadata (i.e., data describing data) to enable (partial) automation of data exploitation and to aid the user in their decision making processes. This simplification supports the differentiation of responsibilities into cohesive roles enhancing data governance.*

1 Introduction

Major Big Data players, such as Google or Amazon, have developed large Big Data systems that align their business goals with complex data management and analysis. These companies exemplify an emerging paradigm shift towards data-driven organizations, where data are turned into valuable knowledge that becomes a key asset for their business. In spite of the inherent complexity of these systems, software engineering methods are still not widely adopted in their construction [57]. Instead, they are currently developed as ad hoc, complex architectural solutions that blend together several software components (usually coming from open-source projects) according to the system requirements.

An example is the Hadoop ecosystem. In Hadoop, lots of specialized Apache projects co-exist and it is up to Big Data system architects to select and orchestrate some of them to produce the desired result. This scenario, typical from immature technologies, raises high-entry barriers for non-expert players who struggle to deploy their own solutions overwhelmed by the amount of available and overlapping components. Furthermore, the complexity of the solutions currently produced requires an extremely high degree of specialization. The system end-user needs to be what is nowadays called a “data scientist”, a data analysis expert proficient in managing data stored in distributed systems to accommodate them to his/her analysis tasks. Thus, s/he needs to master two profiles that are clearly differentiated in traditional Business Intelligence (BI) settings: the data steward and the data analyst, the former responsible of data management and the latter of data analysis. Such combined profile is rare and subsequently entails an increment of costs and knowledge lock-in.

Since the current practice of ad hoc design when implementing Big Data systems is hence undesirable, improved software engineering approaches specialized for Big Data systems are required. In order to contribute towards this goal, we explore the notion of Software Reference Architecture (SRA) and present *Bolster*, an SRA for Big Data systems. SRAs are generic architectures for a class of software systems [10]. They are used as a foundation to derive software architectures adapted to the requirements of a particular organizational context. Therefore, they open the door to effective and efficient production of complex systems. Furthermore, in an emergent class of systems (such as Big Data systems), they make it possible to synthesize in a systematic way a consolidated solution from available knowledge. As a matter of fact, the detailed design of such a complex architecture has already been designated as a major Big Data software engineering research challenge [106, 45]. Well-known examples of SRAs include the AUTOSAR SRA [109] for the automotive industry, the Internet of Things Architecture (IoT-A) [164], an SRA for web browsers [62] and the NIST Cloud Computing Reference Architecture [103].

As an SRA, *Bolster* paves the road to the prescriptive development of software architectures that lie at the heart of every new Big Data system. Using *Bolster*, the work of the software architect is not to produce a new architecture from a set of independent components that need to be assembled. Instead, the software architect knows beforehand what type of components are needed and how they are interconnected. Therefore, his/her main responsibility is the selection of technologies for those components given the concrete requirements and the goals of the organization. *Bolster* is a step towards the homogenization and definition of a Big Data Management System (BDMS), as done in the past for Database Management Systems (DBMS) [53] and Distributed Database Management Systems (DDBMS) [125]. A distinguishing feature of *Bolster* is that it provides an SRA for *semantic-aware* Big Data Systems. These are Big Data system implementations that have components to simplify data definition and data exploitation. In particular, such type of systems leverage on metadata (i.e., data describing data) to enable (partial) automation of data exploitation and to aid the user in their decision making processes. This definition supports the differentiation of responsibilities into cohesive roles, the data steward and the data analyst, enhancing data governance.

Contributions The main contributions of this chapter are as follows:

- Taking as building blocks the five "V's" that define Big Data systems (see Section 2), we define the set of functional requirements sought in each to realize a semantic-aware Big Data architecture. Such requirements will further drive the design of *Bolster*.

- Aiming to study the related work on Big Data architectures, we perform a lightweight Systematic Literature Review. Its main outcome consists of the division of 21 works into two great families of Big Data architectures.

- We present *Bolster*, an SRA for semantic-aware Big Data systems. Combining principles from the two identified families, it succeeds on satisfying all the posed Big Data requirements. *Bolster* relies on the systematic use of semantic annotations to govern its data lifecycle, overcoming the shortcomings present in the studied architectures.

- We propose a framework to simplify the instantiation of *Bolster* to different Big Data ecosystems. For the sake of this chapter, we precisely focus on the components of the Apache Hadoop and Amazon Web Services (AWS) ecosystems.

- We detail the deployment of *Bolster* in three different industrial scenarios, showcasing how it adapts to their specific requirements. Furthermore, we provide the results of its validation after interviewing practitioners in such organizations.

Outline. The chapter is structured as follows. Section 2 introduces the Big Data dimensions and requirements sought. Section 3 presents the Systematic Literature Review. Sections 4, 5 and 6 detail the elements that compose *Bolster*, an exemplar case study implementing it and the proposed instantiation method respectively. Further, Sections 7 report the industrial deployments and validation. Finally, Section 8 wraps up the main conclusions derived from this work.

2 Big Data Definition and Dimensions

Big Data is a natural evolution of BI, and inherits its ultimate goal of transforming raw data into valuable knowledge. Nevertheless, traditional BI architectures, whose de-facto architectural standard is the Data Warehouse (DW), cannot be reused in Big Data settings. Indeed, the so-popular characterization of Big Data in terms of the three "V's (Volume, Velocity and Variety)" [81], refers to the inability of DW architectures, which typically rely on relational databases, to deal and adapt to such large, rapidly arriving and heterogeneous amounts of data. To overcome such limitations, Big Data architectures rely on NOSQL (Not Only SQL), co-relational database systems where the core data structure is not the relation [113], as their building blocks. Such systems propose new solutions to address the three V's by (i) distributing data and processing in a cluster (typically of commodity machines) and (ii) by introducing alternative data models. Most NOSQL systems distribute data (i.e., fragment and replicate it) in order to parallelize its processing while exploiting the data locality principle, ideally yielding a close-to-linear scale-up and speed-up [125]. As enunciated by the CAP theorem [28], distributed NOSQL systems must relax the well-known ACID (Atomicity, Consistency, Isolation, Durability) set of properties and the traditional concept of transaction to cope with large-scale distributed processing. As result, data consistency may be compromised but it enables the creation of fault-tolerant systems able to parallelize complex and time-consuming data processing tasks. Orthogonally, NOSQL systems also focus on new data models to reduce the impedance mismatch [60]. Graph, key-value or document-based modeling provide the needed flexibility to accommodate dynamic data evolution and overcome the traditional staticity of relational DWs. Such flexibility is many times acknowledged by referring to such systems as schemaless databases. These two premises entailed a complete rethought of the internal structures as well as the means to couple data analytics on top of such systems. Consequently, it also gave rise to the Small and Big Analytics concepts [147], which refer to performing traditional OLAP/Query&Reporting to gain quick insight into the data sets by means of descriptive analytics (i.e., Small Analytics) and Data Mining/Machine Learning to enable predictive analytics (i.e., Big Analytics) on Big Data systems, respectively.

In the last years, researchers and practitioners have widely extended the three "V's" definition of Big Data as new challenges appear. Among all existing definitions of Big Data, we claim that the real nature of Big Data can be covered by five of those "V's", namely: (a) Volume, (b) Velocity, (c) Variety, (d) Variability and (e) Veracity. Note that, in contrast to other works, we do not consider Value. Considering that any decision support system (DSS) is the result of a tightly coupled collaboration between business and IT [52], Value falls into the business side while the aforementioned dimensions focus on the IT side. In the rest of this chapter we refer to the above-mentioned "V's" also as Big Data dimensions.

In this section, we provide insights on each dimension as well as a list of linked requirements that we consider a Big Data architecture should fulfill. Such requirements were obtained in two ways: firstly inspired by reviewing related literature on Big Data requirements [51, 7, 137, 48, 34]; secondly they were validated and refined by informally discussing with the stakeholders from several industrial Big Data projects (see Section 7) and obtaining their feedback. Finally, a summary of devised requirements for each Big Data dimension is depicted in Table 2.1. Note that such list does not aim to provide an exhaustive set of requirements for Big Data architectures, but a high-level baseline on the main requirements any Big Data architecture should achieve to support each dimension.

2.1 Volume

Big Data has a tight connection with Volume, which refers to the large amount of digital information produced and stored in these systems, nowadays shifting from terabytes to petabytes (**R1.1**). The most widespread solution for Volume is data distribution and parallel processing, typically using cloud-based technologies. Descriptive analysis [140] (**R1.2**), such as reporting and OLAP, has shown to naturally adapt to distributed data management solutions. However, predictive and prescriptive analysis (**R1.3**) show higher-entry barriers to fit into such distributed solutions [156]. Classically, data analysts would dump a fragment of the DW in order to run statistical methods in specialized software, (e.g., R or SAS) [124]. However, this is clearly unfeasible in the presence of Volume, and thus typical predictive and prescriptive analysis methods must be rethought to run within the distributed infrastructure, exploiting the data locality principle [125].

2.2 Velocity

Velocity refers to the pace at which data are generated, ingested (i.e., dealt with the arrival of), and processed, usually in the range of milliseconds to seconds. This gave rise to the concept of data stream [18] and creates two main challenges. First, data stream ingestion, which relies on a sliding window buffering model to smooth arrival irregularities (**R2.1**). Second, data stream processing, which relies on linear or sublinear algorithms to provide near real-time analysis (**R2.2**).

2.3 Variety

Variety deals with the heterogeneity of data formats, paying special attention to semi-structured and unstructured external data (e.g., text from social networks, JSON/XML-formatted scrapped data, Internet of Things sensors, etc.) (**R3.1**). Aligned with it, the novel concept of Data Lake has emerged [152], a massive repository of data in its original format. Unlike DW that follows a *schema on-write* approach, Data Lake proposes to store data as they are produced without any preprocessing until it is clear how they are going to be analysed (**R3.2**), following the *load-first model-later* principle. The rationale behind a Data Lake is to store raw data and let the data analyst decide how to cook them. However, the extreme flexibility provided by the Data Lake is also its biggest flaw. The lack of schema prevents the system from knowing what is exactly stored and this burden is left on the data analyst shoulders (**R3.3**). Since loading is not that much of a challenge compared to the data transformations (*data curation*) to be done before exploiting the data, the Data Lake approach has received lots of criticism and the uncontrolled dump of data in the Data Lake is referred to as Data Swamp [148].

2.4 Variability

Variability is concerned with the evolving nature of ingested data, and how the system copes with such changes for data integration and exchange. In the relational model, mechanisms to handle evolution of *intension* (**R4.1**) (i.e., schema-based), and *extension* (**R4.2**) (i.e., instance-based) are provided. However, achieving so in Big Data systems entails an additional challenge due to the schemaless nature of NOSQL databases. Moreover, during the lifecycle of a Big Data-based application, data sources may also vary (e.g., including a new social network or because of an outage in a sensor grid). Therefore, mechanisms to handle data source evolution should also be present in a Big Data architecture (**R4.3**).

2.5 Veracity

Veracity has a tight connection with data quality, achieved by means of data governance protocols. Data governance concerns the set of processes and decisions to be made in order to provide an effective management of the data assets [91]. This is usually achieved by means of best practices. These can either be defined at the organization level, depicting the business domain knowledge, or at a generic level by data governance initiatives (e.g., Six Sigma [73]). However, such large and heterogeneous amount of data present in Big Data systems begs for the adoption of an automated data governance protocol, which we believe should include, but might not be limited to, the following elements:

- Data provenance (**R5.1**), related to how any piece of data can be tracked to the sources to reproduce its computation for lineage analysis. This requires storing metadata for all performed transformations into a common data model for further study or exchange (e.g., the Open Provenance Model [114]).

- Measurement of data quality (**R5.2**), providing metrics such as accuracy, completeness, soundness and timeliness, among others [20]. Tagging all data with such adornments prevents analysts from using low quality data that might lead to poor analysis outcomes (e.g., missing values for some data).

- Data liveliness (**R5.3**), leveraging on conversational metadata [152] which records when data are used and what is the outcome users experience from it. Contextual analysis techniques [17] can leverage such metadata in order to aid the user in future analytical tasks (e.g., query recommendation [55]).

- Data cleaning (**R5.4**), comprising a set of techniques to enhance data quality like standardization, deduplication, error localization or schema matching. Usually such activities are part of the preprocessing phase, however they can be introduced along the complete lifecycle. The degree of automation obtained here will vary depending on the required user interaction, for instance any entity resolution or profiling activity will infer better if user aided.

Including the aforementioned automated data governance elements into an architecture is a challenge, as they should not be intrusive. First, they should be transparent to developers and run as under the hood processes. Second, they should not overburden the overall system performance (e.g., [77] shows how automatic data provenance support entails a 30% overhead on performance).

2.6 Summary

The discussion above shows that current BI architectures (i.e., relying on RDMS), cannot be reused in Big Data scenarios. Such modern DSS must adopt

NOSQL tools to overcome the issues posed by Volume, Velocity and Variety. However, as discussed for Variability and Veracity, NOSQL does not satisfy key requirements that should be present in a mature DSS. Thus, *Bolster* is designed to completely satisfy the aforementioned set of requirements, summarized in Table 2.1.

Requirement	
1.	*Volume*
R1.1	The BDA shall provide scalable storage of massive data sets.
R1.2	The BDA shall be capable of supporting descriptive analytics.
R1.3	The BDA shall be capable of supporting predictive and prescriptive analytics.
2.	*Velocity*
R2.1	The BDA shall be capable of ingesting multiple, continuous, rapid, time varying data streams.
R2.2	The BDA shall be capable of processing data in a (near) real-time manner.
3.	*Variety*
R3.1	The BDA shall support ingestion of raw data (structured, semi-structured and unstructured).
R3.2	The BDA shall support storage of raw data (structured, semi-structured and unstructured).
R3.3	The BDA shall provide mechanisms to handle machine-readable schemas for all present data.
4.	*Variability*
R4.1	The BDA shall provide adaptation mechanisms to schema evolution.
R4.2	The BDA shall provide adaptation mechanisms to data evolution.
R4.3	The BDA shall provide mechanisms for automatic inclusion of new data sources.
5.	*Veracity*
R5.1	The BDA shall provide mechanisms for data provenance.
R5.2	The BDA shall provide mechanisms to measure data quality.
R5.3	The BDA shall provide mechanisms for tracing data liveliness.
R5.4	The BDA shall provide mechanisms for managing data cleaning.

Table 2.1: Requirements for a Big Data Architecture (BDA)

3 Related Work

In this section, we follow the principles and guidelines of Systematic Literature Reviews (SLR) as established in [94]. The purpose of this review is to systematically analyse the current landscape of Big Data architectures, with the goal to identify how they meet the devised requirements, and thus aid in the design of an SRA. Nonetheless, in this chapter we do not aim to perform an exhaustive review, but to depict, in a systematic manner, an overview on the landscape of Big Data architectures. To this end, we perform a lightweight SLR, where we focus on high quality works and evaluate them with respect to the previously devised requirements.

3.1 Selection of papers

The search was ranged from 2010 to 2016, as the first works on Big Data architectures appeared by then. The search engine selected was Scopus[1], as it indexes all journals with a JCR impact factor, as well as the most relevant conferences based on the CORE index[2]. We have searched papers with title, abstract or keywords matching the terms “big data” AND “architecture”. The list was further refined by selecting papers only in the “Computer Science” and “Engineering” subject areas and only documents in English. Finally, only conference papers, articles, book chapters and books were selected.

By applying the search protocol we obtained 1681 papers covering the search criteria. After a filter by title, 116 papers were kept. We further applied a filter by abstract in order to specifically remove works describing middlewares as part of a Big Data architecture (e.g., distributed storage or data stream management systems). This phase resulted in 44 selected papers. Finally, after reading them, sixteen papers were considered relevant to be included in this section. Furthermore, five non-indexed works considered grey literature were additionally added to the list, as considered relevant to depict the state of the practice in industry. The process was performed by our research team, and in case of contradictions a meeting was organized in order to reach consensus. Details of the search and filtering process are available at [116].

3.2 Analysis

In the following subsections, we analyse to which extent the selected Big Data architectures fulfill the requirements devised in Section 2. Each architecture is evaluated by checking whether it satisfies a given requirement (✓) or it does not (✗). Results are summarized in Table 2.2, where we make the distinction

[1]`http://www.scopus.com`

[2]`http://www.core.edu.au/conference-portal`

between custom architectures and SRAs. For the sake of readability, references to studied papers have been substituted for their position in Table 2.2.

Requirements on Volume

Most architectures are capable of dealing with storage of massive data sets (**R1.1**). However, we claim those relying on Semantic Web principles (i.e. storing RDF data), [A1,A8] cannot deal with such requirement as they are inherently limited by the storage capabilities of triplestores. Great effort is put on improving such capabilities [172], however no mature scalable solution is available in the W3C recommendations[3]. There is an exception to the previous discussion, as SHMR [A14] stores semantic data on HBase. However, this impacts its analytical capabilities with respect to those offered by triplestores. Oppositely, Liquid [A9] is the only case where no data are stored, offering only real-time support and thus not addressing the Volume dimension of Big Data. Regarding analytical capabilities, most architectures satisfy the descriptive level (**R1.2**) via SQL-like [A4,A10,A11,A18] or SPARQL [A1,A8] languages. Furthermore, those offering MapReduce or similar interfaces [A2,A3,A6,A13,A14,A15,A20] meet the predictive and prescriptive level (**R1.3**). HaoLap [A12] and SHMR [A14] are the only works where MapReduce is narrowed to descriptive queries.

Requirements on Velocity

Several architectures are capable of ingesting data streams (**R2.1**), either by dividing the architecture in specialized Batch and Real-time Layers [A2,A6,A7, A10,A11,A15,A20], by providing specific channels like data feeds [A4] or by solely considering streams as input type [A1,A8,A9]. Regarding processing of such data streams (**R2.2**), all architectures dealing with its ingestion can additionally perform processing, with the exception of AsterixDB [A4] and M3Data [A5], where data streams are stored prior to querying them.

Requirements on Variety

Variety is handled in diverse ways in the studied architectures. Concerning ingestion of raw data (**R3.1**), few proposals cannot deal with such requirement, either because they are narrowed to ingest specific data formats [A8,A16], or because specific wrappers need to be defined on the sources [A1,A19]. Concerning storage of raw data (**R3.2**), many architectures define views to merge and homogenize different formats into a common one (including those that do it at ingestion time) [A4,A5,A10,A12,A14,A15,A17]. On the other hand, the λ-architecture and some of the akin architectures [A2,A6,A7,A11] and

[3]https://www.w3.org/2001/sw/wiki/Category:Triple_Store

Custom Architectures		*Volume*			*Velocity*		*Variety*			*Variability*			*Veracity*			
		R1.1	R1.2	R1.3	R2.1	R2.2	R3.1	R3.2	R3.3	R4.1	R4.2	R4.3	R5.1	R5.2	R5.3	R5.4
A1	CQELS [129]	✗	✓	✗	✓	✓	✗	✗	✓	✓	✗	✓	✗	✗	✗	✗
A2	AllJoyn Lambda [160]	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
A3	CloudMan [132]	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
A4	AsterixDB [9]	✓	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✗	✗	✗
A5	M3Data [78]	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓
A6	[157]	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
A7	λ-arch. [111]	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
A8	Solid [110]	✗	✓	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
A9	Liquid [47]	✗	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗
A10	RADStack [169]	✓	✓	✗	✓	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓
A11	[97]	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
A12	HaoLap [145]	✓	✓	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
A13	[163]	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗	✗	✓	✓	✗	✓
A14	SHMR [66]	✓	✓	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
A15	Tengu [158]	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗	✓	✗	✗	✗	✗
A16	[168]	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗
A17	[43]	✓	✓	✓	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓
A18	D-Ocean [175]	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗

Software Reference Architectures		*Volume*			*Velocity*		*Variety*			*Variability*			*Veracity*			
		R1.1	R1.2	R1.3	R2.1	R2.2	R3.1	R3.2	R3.3	R4.1	R4.2	R4.3	R5.1	R5.2	R5.3	R5.4
A19	NIST [58]	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗	✓	✗	✓	✓	✓
A20	[126]	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓
A21	[54]	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
	Bolster	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 2.2: Fulfillment of each requirement in the related work

[A20] are the only ones natively storing raw data. In schema management (**R3.3**), all those architectures that favored ingesting and storing raw data cannot deal with such requirement, as no additional mechanism is present to handle it. Oppositely, the ones defining unified views are able to manage them, likewise relational database schemas. There is an exception to the previous discussion, D-Ocean [A18], which defines a data model for unstructured data, hence favouring all requirements.

Requirements on Variability

Requirements on Variability are poorly covered among the reviewed works. Schema evolution is only handled by CQELS [A1], AsterixDB [A4] and D-Ocean [A18]. CQELS uses specific wrapper configuration files which via a user interface map new elements to ontology concepts. On the other hand, AsterixDB parses schemas at runtime. Finally, D-Ocean's unstructured data model embraces the addition of new features. Furthermore, only AsterixDB considers data evolution (**R4.2**) using adaptive query processing techniques. With respect to automatic inclusion of data sources (**R4.3**), CQELS has a service allowing wrappers to be plugged at runtime. Moreover, other architectures provide such feature as AsterixDB with the definition of external tables at runtime, [A19] providing a discovery channel or Tengu [A15] by means of an Enterprise Service Bus.

Requirements on Veracity

Few of the studied architectures satisfy requirements on Veracity. All works covering data provenance (**R5.1**) log the operations applied on derived data in order to be reproduced later. On the other hand, measurement of data quality (**R5.2**) is only found in [A19] and [A13], the former by storing such metadata as part of its Big Data lifecycle and the latter by tracking data quality rules that validate the stored data. Regarding data liveliness (**R5.3**), [A16] tracks it in order to boost reusage of results computed by other users. Alternatively, [A19] as part of its Preservation Management activity applies aging strategies, however it is limited to its data retention policy. Finally, with respect to data cleaning (**R5.4**) we see two different architectures. In [A5,A13,A17,A19] cleansing processes are triggered as part of the data integration phase (i.e. before being stored). Differently, [A10,A20] execute such processes on unprocessed raw data before serving them to the user.

3.3 Discussion

Besides new technological proposals, we devise two main families of works in the Big Data architectures landscape. On the one hand, those presented as an evolution of the λ-architecture [A7] after refining it [A2,A6,A10,A11,A15]; and, on the other hand, those positioned on the Semantic Web principles [A1,A8]. Some architectures aim to be of general-purpose, while others are tailored to specific domains, such as: multimedia data [A14], cloud manufacturing [A3], scientific testing [A15], Internet of Things [A2] or healthcare [A13].

It can be concluded from Table 2.2 that requirements related to Volume, Velocity and Variety are more fulfilled with respect to those related to Variability and Veracity. This is due to the fact, to some extent, that Volume, Velocity and partly Variety (i.e., **R3.1**, **R3.2**) are core functionalities in NOSQL systems, and thus all architectures adopting them benefit from that. Furthermore, such dimensions have a clear impact on the performance of the system. Most of the architectures based on the λ-architecture naturally fulfil them for such reason. On the other hand, partly Variety (i.e., **R3.3**), Variability and Veracity are dimensions that need to be addressed by respectively considering evolution and data governance as first-class citizens. However, this fact has an impact on the architecture as a whole, and not on individual components, hence causing such low fulfiment across the studied works.

4 Bolster: a Semantic Extension for the λ-Architecture

In this section, we present *Bolster*, an SRA solution for Big Data systems that deals with the 5 "Vs". Briefly, *Bolster* adopts the best out of the two families of Big Data architectures (i.e., λ-architecture and those relying on Semantic Web principles). Building on top of the λ-architecture, it ensures the fulfillment of requirements related to Volume and Velocity. However, in contrast to other approaches, it is capable of completely handling Variety, Variability and Veracity leveraging on Semantic Web technologies to represent machine-readable metadata, oppositely to the studied Semantic Web-based architectures representing data. We first present the methodology used to design the SRA. Next, we present the conceptual view of the SRA and describe its components.

4.1 The design of *Bolster*

Bolster has been designed following the framework for the design of empirically-grounded reference architectures [50], which consists of a six-step process described as follows:

Step 1: decision on type of SRA. The first step is to decide the type of SRA to be designed, which is driven by its purpose. Using the characterization from [10], we conclude that *Bolster* should be of type 5 (a preliminary, facilitation architecture designed to be implemented in multiple organizations). This entails that the purpose of its design is to facilitate the design of Big Data systems, in multiple organizations and performed by a research-oriented team.

Step 2: selection of design strategy. There are two strategies to design SRAs, from scratch or from existing architectures. We will design *Bolster* based on the two families of Big Data architectures identified in Section 3.

Step 3: empirical acquisition of data. In this case, we leverage on the Big Data dimensions (the five "V's") discussed in Section 2 and the requirements defined for each of them. Such requirements, together with the design strategy, will drive the design of *Bolster*.

Step 4: construction of SRA. The rationale and construction of *Bolster* is depicted in Section 4.2, where a conceptual view is presented. A functional description of its components is later presented in Section 4.3, and a functional example in Section 5.

Step 5: enabling SRA with variability. The goal of enabling an SRA with variability is to facilitate its instantiation towards different use cases. To this end, we provide the annotated SRA using a conceptual view as well as the description of components, which can be selectively instantiated. Later, in Section 6, we present methods for its instantiation.

Step 6: evaluation of the SRA. The last step of the design of an SRA is its evaluation. Here, and leveraging on the industrial projects where *Bolster* has been adopted, in Section 7.2, we present the results of its validation.

4.2 Adding semantics to the λ-architecture

The λ-architecture is the most widespread framework for scalable and fault-tolerant processing of Big Data. Its goal is to enable efficient real-time data management and analysis by being divided into three layers (Figure 2.1).

- The *Batch Layer* stores a copy of the master data set in raw format as data are ingested. This layer also pre-computes *Batch Views* that are provided to the *Serving Layer*.
- The *Speed Layer* ingests and processes real-time data in form of streams. Results are then stored, indexed and published in *Real-time Views*.
- The *Serving Layer*, similarly as the *Speed Layer*, also stores, indexes and publishes data resulting from the *Batch Layer* processing in *Batch Views*.

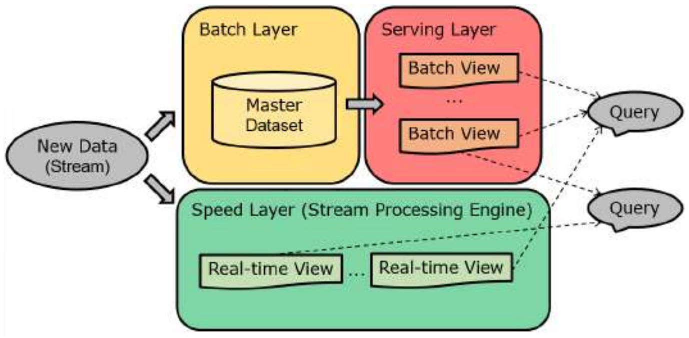


Fig. 2.1: λ-architecture

The λ-architecture succeeds at Volume requirements, as tons of heterogeneous raw data can be stored in the master data set, while fast querying through the Serving Layer. Velocity is also guaranteed thanks to the Speed Layer, since real-time views complement query results with real-time data. For these reasons, the λ-architecture was chosen as departing point for *Bolster*. Nevertheless, we identify two main drawbacks. First, as pointed out previously, it completely overlooks Variety, Variability and Veracity. Second, it suffers from a vague definition, hindering its instantiation. For example, the Batch Layer is a complex subsystem that needs to deal with data ingestion, storage and processing. However, as the λ-architecture does not define any further component of this layer, its instantiation still remains challenging. *Bolster* (Figure 2.2) addresses the two drawbacks identified in the λ-architecture:

- Variety, Variability and Veracity are considered first-class citizens. With this purpose, *Bolster* includes the Semantic Layer where the Metadata Repository stores machine-readable semantic annotations, in an analogous purpose as of the relational DBMS catalog.
- Inspired by the functional architecture of relational DBMSs, we refine the λ-architecture to facilitate its instantiation. These changes boil down to a precise definition of the components and their interconnections. We therefore introduce possible instantiations for each component by means of off-the-shell software or service.

Finally, note that this SRA aims to broadly cover different Big Data use cases, however it can be tailored by enabling or disabling components according to each particular context. In the following subsections we describe each layer present in *Bolster* as well as their interconnections. In bold, we highlight the necessary functionalities they need to implement to cope with the respective requirements.

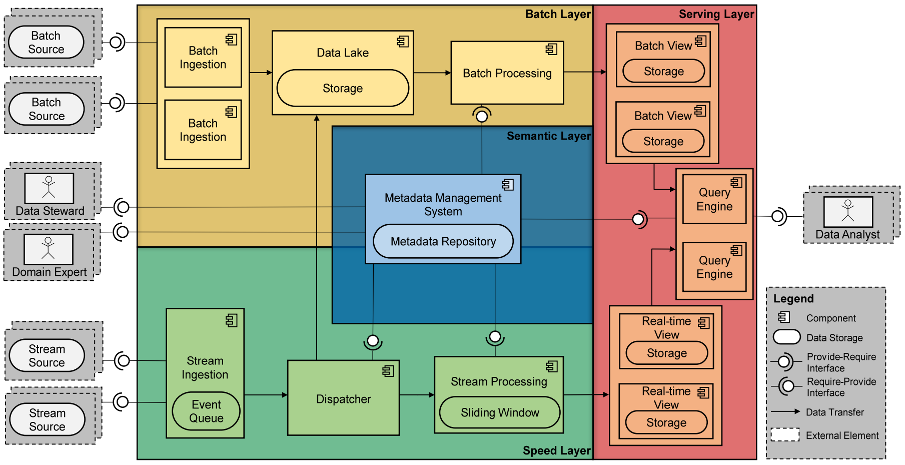


Fig. 2.2: *Bolster* SRA conceptual view

4.3 *Bolster* components

In this subsection, we present, for each layer composing *Bolster*, the list of its components and functional description.

Semantic Layer

The Semantic Layer (depicted blue in Figure 2.2) contains the Metadata Management System (MDM), the cornerstone for a semantic-aware Big Data system. It is responsible of providing the other components with the necessary information to describe and model raw data, as well as keeping the footprint about data usage. With this purpose, the MDM contains all the metadata artifacts, represented by means of RDF ontologies leveraging the benefits provided by Semantic Web technologies, needed to deal with data governance and assist data exploitation. We list below the main artifacts and refer the interested reader to [159, 24] for further details:

1. Data analysts should work using their day-by-day vocabulary. With this purpose, the **Domain Vocabulary** contains the business concepts (e.g., `customer`, `order`, `lineitem`) and their relationships (**R5.1**).

2. In order to free data analysts from data management tasks and decouple this role from the data steward, each vocabulary term must be mapped to the system views. Thus, the MDM must be aware of the **View Schemata** (**R3.3**) and the mappings between the vocabulary and such schemata.

3. Data analysts tend to repeat the same data preparation steps prior to conducting their analysis. To enable reusability and a collaborative exploitation of the data, on the one hand, the MDM must store **Pre-processing Domain Knowledge** about data preparation rules (e.g., data cleaning, discretization, etc.) related to a certain domain (**R5.4**), and on the other hand descriptive statistics to assess data evolution (**R4.2**).

4. To deal with automatic inclusion of new data sources (**R4.3**), each ingested element must be annotated with its schema information (**R4.1**). To this end, the **Data Source Register** tracks all input data sources together with the required information to parse them, the physical schema, and each schema element has to be linked to the attributes it populates, the logical schema (**R3.3**). Furthermore, for data provenance (**R5.1**), the **Data Transformations Log** has to keep track of the performed transformation steps to produce the views, the last processing step within the Big Data system.

Populating these artifacts is a challenge. Some of them can be automatically populated and some others must be manually annotated. Nonetheless, all of these artifacts are essential to enable a centralized master metadata management and hence, fulfil the requirements related to Variety, Variability and Veracity. Analogously to database systems, data stewards are responsible of populating and maintaining such artifacts. That is why we claim for the need that the MDM provides a user friendly interface to aid such processes. Finally, note that most of the present architectural components must be able to interact with the MDM, hence it is essential that it provides language-agnostic interfaces. Moreover, such interfaces cannot pose performance bottlenecks, as doing so would highly impact in the overall performance of the system.

Batch Layer

This layer (depicted yellow in Figure 2.2) is in charge of storing and processing massive volumes of data. In short, we first encounter Batch Ingestion, responsible for periodically ingesting data from the batch sources, then the Data Lake, capable of managing large amounts of data. The last step is the Batch Processing component, which prepares, transforms and runs iterative algorithms over the data stored in the Data Lake to shape them accordingly to the analytical needs of the use-case at hand.

Batch Ingestion. Batch sources are commonly big static raw data sets that require periodic synchronizations (**R3.1**). Examples of batch sources can be relational databases, structured files, etc. For this reason, we advocate for a multiple component instantiation, as required by the number of sources and type. These components need to know which data have already been moved to the Data Lake by means of **Incremental Bulks Scheduling and Orchestration**. The MDM then comes into play as it traces this information. Interaction between the ingestion components and the MDM occurs in a two-phase manner. First, they learn which data are already stored in the Data Lake, to identify the according incremental bulk can be identified. Second, the MDM is enriched with specific information regarding the recently brought data (**R5.3**). Since Big Data systems are multi-source by nature, the ingestion components must be built to guarantee its adaptability in the presence of new sources (**R4.3**).

Data Lake. This component is composed of a **Massive Storage** system (**R1.1**). Distributed file systems are naturally good candidates as they were born to hold large volumes of data in their source format (**R3.2**). One of their main drawbacks is that its read capabilities are only sequential and no complex querying is therefore feasible. Paradoxically, this turns out to be beneficial for the Batch Processing, as it exploits the power of cloud computing. Different file formats pursuing high performance capabilities are available, focusing on different types of workload [115]. They are commonly classified as horizontal, vertical and hybrid, in an analogous fashion as row-oriented and column-oriented databases, respectively.

Batch Processing. This component models and transforms the Data Lake's files into Batch Views ready for the analytical use-cases. It is responsible to schedule and execute **Batch Iterative Algorithms**, such as sorting, searching, indexing (**R1.2**) or more complex algorithms such as PageRank, Bayesian classification or genetic algorithms (**R1.3**). The processing components, must be designed to maximize reusability by creating building blocks (from the domain-knowledge metadata artifacts) that can be reused in several views. Consequently, in order to track **Batch Data Provenance**, all performed transformations must be communicated to the MDM (**R5.1**).

Batch processing is mostly represented by the MapReduce programming model. Its drawbacks appear twofold. On one hand, when processing huge amounts of batch data, several jobs may usually need to be chained so that more complex processing can be executed as a single one. On the other hand, intermediate results from Map to Reduce phases are physically stored in hard disk, completely detracting the Velocity (in terms of response time). Massive efforts are currently put on designing new solutions to overcome the issues posed by MapReduce. For instance, by natively including other more atomic relational algebra operations, connected by means of a directed acyclic graph; or by keeping intermediate results in main memory.

Speed Layer

The Speed Layer (depicted green in Figure 2.2) deals primarily with Velocity. Its input are continuous, unbounded streams of data with high timeliness and therefore require novel techniques to accommodate such arrival rate. Once ingested, data streams can be dispatched either to the Data Lake, in order to run historical queries or iterative algorithms, or to the Stream Processing engine, in charge of performing one-pass algorithms for real-time analysis.

Stream Ingestion. The Stream Ingestion component acts as a message queue for raw data streams that are pushed from the data sources (**R3.1**). Multiple sources can continuously push data streams (e.g., sensor or social network data), therefore such component must be able to cope with high throughput rates and scale according to the number of sources (**R2.1**). One of the key responsibilities is to enable the ingestion of all incoming data (i.e., adopt a **No Event Loss** policy). To this end, it relies on a distributed memory or disk-based storage buffer (i.e. event queue), where streams are temporarily stored.

This component does not require any knowledge about the data or schema of incoming data streams, however, for each event, it must know its source and type, for further matching with the MDM. To assure fault-tolerance and durability of results in such a distributed environment, techniques such as write-ahead logging or the two-phase commit protocol are used, nevertheless that has a clear impact on the availability of data to next components.

Dispatcher. The responsibilities of the Dispatcher are twofold. On the one hand, to ensure data quality, via MDM communication, it must register and validate that all ingested events follow the specified schema and rules for the event on hand (i.e., **Schema Typechecking** (**R4.1**, **R5.2**)). Error handling mechanisms must be triggered when an event is detected as invalid, and various mitigation plans can be applied. The simplest alternative is event rejection, however most conservative approaches like routing invalid events to the Data Lake for future reprocess can contribute to data integrity.

On the other hand, the second responsibility of the Dispatcher is to perform **Event Routing**, either to be processed in a real-time manner (i.e., to the Stream Processing component), or in a batch manner (i.e., to the Data Lake) for delayed process. In contrast to the λ-architecture, which duplicates all input streams to the Batch Layer, here only those that will be used by the processing components will be dispatched if required. Moreover, before dispatching such events, different routing strategies can influence the decision on where data is shipped, for instance by means of evaluating QoS cost models or analysing the system workload, as done in [97]. Other approaches like sampling or load shedding can be used here, to ensure that either real-time processing or Data

Lake ingestion are correctly performed.

Stream Processing. The Stream Processing component is responsible of performing **One-Pass Algorithms** over the stream of events. The presence of a summary is required as most of these algorithms leverage on in-memory stateful data structures (e.g., the Loosy Counting algorithm to compute heavy hitters, or HyperLogLog to compute distinct values). Such data structures can be leveraged to maintain aggregates over a sliding window for a certain period of time. Different processing strategies can be adopted, being the most popular tuple-at-a-time and micro-batch processing, the former providing low latency while the latter providing high throughput (**R2.2**). Similarly as the Batch Processing, this component must communicate to the MDM all transformations applied to populate Real-time Views in order to guarantee **Stream Data Provenance** (**R5.1**).

Serving Layer

The Serving Layer (depicted red in Figure 2.2) holds transformed data ready to be delivered to end-users (i.e. it acts as a set of database engines). Precisely, it is composed of Batch and Real-time Views repositories. Different alternatives exist when selecting each view engine, however as they impose a data model (e.g., relational or key-value), it is key to perform a goal-driven selection according to end-user analytical requirements [74]. It is worth noting that views can also be considered new sources, in case it is required to perform transformations among multiple data models, resembling a feedback loop. Further, the repository of Query Engines is the entry point for data analysts to achieve their analytical task, querying the views and the Semantic Layer.

Batch Views. As in the λ-architecture, we seek **Scalable and Fault-Tolerant Databases** capable to provide **Random Reads**, achieved by indexing, and the execution of **Aggregations and UDFs** (user defined functions) over large stable data sets (**R1.1**). The λ-architecture advocates for recomputing Batch Views every time a new version is available, however we claim incremental approaches should be adopted to avoid unnecessary writes and reduce processing latency. A common example of Batch View is a DW, commonly implemented in relational or columnar engines. However databases implementing other data models such as graph, key-value or documents also can serve the purpose of Batch Views. Each view must provide a high-level query language, serving as interface with the Query Engine (e.g., SQL), or a specific wrapper on top of it providing such functionalities.

Real-time Views. As opposite to Batch Views, Real-time Views need to provide **Low Latency Querying** over dynamic and continuously changing data sets (**R2.1**). In order to achieve so, in-memory databases are currently the most suitable option, as they dismiss the high cost it entails to retrieve data from disk. Additionally, Real-Time views should support low cost of updating in order to maintain **Sketches and Sliding Windows**. Finally, similarly to Batch Views, Real-time Views must provide mechanisms to be queried, considering as well **Continuous Query Languages**.

Query Engines. Query Engines, play a crucial role to enable efficiently querying the views in a friendly manner for the analytical task on hand. Data analysts query the system using the vocabulary terms and apply domain-knowledge rules on them (**R1.2, R1.3**). Thanks to the MDM artifacts, the system must internally perform the translation from **Business Requirements to Database Queries** over Batch and Real-time Views (**R3.3**), hence making data management tasks transparent to the end-user. Furthermore, the Query Engine must provide to the user the ability for **Metadata Query and Exploration** on what is stored in the MDM (**R5.1, R5.2, R5.3**).

Summary

Table 2.3 summarizes for each component the fulfilled requirements discussed in Section 2.

Component	*Volume*			*Velocity*		*Variety*			*Variability*			*Veracity*			
	R1.1	R1.2	R1.3	R2.1	R2.2	R3.1	R3.2	R3.3	R4.1	R4.2	R4.3	R5.1	R5.2	R5.3	R5.4
Metadata Management System								✓	✓	✓	✓	✓			✓
Batch Ingestion						✓					✓			✓	
Data Lake	✓						✓								
Batch Processing		✓	✓									✓			
Stream Ingestion				✓		✓									
Dispatcher									✓				✓		
Stream Processing					✓							✓			
Batch Views	✓														
Real-time Views				✓											
Query Engines		✓	✓					✓				✓	✓	✓	

Table 2.3: *Bolster* components and requirements fulfilled

5 Exemplar Use Case

The goal of this section is to provide an exemplar use case to illustrate how *Bolster* would accommodate a Big Data management and analytics scenario. Precisely, we consider the online social network benchmark described in [173]. Such benchmark aims to provide insights on the stream of data provided by Twitter's Streaming API, and is characterized by workloads in media, text, graph, activity and user analytics.

5.1 Semantic representation

Figure 2.3 depicts a high level excerpt of the content stored in the MDM. In dark and light blue, the domain knowledge and business vocabulary respectively which has been provided by the Domain Expert. In addition, the data steward has, possibly in a semi-automatic manner [119], registered a new source (Twitter Stream API[4]) and provided mappings for all JSON fields to the logical attributes (in red). For the sake of brevity, only the relevant subgraph of the ontology is shown. Importantly, to meet the Linked Open Data principles, this ontology should be further linked to other ontologies (e.g., the Open Provenance Model [114]).

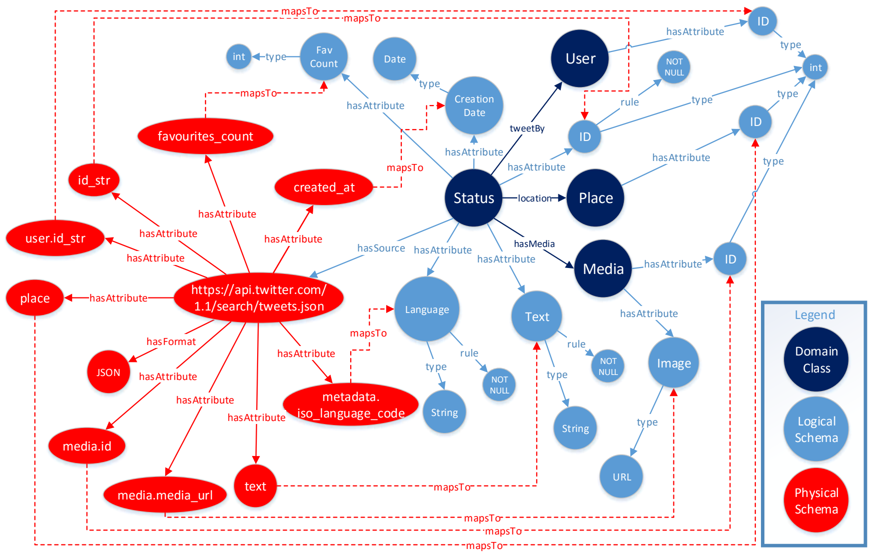


Fig. 2.3: Excerpt of the content in the Metadata Repository

5.2 Data ingestion

As raw JSON events are pushed to the Stream Ingestion component, they are temporary stored in the Event Queue. Once replicated, to guarantee durability and fault tolerance, they are made available to the Dispatcher, which is aware on how to retrieve and parse them by querying the MDM. Twitter's documentation[5] warns developers that events with missing counts rarely happen. To guarantee data quality such aspect must be checked. If an invalid event

[4]https://dev.twitter.com/streaming/overview
[5]https://dev.twitter.com/streaming/overview/processing

is detected, it should be discarded. After this validation, the event at hand must be registered in the MDM to guarantee lineage analysis. Furthermore the Dispatcher sends the raw JSON event to the Stream Processing and Data Lake components. At this point, there is a last ingestion step missing before processing data. The first workload presented in the benchmark concerns media analytics, however as depicted in Figure 2.3, the API only provides the URL of the image. Hence, it is necessary to schedule a batch process periodically fetching such remote images and loading them into the Data Lake.

5.3 Data processing and analysis

Once all data are available to be processed in both Speed and Batch Layers, we can start executing the required workloads. Many of such workloads concern predictive analysis (e.g., topic modeling, sentiment analysis, location prediction or collaborative filtering). Hence, the proposed approach is to periodically refresh statistical models in an offline manner (i.e., in the Batch Layer), in order to assess predictions in an online manner (i.e., in the Speed Layer). We distinguish between those algorithms generating metadata (e.g., Latent Dirichlet Allocation (LDA)) and those generating data (e.g., PageRank). The former will store its results in the MDM using a comprehensive vocabulary (e.g., OntoDM [127]); and the latter will store them into Batch Views. Once events have been dispatched, the required statistical model has to be retrieved from the MDM to assess predictions and store outcomes into Real-time Views. Finally, as described in [173], the prototype application provides insights based on tweets related to companies in the S&P 100 index. Leveraging on the MDM, the Query Engine is capable of generating queries to Batch and Real-time Views.

6 *Bolster* Instantiation

In this section we list a set of candidate tools, with special focus on the Apache Hadoop and Amazon Web Services ecosystems, to instantiate each component in *Bolster*. In the case when few tools from such ecosystems were available, we propose other available commercial tools which were considered in the industrial projects where *Bolster* was instantiated. Further, we present a method to instantiate the reference architecture. To this end, we propose a systematic scoring process driven by standard software product quality characteristics. The result of using our method yields, for each component, the most suitable tool.

6.1 Available tools

Semantic Layer

Metadata Management System. Two different off-the-shelf open source products can instantiate this layer, namely *Apache Stanbol*[6] and *Apache Atlas*[7]. Nevertheless, the features of the former fall short for the proposed requirements of the MDM. Not surprisingly, this is due to the novel nature of *Bolster*'s Semantic Layer. *Apache Atlas* satisfies the required functionalities more naturally and it might appear as a better choice, however it is currently under heavy development as an *Apache Incubator* project. Commercial tools such as *Cloudera Navigator*[8] or *Palantir*[9] are also candidate tools.

Metadata Storage. We advocate for the adoption of Semantic Web storage technologies (i.e. triplestores), to store all the metadata artifacts. Even though such tools allow storing and reasoning over large and complex ontologies, that is not the pursued purpose here, as our aim is to allow a simple and flexible representation of machine-readable schemas. That is why triplestores serve better the purpose of such storage. *Virtuoso*[10] is at the moment the most mature triplestore platform, however other options are available such as *4store*[11] or *GraphDB*[12]. Nonetheless, given the graph nature of triples, any graph database can as well serve the purpose of metadata storage (e.g., *AllegroGraph*[13] or *Neo4j*[14]).

Batch Layer

Batch Ingestion. This components highly depends on the format of the data sources, hence it is complex to derive a universal driver due to technological heterogeneity. Instantiating this component usually means developing ad hoc scripting solutions adapting to the data sources as well as enabling communication with the MDM. Massive data transfer protocols such as FTP or Hadoop's *copyFromLocal*[15] will complement such scripts. However, some drivers for specific protocols exist such as *Apache Sqoop*[16], the most widespread solution to load data from/to relational sources through JDBC drivers.

[6]https://stanbol.apache.org
[7]http://atlas.incubator.apache.org
[8]https://www.cloudera.com/products/cloudera-navigator.html
[9]https://www.palantir.com
[10]http://virtuoso.openlinksw.com
[11]http://4store.org
[12]http://graphdb.ontotext.com/graphdb
[13]http://allegrograph.com
[14]http://neo4j.com
[15]https://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-common/FileSystemShell.html#copyFromLocal
[16]http://sqoop.apache.org

Data Lake. *Hadoop Distributed File System* and *Amazon S3*[17] perfectly fit in this category, as they are essentially file systems storing plain files. Regarding data file formats, some current popular options are *Apache Avro*[18], *Yahoo Zebra*[19] or *Apache Parquet*[20] for horizontal, vertical and hybrid fragmentation respectively.

Batch Processing. *Apache MapReduce*[21] and *Amazon Elastic MapReduce*[22] are nowadays the most popular solutions. Alternatively, *Apache Spark*[23] and *Apache Flink*[24] are gaining great popularity as next generation replacement for the MapReduce model. However, to the best of our knowledge, only *Quarry* [84] is capable to interact with the MDM and, based on the information there stored, automatically produce batch processes based on user-defined information requirements.

Speed Layer

Stream Ingestion. All tools in the family of "message queues" are candidates to serve as component for Stream Ingestion. Originated with the purpose of serving as middleware to support enterprise messaging across heterogeneous systems, they have been enhanced with scalability mechanisms to handle high ingestion rates preserving durability of data. Some examples of such systems are *Apache ActiveMQ*[25] or *RabbitMQ*[26]. However, some other tools were born following similar principles but aiming Big Data systems since its inception, being *Apache Kafka*[27] and *AWS Kinesis Firehose*[28] the most popular options.

Dispatcher. Here we look for tools that allow developers to define data pipelines routing data streams to multiple and heterogeneous destinations. It should also allow the developer to programmatically communicate with the MDM for quality checks. *Apache Flume*[29] and *Amazon Kinesis Streams*[30] are nowadays the most prevalent solutions.

[17] `https://aws.amazon.com/s3`
[18] `https://avro.apache.org`
[19] `http://pig.apache.org/docs/r0.9.1/zebra_overview.html`
[20] `https://parquet.apache.org`
[21] `https://hadoop.apache.org`
[22] `https://aws.amazon.com/elasticmapreduce`
[23] `http://spark.apache.org`
[24] `https://flink.apache.org`
[25] `http://activemq.apache.org`
[26] `https://www.rabbitmq.com`
[27] `http://kafka.apache.org`
[28] `https://aws.amazon.com/kinesis/firehose`
[29] `https://flume.apache.org`
[30] `https://aws.amazon.com/kinesis/streams`

Stream Processing. In contrast to Batch Processing, it is unfeasible to adopt classical MapReduce solutions considering the performance impact they yield. Thus, in-memory distributed stream processing solutions like *Apache Spark Streaming*[31], *Apache Flink Streaming*[32] and *Amazon Kinesis Analytics*[33] are the most common alternatives.

Serving Layer

Batch Views. A vast range of solutions are available to hold specialized views. We distinguish among three families of databases: (distributed) relational, NOSQL and NewSQL. The former is mostly represented by major vendors who evolved their traditional centralized databases into distributed ones seeking to improve its storage and performance capabilities. Some common solutions are *Oracle*[34], *Postgres-XL*[35] or *MySQL Cluster*[36]. Secondly, in the NOSQL category we might drill-down to the specific data model implemented: *Apache HBase*[37] or *Apache Cassandra*[38] for column-family key-value; *Amazon DynamoDB*[39] or *Voldemort*[40] for key-value; *Amazon Redshift*[41] or *Apache Kudu*[42] for column oriented; *Neo4j*[43] or *OrientDB*[44] for graph; and *MongoDB*[45] or *RethinkDB*[46] for document. Finally, NewSQL are high-availability main memory databases which usually are deployed in specialized hardware, where we encounter *SAP Hana*[47], *NuoDB*[48] or *VoltDB*[49].

Real-time Views. In-memory databases are currently the most popular options, for instance *Redis*[50], *Elastic*[51], *Amazon ElastiCache*[52]. Alternatively,

[31]`http://spark.apache.org/streaming`
[32]`https://flink.apache.org`
[33]`https://aws.amazon.com/kinesis/analytics`
[34]`https://www.oracle.com/database`
[35]`http://www.postgres-xl.org`
[36]`https://www.mysql.com/products/cluster`
[37]`https://hbase.apache.org`
[38]`http://cassandra.apache.org`
[39]`https://aws.amazon.com/dynamodb`
[40]`http://www.project-voldemort.com/voldemort`
[41]`https://aws.amazon.com/redshift`
[42]`http://getkudu.io`
[43]`http://neo4j.com`
[44]`http://orientdb.com/orientdb`
[45]`https://www.mongodb.org`
[46]`https://www.rethinkdb.com`
[47]`https://hana.sap.com`
[48]`http://www.nuodb.com`
[49]`https://voltdb.com`
[50]`http://redis.io`
[51]`https://www.elastic.co`
[52]`https://aws.amazon.com/elasticache`

PipelineDB[53] offers mechanism to query a data stream via continuous query languages.

Query Engine. There is a vast variety of tools available for query engines. OLAP engines such as *Apache Kylin*[54] provide multidimensional analysis capabilities, on the other hand solutions like *Kibana*[55] or *Tableau*[56] enable the user to easily define complex charts over the data views.

6.2 Component selection

Selecting components to instantiate *Bolster* is a typical (C)OTS (commercial off-the-shelf) selection problem [95]. Considering a big part of the landscape of available Big Data tools is open source or well-documented, we follow a quality model approach for their selection, as done in [22]. To this end, we adopt the ISO/IEC 25000 SQuaRE standard (*Software Product Quality Requirements and Evaluation*) [79] as reference quality model. Such model is divided into characteristics and subcharacteristics, where the latter allows the definition of metrics (see ISO 25020). In the context of (C)OTS, the two former map to the hierarchical criteria set, while the latter to evaluation attributes. Nevertheless, the aim of this chapter is not to provide exhaustive guidelines on its usage whatsoever, but to supply a blueprint to be tailored to each organization. Figure 2.4 depicts the subset of characteristics considered relevant for such selection. Note that not all subcharacteristics are applicable, given that we are assessing the selection of off-the-shelf software for each component.

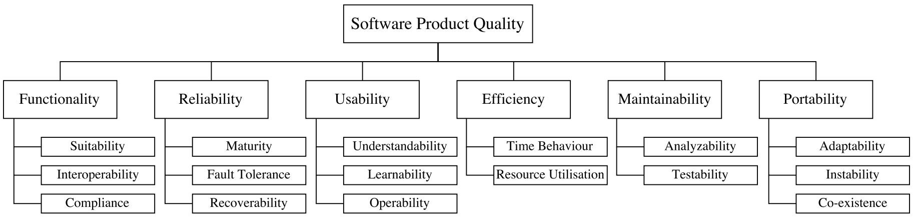


Fig. 2.4: Selected characteristics and subcharacteristics from SQuaRE

Evaluation attributes

Previously, we discussed that ISO 25020 proposes candidate metrics for each present subcharacteristic. However, we believe that they do not cover the singularities required for selecting open source Big Data tools. Thus, in the

[53]https://www.pipelinedb.com
[54]http://kylin.apache.org
[55]https://www.elastic.co/products/kibana
[56]http://www.tableau.com

following subsections we present a candidate set of evaluation attributes which were used in the use case applications described in Section 7. Each has associated a set of ordered values from worst to better and its semantics.

Functionality. After analysing the artifacts derived from the requirement elicitation process, a set of target functional areas should be devised. For instance, in an agile methodology, it is possible to derive such areas by clustering user stories. Some examples of functional areas related to Big Data are: *Data and Process Mining*, *Metadata Management*, *Reporting*, *BI 2.0* or *Real-time Analysis*. *Suitability* specifically looks at such functional areas, while with the other evaluation attributes we evaluate information exchange and security concerns.

Suitability
Number of functional areas targeted in the project which benefit from its adoption.
Interoperability
1, no input/output connectors with other considered tools
2, input/output connectors available with some other considered tools
3, input/output connectors available with many other considered tools
Compliance
1, might rise security or privacy issues
2, does not raise security or privacy issues

Reliability. It deals with trustworthiness and robustness factors. *Maturity* is directly linked to the stability of the software at hand. To that end, we evaluate it by means of the Semantic Versioning Specification[57]. The other two factors, *Fault Tolerance* and *Recoverability*, are key Big Data requirements to ensure the overall integrity of the system. We acknowledge it is impossible to develop a fault tolerant system, thus our goal here is to evaluate how the system reacts in the presence of faults.

[57]`http://semver.org`

Maturity
1, major version zero (0.y.z)
2, public release (1.0.0)
3, major version (x.y.z)
Fault Tolerance
1, the system will crash if there is a fault
2, the system can continue working if there is a fault but data might be lost
3, the system can continue working and guarantees no data loss
Recoverability
1, requires manual attention after a fault
2, automatic recovery after fault

Usability. In this subcharacteristic, we look at productive factors regarding the development and maintenance of the system. In *Understandability*, we evaluate the complexity of the system's building blocks (e.g., parallel data processing engines require knowledge of functional programming). On the other hand, *Learnability* measures the learning effort for the team to start developing the required functionalities. Finally, in *Operability*, we are concerned with the maintenance effort and technical complexity of the system.

Understandability
1, high complexity
2, medium complexity
3, low complexity
Learnability
1, the operating team has no knowledge of the tool
2, the operating team has small knowledge of the tool and the learning curve is known to be long
3, the operating team has small knowledge of the tool and the learning curve is known to be short
4, the operating team has high knowledge of the tool
Operability
1, operation control must be done using command-line
2, offers a GUI for operation control

Efficiency. Here we evaluate efficiency aspects. *Time Behaviour* measures the performance at processing capabilities, measured by the way the evaluated tool shares intermediate results, which has a direct impact on the response time. On the other hand, *Resource Utilisation* measures the hardware needs for the system at hand, as it might affect other coexisting software.

Time Behaviour
1, shares intermediate results over the network
2, shares intermediate results on disk
3, shares intermediate results in memory

Resource Utilisation
1, high amount of resources required (on both master and slaves)
2, high amount of resources required (either on master or slaves)
3, low amount of resources required

Maintainability. It concerns continuous control of software evolution. If a tool provides fully detailed and transparent documentation, it will allow developers to build robust and fault-tolerant software on top of them (*Analysability*). Furthermore, if such developments can be tested automatically (by means of unit tests) the overall quality of the system will be increased (*Testability*).

Analysability
1, online up to date documentation
2, online up to date documentation with examples
3, online up to date documentation with examples and books available

Testability
1, doesn't provide means for testing
2, provides means for unit testing
3, provides means for integration testing

Portability. Finally, here we evaluate the adjustment of the tool to different environments. In *Adaptability*, we analyse the programming languages offered by the tool. *Instability* and *Co-existence* evaluate the effort required to install such tool and coexistence constraints respectively.

Adaptability
1, available in one programming language
2, available in many programming languages
3, available in different programming languages and offering API access

Instability
1, requires manual build
2, self-installing package
3, shipped as part of a platform distribution

Co-existence
1, cannot coexist with other selected tools
2, can coexist with all selected tools

6.3 Tool evaluation

The purpose of the evaluation process is, for each of the candidate tools to instantiate *Bolster*, to derive a ranking of the most suitable one according to the evaluation attributes previously described. The proposed method is based on the weighted sum model (WSM), which allows weighting criteria (w_i) in order to prioritize the different subcharacteristics. Weights should be assigned according to the needs of the organization. Table 2.4 depicts an example selection for the *Batch Processing* component for the use case described in Section 7.1. For each studied tool, the *Atomic* and *Weighted* columns indicate its unweighted (f_i) and weighted score ($w_i f_i$), respectively using a range from one to five. For each characteristic, the weighted average of each component is shown in light grey (i.e., the average of each weighted subcharacteristic $\sum_i f_i / \sum_i w_i$). Finally, in black, the final score per tool is depicted. From the exemplar case of Table 2.4, we can conclude that, for the posed weights and evaluated scores, *Apache Spark* should be the selected tool, in from of *Apache MapReduce* and *Apache Flink* respectively.

			Evaluated Software					
			Apache Spark		Apache MapReduce		Apache Flink	
Characteristic	Subcharacteristic	Weight	Atomic	Weighted	Atomic	Weighted	Atomic	Weighted
Functionality	Suitability	2	3	6	2	4	3	6
	Interoperability	3	3	9	1	1	1	3
	Compliance	1	2	2	2	2	2	2
			2.83		1.50		1.83	
Reliability	Maturity	1	3	3	3	3	1	1
	Fault Tolerance	5	3	15	3	15	3	15
	Recoverability	2	2	4	2	4	2	4
			2.75		2.75		2.50	
Usability	Understandability	5	2	10	3	15	2	10
	Learnability	3	4	12	4	12	2	6
	Operability	2	2	4	1	2	2	4
			2.60		2.90		2.00	
Efficiency	Time Behaviour	3	3	9	1	3	3	9
	Resource Utilisation	4	1	4	2	8	1	4
			1.86		1.57		1.86	
Maintainability	Analysability	4	3	12	3	12	2	8
	Testability	2	2	4	1	2	1	2
			2.67		2.33		1.67	
Portability	Adaptability	3	2	6	1	3	2	6
	Instability	4	3	12	3	12	2	8
	Co-existence	1	2	2	2	2	2	2
			2.50		2.13		2.00	
			2.53		2.27		2.00	

Table 2.4: Example tool selection for *Batch Processing*

7 Industrial Experiences

In this section we depict three industrial projects, involving five organizations, where *Bolster* has been successfully adopted. For each project, we describe the use case context and the specific Bolster instantiation in graphical form. Finally we present the results of a preliminary validation that measure the perception of *Bolster* from the relevant industrial stakeholders.

7.1 Use cases and instantiation

BDAL: Big Data Analytics Lab

This project takes place in a multinational company in Barcelona[58]. It runs a data-driven business model and decision making relies on predictive models. Three main design issues were identified: (a) each department used its own processes to create data matrices, which were then processed to build predictive models. For reusability, data sets were preprocessed in ad hoc repositories (e.g., Excel sheets), generating a data governance problem; (b) data analysts systematically performed data management tasks, such as parsing continuous variable discretization or handling missing values, with a negative impact on their efficiency; (c) data matrices computation resulted in an extremely time consuming process due to their large volumes. Thus, their update rate was usually in the range of weeks to months.

The main goal was to develop a software solution to reduce the exposure of data analysts to data management and governance tasks, as well as boost performance in data processing.

***Bolster* instantiation.** *Bolster*'s Semantic Layer allowed the organization to overcome the data governance problem, consider additional data sources, and provide automation of data management processes. Additionally, there was a boost of performance in data processing thanks to the distributed computing and parallelism in the storage and processing of the Batch and Serving Layers. The nature of the data sources and analytical requirements did not justify the components in the Speed Layer, thus *Bolster*'s instantiation was narrowed to Batch, Semantic and Serving Layers. Figure 2.5 depicts the tools that compose *Bolster*'s instantiation instantiation for this use case.

H2020 SUPERSEDE Project

The SUPERSEDE[59] project proposes a feedback-driven approach for software life-cycle management. It considers user feedback and runtime data as an integral part of the design, development, and maintenance of software services and applications. The ultimate goal is to improve the quality perceived by software end-users as well as support developers and engineers to make the right software adaptation and evolution decisions. Three use cases proposed

[58]No details about the company can be revealed due to non-disclosure agreements.

[59]`https://www.supersede.eu`

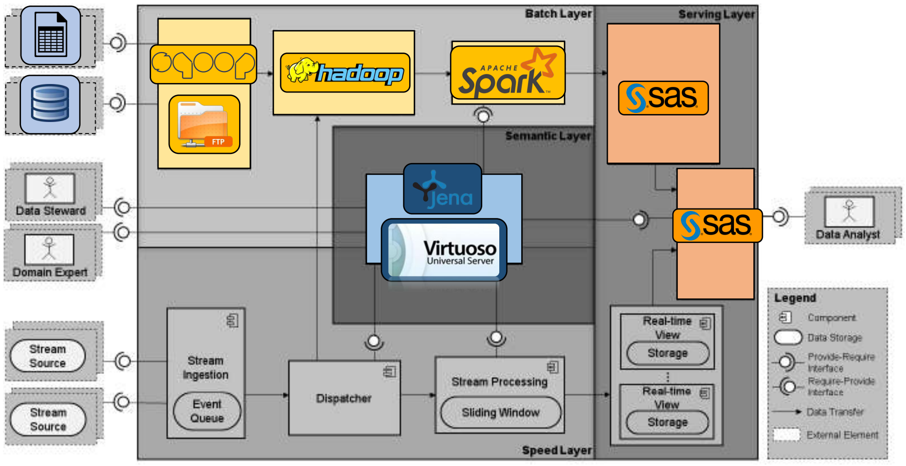


Fig. 2.5: *Bolster* instantiation for the BDAL use case

by industrial partners, namely: *Siemens AG Oesterreich* (Austria), *Atos* (Spain) and *SEnerCon GmbH* (Germany), are representative of different data-intensive application domains in the areas of energy consumption management in home automation and entertainment event webcasting.

SUPERSEDE's Big Data architecture is the heart of the analysis stage that takes place in the context of a monitor-analyse-plan-execute (MAPE) process [90]. Precisely, some of its responsibilities are (i) collecting and analysing user feedback from a variety of sources, (ii) supporting decision making for software evolution and adaptation based on the collected data, and (iii) enacting the decision and assessing its impact. This set of requirements yielded the following challenges: (a) ingest multiple fast arriving data streams from monitored data and process them in real-time, for instance with sliding window operations; (b) store and integrate user feedback information from multiple and different sources; (c) use all aforementioned data in order to analyse multi-modal user feedback, identify profiles, usage patterns and identify relevant indicators for usefulness of software services. All implemented in a performance oriented manner in order to minimize overhead.

***Bolster* instantiation.** *Bolster* allowed the definition of a data governance protocol encompassing the three use cases in a single instantiation of the architecture, while preserving data isolation. The Speed Layer enabled the ingestion of continuous data streams from a variety of sources, which were also dispatched to the Data Lake. The different analytical components in the Serving Layer allowed data analysts to perform an integrated analysis. Figure

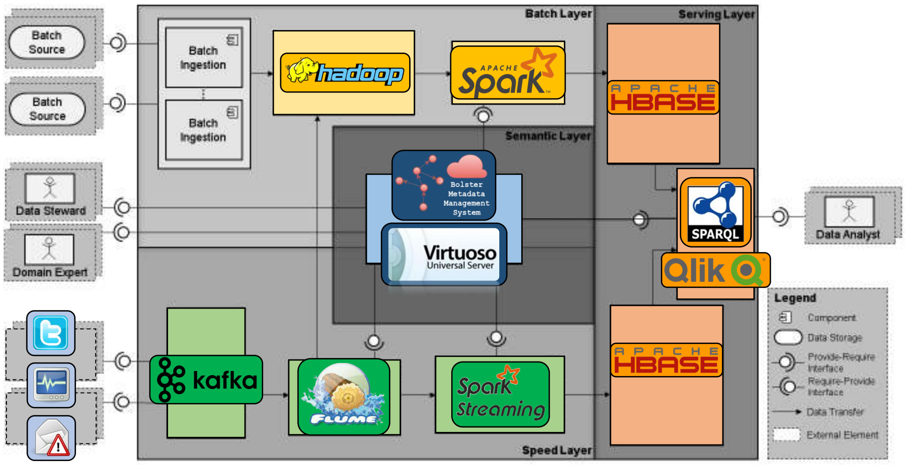


Fig. 2.6: *Bolster* instantiation for the SUPERSEDE use case

2.6 depicts the tools that compose *Bolster*'s instantiation for this use case.

WISCENTD: The WHO Information System to Control and Eliminate NTDs

The WISCENTD[60] project funded by the World Health Organization (WHO) is part of the *Programme on Control of neglected tropical diseases (NTDs)*. This project has the goal of strengthen health information systems in endemic countries in order to empower them in taking evidence-based decisions to tailor control interventions; and to capture, clean, store, consolidate and analyse all available information in order to permit WHO to efficiently monitor advances in control and finally verify the elimination of selected NTDs. To this end, the aim is to build an information system serving as an integrated repository of all information, from different countries and organizations, related to NTDs. The ultimate beneficiaries of this information system will be the affected neglected populations whose health will improve if the appropriate interventions are implemented based on use of good-quality data.

The role of the Big Data architecture is to ingest and integrate data from a variety of data sources and formats. Currently, the big chunk of data is ingested from DHIS2[61], an information system where national ministries enter data related to inspections, diagnoses, etc. Additionally, NGOs make available similar information according to their actions. The information dealt with is continuously changing by nature at all levels: data, schema and sources.

[60] `https://www.who.int/neglected_diseases/disease_management/wiscentds/en`
[61] `https://www.dhis2.org`

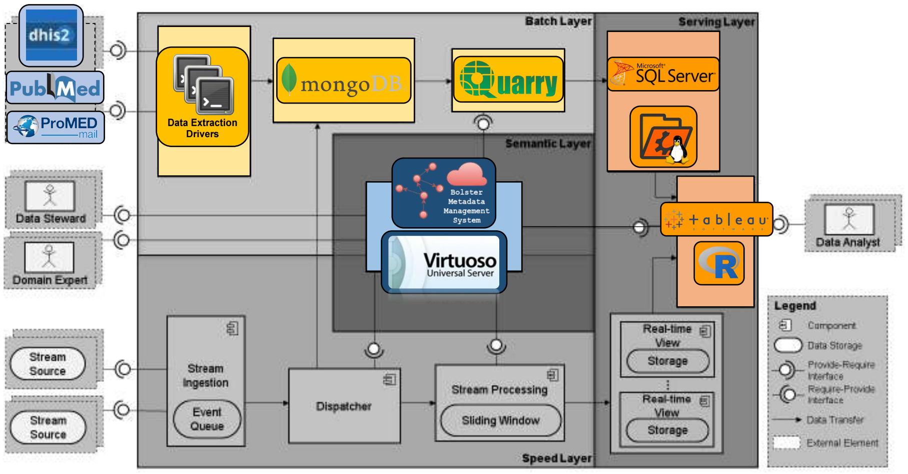


Fig. 2.7: *Bolster* instantiation for the WISCENTD use case

Thus, the challenge falls in the flexibility of the system to accommodate such information and the one to come. Additionally, flexible mechanisms to query such data should be defined, as future information requirements will be totally different from today's.

***Bolster* instantiation.** Instantiating *Bolster* favored a centralized management, in the Semantic Layer, of the different data sources along with the provided schemata, a feature that facilitated the data integration and Data Lake management tasks. Similarly to the BDAL use case, the ingestion and analysis of data was performed with batch processes, hence dismissing the need to instantiate the Speed Layer. Figure 2.7 depicts the tools that compose *Bolster*'s instantiation for this use case.

Summary

In this subsection, we discuss and summarize the previously presented instantiations. We have shown how, as an SRA, *Bolster* can flexibly accomodate different use cases with different requirements by selectively instantiating its components. Due to space reasons, we cannot show the tool selection tables per component, instead we present the main driving forces for such selection using the dimensions devised in Section 2. Table 2.5 depicts the key dimensions that steered the instantiation of *Bolster* in each use case.

Most of the components have been successfully instantiated with off-the-shelf tools. However, in some cases it was necessary to develop customized solutions to satisfy specific project requirements. This was especially the case

Use Case	*Volume*	*Velocity*	*Variety*	*Variability*	*Veracity*
BDAL	✓		✓	✓	✓
SUPERSEDE		✓	✓	✓	✓
WISCENTD			✓	✓	✓

Table 2.5: Characterization of use cases and Big Data dimensions

for the MDM, for which off-the-shelf tools were unsuitable in two out of three projects. It is also interesting to see that, due to the lack of connectors between components, it has been necessary to use glue code techniques (e.g., in WISCENTD dump files to a UNIX file system and batch loading in R). As final remark, note that the deployment of *Bolster* in all described use cases occurred in the context of research projects, which usually entail a low risk. However, in data-driven organizations such information processing architecture is the business's backbone, and adopting *Bolster* can generate risk as few components from the legacy architecture will likely be reused. This is due to the novelty in the landscape of Big Data management and analysis tools, which lead to a paradigm shift on how data are stored and processed.

7.2 Validation

The overall objective of the validation is to "assess to which extent *Bolster* leads to a perceived quality improvement in the software or service targeted in each use case". Hence, the validation of the SRA involves a quality evaluation where we investigated how Big Data practitioners perceive *Bolster*'s quality improvements. To this end, as before, we rely on SQuaRE's quality model, however now focusing on the quality-in-use model. The model is hierarchically composed of a set of characteristics and sub-characteristics. Each (sub-)characteristic is quantified by a Quality Measure (QM), which is the output of a measurement function applied to a number of Quality Measure Elements (QME).

Selection of participants

For each of the five aforementioned organizations, in the three use cases, a set of practitioners was selected as participants to report their perception about the quality improvements achieved with *Bolster* using the data collection method detailed in Section 7.2. Care was taken in selecting participants with different backgrounds (e.g., a broad range of skills, different seniority levels) and representative of the actual target population of the SRA. This is summarized in Table 2.6, which depicts the characteristics of the respondents in each organization. Recall that the SUPERSEDE project involves three industrial partners, hence we refer by SUP-1, SUP-2 and SUP-3 to, respectively, *Siemens*, *Atos* and *SEnerCon*.

ID	Org.	Function	Seniority	Specialties
#1	BDAL	Data analyst	Senior	Statistics
#2	BDAL	SW architect	Junior	Non-relational databases, Java
#3	SUP-1	Research scientist	Senior	Statistics, machine learning
#4	SUP-1	Key expert	Senior	Software engineering
#5	SUP-1	SW developer	Junior	Java, security
#6	SUP-1	Research scientist	Senior	Stream processing, semantic web
#7	SUP-2	Dev. team head	Senior	CDN, relational databases
#8	SUP-2	Project manager	Senior	Software engineering
#9	SUP-3	SW developer	Junior	Web technologies, statistics
#10	SUP-3	SW developer	Junior	Java, databases
#11	SUP-3	SW architect	Senior	Web technologies, project leader
#12	WISCENTD	SW architect	Senior	Statistics, software engineering
#13	WISCENTD	Research scientist	Senior	Non-relational databases, semantic web
#14	WISCENTD	SW developer	Junior	Java, web technologies

Table 2.6: List of participants per organization

Definition of the data collection methods

The quality characteristics were evaluated by means of questionnaires. In other words, for each characteristic (e.g., trust), the measurement method was the question whether a participant disagrees or agrees with a descriptive statement. The choice of the participant (i.e., the extent of agreement in a specific rating scale) was the QME. For each characteristic, a variable numbers of QMEs were collected (i.e., one per participant). The final QM was represented by the mean opinion score (MOS), computed by the measurement function $\sum_i^N QME_i/N$, where N is the total number of participants. We used a 7-values rating scale, ranging from 1 strongly disagree to 7 strongly agree. Table 2.7 depicts the set of questions in the questionnaire along with the quality subcharacteristic they map to.

Execution of the validation

The heterogeneity of organizations and respondents called for a strict planning and coordination for the validation activities. A thorough time-plan was elaborated, so as to keep the progress of the evaluation among use cases. The actual collection of data spanned over a total duration of three weeks. Within these weeks, each use case evaluated the SRA in a 3-phase manner:

1. *(1 week)*: A description of Bolster in form of an excerpt of Section 4 of this chapter was provided to the respondents, as well as access to the proposed solution tailored to each organization.

Subcharacteristic	Question
Usefulness	• The presented Big Data architecture would be useful in my UC
Satisfaction	• Overall I feel satisfied with the presented architecture
Trust	• I would trust the Big Data architecture to handle my UC data
Perceived Relative Benefit	• Using the proposed Big Data architecture would be an improvement with respect to my current way of handling and analysing UC data
Functional Completeness	• In general, the proposed Big Data architecture covers the needs of the UC (subdivided into user stories)
Functional Appropriateness	• The proposed Big Data architecture facilitates the storing and management of the UC data • The proposed Big Data architecture facilitates the analysis of historical UC data • The proposed Big Data architecture facilitates the real-time analysis of UC data stream • The proposed Big Data architecture facilitates the exploitation of the semantic annotation of UC data • The proposed Big Data architecture facilitates the visualization of UC data statistics
Functional Correctness	• The extracted metrics obtained from the Big Data architecture (test metrics) match the results rationally expected
Willingness to Adopt	• I would like to adopt the Big Data architecture in my UC

Table 2.7: Validation questions along with the subcharacteristics they map to

2. *(1 hour)*: For each organization, a workshop involving a presentation on the SRA and a Q&A session was carried out.

3. *(1 day)*: The questionnaire was provided to each respondent to be answered within a day after the workshop.

Once the collection of data was completed, we digitized the preferences expressed by the participants in each questionnaire. We created summary spreadsheets merging the results for its analysis.

Analysis of validation results

Figure 2.8 depicts, by means of boxplots, the aggregated MOS for all respondents (we acknowledge the impossibility to average ordinal scales, however

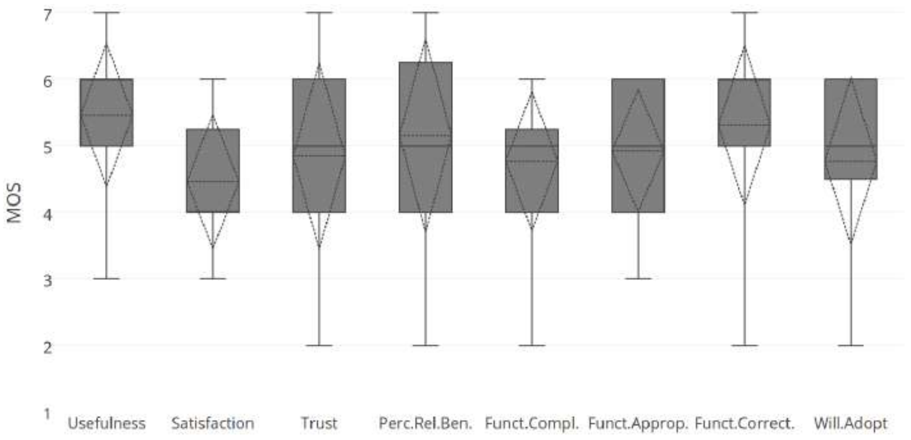


Fig. 2.8: Validation per Quality Factor

we consider them as their results fall within the same range). The top and bottom boxes respectively denote the first and third quartile, the solid line the median and the whiskers maximum and minimum values. The dashed line denotes the average, and the diamond shape the standard deviation. Note that *Functional Appropriateness* is aggregated into the average of the 5 questions that compose it, and functional completeness is aggregated into the average of multiple user-stories (a variable number depending on the use case).

We can see that, when taking the aggregated number, none of the characteristics scored below the mean of the rating scale (1-7) indicating that *Bolster* was on average well-perceived by the use cases. Satisfaction sub-characteristics (i.e., *Satisfaction*, *Trust*, and *Usefulness*) present no anomaly, with *Usefulness* standing out as the highest rated one. Regarding *Functional Appropriateness*, *Bolster* was perceived to be overall effective, with some hesitation regarding the functionality offered for the semantic exploitation of the data. All other scores are considerably satisfactory. The SRA is marked as functionally complete, correct, and expected to bring benefits in comparison to current techniques used in the use cases. Ultimately this leads to a large intention to use.

Discussion. We can conclude that generally user's perception is positive, being most answers in the range from *Neutral* to *Strongly Agree*. The preliminary assessment shows that the potential of the Bolster SRA is recognized also in the industry domain and its application is perceived to be beneficial in improving the quality-in-use of software products. It is worth noting, however, that some respondents showed reluctancy regarding the Semantic Layer in *Bolster*. We believe this aligns with the fact that Semantic Web technologies have not yet been widely adopted in industry. Thus, lack of known successful industrial use cases may raise caution among potential adopters.

8 Conclusions

Despite their current popularity, Big Data systems engineering is still in its inception. As any other disruptive software-related technology, the consolidation of emerging results is not easy and requires the effective application of solid software engineering concepts. In this chapter, we have focused on an architecture-centric perspective and have defined an SRA, *Bolster*, to harmonize the different components that lie in the core of such kind of systems. The approach uses the semantic-aware strategy as main principle to define the different components and their relationships. The benefits of *Bolster* are twofold. On the one hand, as any SRA, it facilitates the technological work of Big Data adopters by providing a unified framework which can be tailored to a specific context instead of a set of independent components that are glued together in an ad hoc manner. On the other hand, as a semantic-aware solution, it supports non-expert Big Data adopters in the definition and exploitation of the data stored in the system by facilitating the decoupling of the data steward and analyst profiles. However, we anticipate that in the long run, with the maturity of such technologies, the role of software architect will be replaced in favor of the database administrator. In this initial deployment, *Bolster* includes components for data management and analysis as a first step towards the systematic development of the core elements of Big Data systems. Thus, *Bolster* currently maps to the role played by a relational DBMS in traditional BI systems.

Acknowledgements. We thank Gerhard Engelbrecht for his assistance in setting up the validation process, and Silverio Martínez for his comments and insights that helped to improve this chapter. This work was partly supported by the H2020 SUPERSEDE project, funded by the EU Information and Communication Technologies Programme under grant agreement no 644018, and the GENESIS project, funded by the Spanish Ministerio de Ciencia e Innovación under project TIN2016-79269-R.

Chapter 3

An Integration-Oriented Ontology to Govern Evolution in Data-Intensive Ecosystems

This chapter is composed of the following papers:

- Proceedings of the Nineteenth International Workshop On Design, Optimization, Languages and Analytical Processing of Big Data (2017). CEUR Workshop Proceedings Volume 1810.
  DOI: `http://ceur-ws.org/Vol-1810/DOLAP_paper_09.pdf`
- Information Systems, 79: 3-19 (2019).
  DOI: `https://doi.org/10.1016/j.is.2018.01.006`

The layout of the papers has been revised.

Abstract

Big Data architectures allow to flexibly store and process heterogeneous data, from multiple sources, in their original format. The structure of those data, commonly supplied by means of REST APIs, is continuously evolving. Thus data analysts need to adapt their analytical processes after each API release. This gets more challenging when performing an integrated or historical analysis. To cope with such complexity, in this chapter, we present the Big Data Integration ontology, the core construct to govern the data integration process under schema evolution by systematically annotating it with information regarding the schema of the sources. To cope with syntactic evolution in the sources, we present an algorithm that semi-automatically adapts the ontology upon new releases. A functional and performance evaluation on real-world APIs is performed to validate our approach.

1 Introduction

Big Data ecosystems enable organizations to evolve their decision making processes from classic stationary data analysis [2] (e.g., transactional) to situational data analysis [104] (e.g., social networks). Situational data are commonly obtained in the form of data streams supplied by third party data providers (e.g., Twitter or Facebook), by means of web services (or APIs). Those APIs offer a part of their data ecosystem at a certain price allowing external data analysts to enrich their data pipelines with them. With the rise of the RESTful architectural style for web services [128], providers have flexible mechanisms to share such data, usually semi-structured (i.e., JSON), over web protocols (e.g., HTTP). However, such flexibility can be often a disadvantage for analysts. In contrast to other protocols offering machine-readable contracts for the structure of the provided data (e.g., SOAP), web services using REST typically do not publish such information. Hence, *analysts need to go over the tedious task of carefully studying the documentation and adapting their processes to the particular schema provided*. Besides the aforementioned complexity imposed by REST APIs, there is a second challenge for data analysts. *Data providers are constantly evolving such endpoints*[1][2], hence *analysts need to continuously adapt the dependent processes to such changes*. Previous work on schema evolution has focused on software obtaining data from relational views [107, 144]. Such approaches rely on the capacity to veto changes affecting consumer applications. Those techniques are not valid in our setting, due to the lack of explicit schema information and the impossibility to prevent changes from third party data providers.

Given this setting, the problem is how to aid the data analyst in the presence of schema changes by (a) understanding what parts of the data structure change and (b) adapting her code to this change.

Providing an integrated view over an evolving and heterogeneous set of data sources is a challenging problem, commonly referred as the data variety challenge [75], that traditional data integration techniques fail to address. An approach to tackle it is to leverage on Semantic Web technologies, and the so-called ontology-based data access (OBDA). OBDA are a class of systems that enable end-users to query an integrated set of heterogeneous and disparate data sources decreasing the need for IT support [130]. OBDA achieves its purpose by providing a conceptualization of the domain of interest, via an ontology, allowing users to pose ontology-mediated queries (OMQs), and thus creating a separation of concerns between the conceptual and the database level. Due to the simplicity and flexibility of ontologies, they constitute an ideal tool to model such heterogeneous environments. However, such flexibility is

[1]`https://dev.twitter.com/ads/overview/recent-changes`
[2]`https://developers.facebook.com/docs/apps/changelog`

also one of its biggest drawbacks, as OBDA currently has no means to provide continuous adaptation to changes in the sources (e.g., schema evolution), and thus causing queries to crash.

The problem is not straightforwardly addressable, as current OBDA approaches, which are built upon generic reasoning in description logics (DLs), represent schema mappings following the *global-as-view* (GAV) approach [98]. In GAV, elements of the ontology are characterized in terms of a query over the source schemata. This provides simplicity in the query answering methods, which consists of unfolding the queries to the sources. Changes in the source schemata, however, will invalidate the mappings. In contrast, *local-as-view* (LAV) schema mappings characterize elements of the source schemata in terms of a query over the ontology. They are naturally suited to accomodate dynamic environments, as we will see. The trade-off however, comes at the expense of query answering, which becomes a computationally complex task that might require reasoning [82]. To this end, we aim to bridge this gap by providing a new approach to data integration using ontologies with LAV mapping assertions, while maintaining query answering tractable.

In this chapter, we focus on the definition of the ontological vocabulary, which relies on a tailored metadata model to design the ontology (i.e., a set of design guidelines). This allows to annotate the data integration constructs with semantic annotations, enabling to automate the process of evolution. Our approach builds upon the well-known framework for data integration [98], and it is divided in two levels represented by graphs (i.e., Global and Source graphs) in order to provide analysts with an integrated and format-agnostic view of the sources. By relying on wrappers (from the well-known mediator/wrapper architecture for data integration [53]) we are able to accomodate different kinds of data sources, as the query complexity is delegated to wrappers and the ontology is only concerned with how to join them and what attributes are projected. Additionally, we allow the ontology to contain elements that do not exist in the sources (i.e., syntactic sugar for data analysts), such as taxonomies, to facilitate querying. Note that the definition of the metadata model is highly tailored to the query rewriting process using it, which is the focus of next chapter (see Chapter 4).

Contributions The main contributions of this chapter are as follows:

- We introduce a structured ontology based on an RDF vocabulary that allows to model and integrate evolving data from multiple providers. As an add-on, we take advantage of RDF's nature to semantically annotate the data integration process.

- We provide a method that handles schema evolution on the sources. According to our industry applicability study, we flexibly accommodate source changes by only applying changes to the ontology, dismissing the need to change the analyst's queries.

- We assess our method by performing a functional and performance evaluation. The former reveals that our approach is capable of semi-automatically accomodating all structural changes concerning data ingestion, which on average makes up 71.62% of the changes occurring on widely used APIs.

Outline. The rest of the chapter is structured as follows. Section 2 describes a running example and formalizes the problem at hand. Section 3 discusses the constructs of the Big Data Integration ontology and its RDF representation. Section 4 introduces the techniques to manage schema evolution. Section 5 reports on the evaluation results. Sections 6 and 7 discuss related work and conclude the chapter, respectively.

2 Overview

Our approach (see Figure 3.1) relies on a two-level ontology of RDF named graphs to accommodate schema evolution in the data sources. Such graphs are built based on a RDF vocabulary tailored for data integration. Precisely, we divide it into the *Global graph* ($\mathcal{G}$), and the *Source graph* ($\mathcal{S}$). Briefly, $\mathcal{G}$ represents an integrated view of the domain of interest (also known as domain ontology), while $\mathcal{S}$ represents data sources, wrappers and their schemata. On the one hand, data analysts issue OMQs to $\mathcal{G}$. We also assume a triplestore with a SPARQL endpoint supporting the RDFS entailment regime (e.g., subclass relations are automatically inferred) [151]. On the other hand, we have a set of data sources, each with a set of wrappers querying it. Different wrappers for a data source represent different schema versions. Under the assumption that wrappers provide a flat structure in first normal form, we can easily depict an accurate representation of their schema into $\mathcal{S}$. To accommodate a LAV approach, each wrapper in $\mathcal{S}$ is related to the fragment of $\mathcal{G}$ for which it provides data.

The management of such a complex structure (i.e., modifying it upon schema evolution in the sources) is a hard task to automate. To this end, we introduce the role of data steward as an analogy to the database administrator in traditional relational settings. Aided by semi-automatic techniques, s/he is responsible for (a) registering the wrappers of newly incoming, or evolved, data sources in $\mathcal{S}$, and (b) make such data available to analysts by defining LAV mappings to $\mathcal{G}$ (i.e., enriching the ontology with the mapping representations). With such setting, intuitively the problem consists of given a query over $\mathcal{G}$, to derive an equivalent query over the wrappers leveraging on $\mathcal{S}$. Throughout the rest of this section, we introduce the running example and the formalism behind our approach. To make a clear distinction among concepts, hereinafter, we will use *italics* to refer to elements in $\mathcal{G}$, while sans serif font to refer to elements in $\mathcal{S}$. Additionally, to refer to RDF constructs, we will use `typewriter` font.

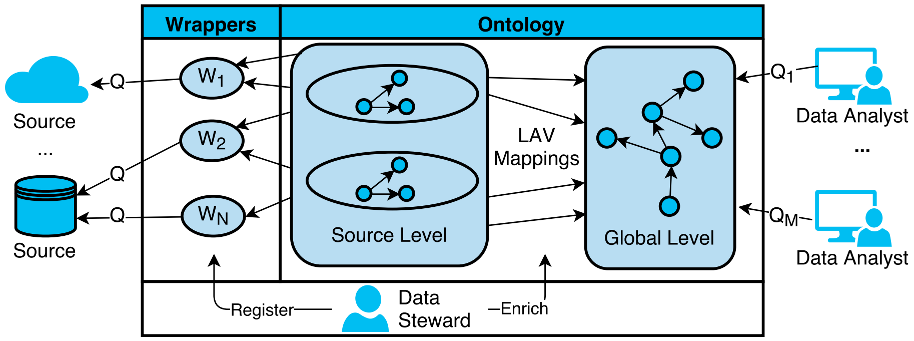


Fig. 3.1: High-level overview of our approach

2.1 Running example

As an exemplary use case we take the H2020 SUPERSEDE project[3]. It aims to support decision-making in the evolution and adaptation of software services and applications (i.e., *SoftwareApps*) by exploiting end-user feedback and monitored runtime data, with the overall goal of improving end-users' quality of experience. For the sake of this case study, we narrow the scope to video on demand (VoD) monitored data (i.e., *Monitor* tools generating *InfoMonitor* events) and textual feedback from social networks such as Twitter (i.e., *FeedbackGathering* tools generating *UserFeedback* events). This scenario is conceptualized in the UML depicted in Figure 3.2, which we use as a starting point to provide a high-level representation of the domain of interest that is later used to generate the ontological knowledge captured in $\mathcal{G}$. Figure 3.3 in Section 3 depicts the RDF-based representation of the UML diagram used in our approach, which we will introduce in detail in that section.

Next, let us assume three data sources, in the form of REST APIs, and respectively one wrapper querying each. The first data source provides information related to the VoD monitor, which consist of JSON documents as depicted in Code 3.1. We additionally define a wrapper on top of it obtaining the monitorId of the monitor and computing the lag ratio metric (a quality of service measure computed as the fraction of wait and watch time) indicating the percentage of time a user is waiting for a video. The query of this wrapper

[3]`https://www.supersede.eu`

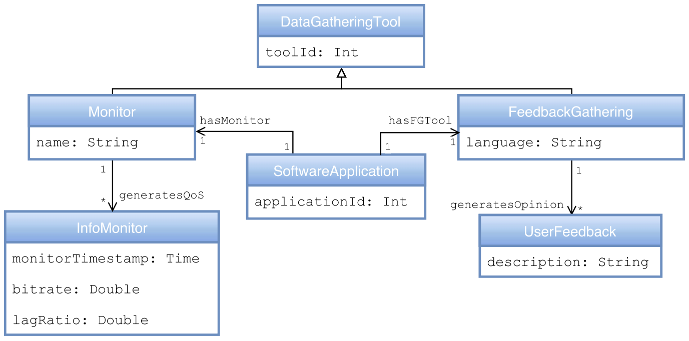


Fig. 3.2: UML conceptual model for the SUPERSEDE case study

is depicted in Code 3.2 using MongoDB syntax[4], where for each tuple the attribute VoDmonitorId (renamed from monitorId in the JSON) and lagRatio are projected (respectively mapping to the conceptual attributes *toolId* and *lagRatio*).

```
{
  "monitorId": 12,
  "timestamp": 1475010424,
  "bitrate": 6,
  "waitTime": 3,
  "watchTime": 4
}
```

Code 3.1: Sample JSON for VoD monitors

```
db.getCollection("vod").aggregate([
  {$project: {
    "VoDmonitorId":"$monitorId",
    "lagRatio": {$divide : ["$waitTime","
         $watchTime"]}}
  }
])
```

Code 3.2: Wrapper projecting attributes VoDmonitorId and lagRatio (using MongoDB's Aggregation Framework syntax)

For the sake of simplicity, hereinafter, we will represent wrappers as relations where their schema are the attributes projected by the queries, dismissing the details of the underlying query. Hence, the previous wrapper would be depicted as $w_1(\text{VoDmonitorId}, \text{lagRatio})$ (note that the JSON key monitorId has been renamed to VoDmonitorId). To complete our running example, we define a wrapper $w_2(\text{FGId}, \text{tweet})$ providing, respectively, the *toolId* for the *FeedbackGathering* at hand and the *description* for such *UserFeedback*. Finally, the wrapper $w_3(\text{TargetApp}, \text{MonitorId}, \text{FeedbackId})$ states for each *SoftwareApplication* the *toolId* of its associated *Monitor* and *FeedbackGathering* tools. Table 3.1 depicts an example of the output generated by each wrapper.

Now, the goal is to enable data analysts to query the attributes of the ontology-based representation of the UML diagram (i.e., $\mathcal{G}$) by navigating over the classes, such that the sources are automatically accessed. Assume the

[4]Note that the use of the `aggregate` keyword is used to invoke the aggregate querying framework. The `aggregate` keyword does not entail grouping unless the `$group` keyword is used. Thus, note no aggregation is performed in this query.

w_1	
VoDmonitorId	lagRatio
12	0.75
12	0.90
18	0.1

w_2	
FGId	tweet
77	"I continuously see the loading symbol"
45	"Your video player is great!"

w_3		
TargetApp	MonitorId	FeedbackId
1	12	77
2	18	45

Table 3.1: Sample output for each of the exemplary wrappers.

analyst wants to retrieve for each *applicationId* its *lagRatio* instances. Hence, the task consists of rewriting such OMQ to an equivalent one over the wrappers, which can be translated to the following relational algebra expression: $\Pi_{w_3.\text{TargetApp},w_1.\text{lagRatio}}(w_1 \underset{\text{VoDmonitorId=MonitorId}}{\bowtie} w_3)$. Recall such rewriting process is depicted in Chapter 4.

Assume now that the first data source releases a new version of its API and in the new schema lagRatio has been renamed to bufferingRatio. Hence, a new wrapper $w_4(\text{VoDmonitorId}, \text{bufferingRatio})$ is defined. With such setting, the analyst should not be aware of such schema evolution, but now the query should consider both versions and be automatically rewritten to the following expression: $\Pi_{w_3.\text{TargetApp},w_1.\text{lagRatio}}(w_1 \underset{\text{VoDmonitorId=MonitorId}}{\bowtie} w_3) \bigcup \Pi_{w_3.\text{TargetApp},w_4.\text{bufferingRatio}}(w_4 \underset{\text{VoDmonitorId=MonitorId}}{\bowtie} w_3)$.

2.2 Notation

We consider a set of data sources $D = \{D_1, \ldots, D_n\}$, where each D_i consists of a set of wrappers $\{w_1, \ldots, w_m\}$ representing views over different schema versions. We define the operator $source(w)$, which returns the data source D to which w belongs to. As previously stated, a wrapper is represented as a relation with the attributes its query projects. We distinguish between ID and non-ID attributes, hence a wrapper is defined as $w(\overline{a_{ID}}, \overline{a_{nID}})$, where $\overline{a_{ID}}$ and $\overline{a_{nID}}$ are respectively the set of its ID attributes and non-ID attributes.

Example 2.1
The VoD monitoring API would be depicted as $D_1 = \{w_1(\{\text{VoDmonitorId}\}, \{\text{lagRatio}\}), w_4(\{\text{VoDmonitorId}\}, \{\text{bufferingRatio}\})\}$, the feedback gathering API as $D_2 = \{w_2(\{\text{FGId}\}, \{\text{tweet}\})$ and the relationship API as $D_3 = \{w_3(\{\text{TargetApp}, \text{MonitorId}, \text{FeedbackId}\}, \{\})$.

Wrappers can be joined to each other by means of a restricted equi join on IDs ($\widetilde{\bowtie}$). The semantics of $\widetilde{\bowtie}$ are those of an equi join ($w_i \underset{a=b}{\bowtie} w_j$), but only valid if $a \in w_i.\overline{a_{ID}}$ and $b \in w_j.\overline{a_{ID}}$. We also define the projection operator $\widetilde{\Pi}$, whose semantics are likewise a standard projection for non-ID attributes. We do not permit to project out any ID attribute, as they are necessary for $\widetilde{\bowtie}$. With such constructs, we can now define the concept of a walk over the wrappers (W), which consists of a relational algebra expression where wrappers are joined ($\widetilde{\bowtie}$) and their attributes are projected ($\widetilde{\Pi}$). Thus, we formally define a walk as $W = \widetilde{\Pi}(w_1)\widetilde{\bowtie} \dots \widetilde{\bowtie}\widetilde{\Pi}(w_k)$. Furthermore, we work under the assumption that schema versions from the same data source should not be joined (e.g., w_1 and w_4 in the running example). To formalize this assumption let $wrappers(W)$ denote the set of wrappers used in walk W. Then we require that $\forall w_i, w_j \in wrappers(W) : source(w_i) \neq source(w_j)$. Note that a walk can also be seen as a conjunctive query over the wrappers (i.e., select-project-join expression), thus two walks are equivalent if they join the same wrappers dismissing the order how this is done. Consider, however, that as the operator $\widetilde{\Pi}$ does not project out ID attributes, all ID attributes will be part of the output schema.

Example 2.2
The exemplary query (i.e., for each *applicationId* fetch its *lagRatio* instances) would consist of two walks $W_1 = \widetilde{\Pi}_{\mathsf{lagRatio}}(w_1) \underset{\mathsf{VoDmonitorId=MonitorId}}{\widetilde{\bowtie}} \widetilde{\Pi}_{\mathsf{TargetApp}}(w_3)$ and $W_2 = \widetilde{\Pi}_{\mathsf{bufferingRatio}}(w_4) \underset{\mathsf{VoDmonitorId=MonitorId}}{\widetilde{\bowtie}} \widetilde{\Pi}_{\mathsf{TargetApp}}(w_3)$.

Next, we formalize the ontology $\mathcal{T}$ as a 3-tuple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ of RDF named graphs. The Global graph ($\mathcal{G}$) contains the concepts and relationships that analysts will use to query, the source graph ($\mathcal{S}$) the data sources and the schemata of wrappers, and the mappings graph ($\mathcal{M}$) the LAV mappings between $\mathcal{S}$ and $\mathcal{G}$. Recall that data analysts pose OMQs over $\mathcal{G}$, however we do not allow arbitrary queries. We restrict OMQs to a subset of standard SPARQL defining subgraph patterns of $\mathcal{G}$, and only project elements of such pattern. Code 3.3 depicts the template of the permitted queries. Precisely, $attr_1, \dots, attr_n$ must be attribute URIs (i.e., mapping to the UML attributes in Fig. 3.2), where each $attr_i$ has an invited variable $?v_i$ in the SELECT clause. The set of triples in the WHERE clause must define a connected subgraph of $\mathcal{G}$. On the one hand, it contains triples of the form $\langle s_i, hasFeature, attr_i \rangle$, where s_i are class URIs (i.e., mapping to UML classes) and *hasFeature* a predicate stating that $attr_i$ is attribute of class s_i. On the other hand, it contains triples of the form $\langle s_j, p_j, o_j \rangle$, where s_j and o_j are class URIs (i.e., mapping to UML classes) and p_i predicate URIs (i.e., mapping to relationships between UML classes).

```
SELECT ?v_1 ... ?v_n
FROM G
WHERE {
 VALUES (?v_1 ... ?v_n) { (attr_1 ... attr_n) }
 s_1 p_1 attr_1 .
 ...
 s_n p_n attr_n .
 ...
 s_m p_m o_m
}
```

Code 3.3: Template for accepted SPARQL queries

OMQs are meant to be translated to sets of walks (a process we depict in Chapter 4), to this end the aforementioned SPARQL queries must be parsed and manipulated. This task can be simplified leveraging on SPARQL Algebra[5], where the semantics of the query evaluation are specified. Libraries such as ARQ[6] provide mechanisms to get such algebraic structure for a given SPARQL query. Code 3.4 depicts the algebra structure generated after parsing the subset of permitted SPARQL queries.

```
(project (?v_1 ... ?v_n)
  (join
    (table (vars ?v_1 ... ?v_n)
      (row [?v_1 attr_1] ... [?v_n attr_n])
    )
    (bgp
      (triple s_1 p_1 attr_1)
      ...
      (triple s_n p_n attr_n)
      ...
      (triple s_m p_m o_m)
)))))
```

Code 3.4: SPARQL algebra for the accepted SPARQL queries

In order to easily manipulate such algebraic structures, we formalize the allowed SPARQL queries as $Q_{\mathcal{G}} = \langle \pi, \varphi \rangle$, where π is the set of projected attributes (i.e., the URIs $attr_1, \ldots, attr_n$) and φ the graph pattern specified under the bgp clause (i.e., basic graph pattern). Note that $\pi \subseteq V(\varphi)$, where $V(\varphi)$ returns the vertex set of φ.

[5]https://www.w3.org/2001/sw/DataAccess/rq23/rq24-algebra.html

[6]https://www.w3.org/2011/09/SparqlAlgebra/ARQalgebra

Example 2.3
The exemplary query is depicted using SPARQL in Code 3.5. Alternatively, it would be represented as $\pi = \{lagRatio, applicationId\}$, and φ the subgraph $applicationId \xleftarrow[hasFeature]{} SoftwareApplication \xrightarrow[hasMonitor]{} Monitor \xrightarrow[generatesQoS]{} InfoMonitor \xrightarrow[hasFeature]{} lagRatio$.

```
SELECT ?x ?y
FROM 𝒢
WHERE {
 VALUES (?x ?y) { (applicationId lagRatio) }
 SoftwareApplication hasFeature applicationId .
 SoftwareApplication hasMonitor Monitor .
 Monitor generatesQoS InfoMonitor .
 InfoMonitor hasFeature lagRatio
}
```

Code 3.5: Running example's SPARQL query

The wrappers and the ontology are linked by means of schema mappings. Those are commonly formalized using tuple-generating dependencies (tgds) [46], which are logical expressions of the form $\forall x(\exists y \Phi(x,y) \mapsto \exists z \Psi(x,z))$, where Φ and Ψ are conjunctive queries. However, in our context we serialize such mappings in the graph $\mathcal{M}$, and not as separated logical expressions. Hence, we define a LAV mapping for a wrapper w as $LAV(w) : w \mapsto \varphi_{\mathcal{G}}$, where $\varphi_{\mathcal{G}}$ is a subgraph of $\mathcal{G}$. We additionally consider a function $F : a_w \mapsto a_m$, that translates the name of an attribute in $\mathcal{S}$ to its corresponding conceptual representation in $\mathcal{G}$. Such function allows us to denote semantic equivalence between physical and conceptual attributes in the ontology (respectively, in $\mathcal{S}$ and $\mathcal{G}$). Intuitively, F forces a physical attribute in the sources to map to one and only one conceptual feature in $\mathcal{G}$. As schema mappings, this function is also serialized in $\mathcal{M}$.

Example 2.4
The LAV mapping for w_1 would be the subgraph $Monitor \xrightarrow[generatesQoS]{} InfoMonitor$ (also including all class attributes). Regarding F, the function would make the conversions w_1.VoDmonitorId $\mapsto toolId$ and w_1.lagRatio $\mapsto lagRatio$.

3 Big Data Integration Ontology

In this section, we present the Big Data Integration ontology (BDI), the metadata artifact that enables a systematic approach for the data integration system governance when ingesting and analysing the data. To this end, we have followed the well-known theory on data integration [98] and divided it into two levels (by means of RDF named graphs): the Global and Source graphs, respectively $\mathcal{G}$ and $\mathcal{S}$, linked via mappings $\mathcal{M}$. Thanks to the extensibility of RDF, it further enables us to enrich $\mathcal{G}$ and $\mathcal{S}$ with semantics such as data types. In this section we present the RDF vocabulary to be used to represent $\mathcal{G}$ and $\mathcal{S}$. To do so, we present a metamodel for the global and source ontologies that current models (i.e., $\mathcal{G}$ and $\mathcal{S}$) must mandatorily follow. In the following subsections, we elaborate on each graph and present its RDF representation.

3.1 Global graph

The Global graph $\mathcal{G}$ reflects the main domain concepts, relationships among them and features of analysis (i.e., maps to the role of a UML diagram in a machine-readable format). Its elements are defined in terms of the vocabulary users will use when posing queries. The metadata model for $\mathcal{G}$ distinguishes concepts from features, the former mimicking classes and the latter attributes in a UML diagram. Concepts can be linked by means of domain-specific object properties, which implicitely determine their domain and range. Such properties will be used for data analysts to navigate the graph, dismissing the need of specifying how the underlying sources are joined. The link between a concept and its set of features is represented via `G:hasFeature`. In order to disambiguate the query rewriting process we restrict features to belong to only one concept. Additionally, it is possible to define a taxonomy of features, which will denote related semantic domains (e.g., the feature `sup:monitorId` is subclass of `sc:identifier`). Features can be enriched with new semantics to aid the data management and analysis phases. In this thesis, we narrow the scope to data types for features, widely used in data integrity management.

Code 3.6 provides the triples that compose $\mathcal{G}$ in Turtle RDF notation[7]. It contains the main metaclasses (using the namespace prefix `G`[8] as main namespace) which all features of analysis will instantiate. Concepts and features can reuse existing vocabularies by following the principles of the Linked Data (LD) initiative. Additionally, we include elements for data types on features linked using `G:hasDatatype`, albeit their maintenance is out of the scope of this thesis. Following the same LD philosophy, we reuse the `rdfs:Datatype` class to instantiate data types. With such design, we favor the

[7] `https://www.w3.org/TR/turtle`

[8] `http://www.essi.upc.edu/~snadal/BDIOntology/Global`

elements of $\mathcal{G}$ to be of any of the available types in XML Schema (prefix xsd[9]). Finally, note that we focus on non-complex data types, however our model can be easily extended to include complex types [41].

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix voaf: <http://purl.org/vocommons/voaf#> .
@prefix vann: <http://purl.org/vocab/vann/> .
@prefix G: <http://www.essi.upc.edu/~snadal/BDIOntology/Global/> .

<http://www.essi.upc.edu/~snadal/BDIOntology/Global/> rdf:type voaf:Vocabulary ;
       vann:preferredNamespacePrefix "G";
       vann:preferredNamespaceUri "http://www.essi.upc.edu/~snadal/BDIOntology/Global";
       rdfs:label "The␣Global␣graph␣vocabulary" .

G:Concept rdf:type rdfs:Class;
       rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Global/> .

G:Feature rdf:type rdfs:Class;
       rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Global/> .

G:hasFeature rdf:type rdf:Property ;
       rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Global/> ;
       rdfs:domain G:Concept ;
       rdfs:range G:Feature .

G:hasDataType rdf:type rdf:Property ;
       rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Global/> ;
       rdfs:domain G:Feature ;
       rdfs:range rdfs:Datatype .
```

Code 3.6: Metadata model for $\mathcal{G}$ in Turtle notation

Example 3.1
Figure 3.3 depicts the instantiation of $\mathcal{G}$ in the SUPERSEDE case study, as presented in the UML diagram in Figure 3.2 (for the sake of conciseness only a fragment is depicted). The color of the elements represent typing (i.e., `rdf:type` links). Note that, in order to comply with the design constraints of $\mathcal{G}$ (i.e., a feature can only belong to one concept), the *toolId* feature has been explicited and made distinguishable to `sup:monitorId` and `sup:feedbackGatheringId` respectively for classes *Monitor* and *FeedbackGathering*. When possible, vocabularies are reused, namely `https://www.w3.org/TR/vocab-duv` (prefix `duv`) for feedback elements as well as `http://dublincore.org/documents/dcmi-terms` (prefix `dct`) or `http://schema.org` (prefix `sc`). However, when no vocabulary is available we define the custom SUPERSEDE vocabulary (prefix sup).

[9]`http://www.w3.org/2001/XMLSchema`

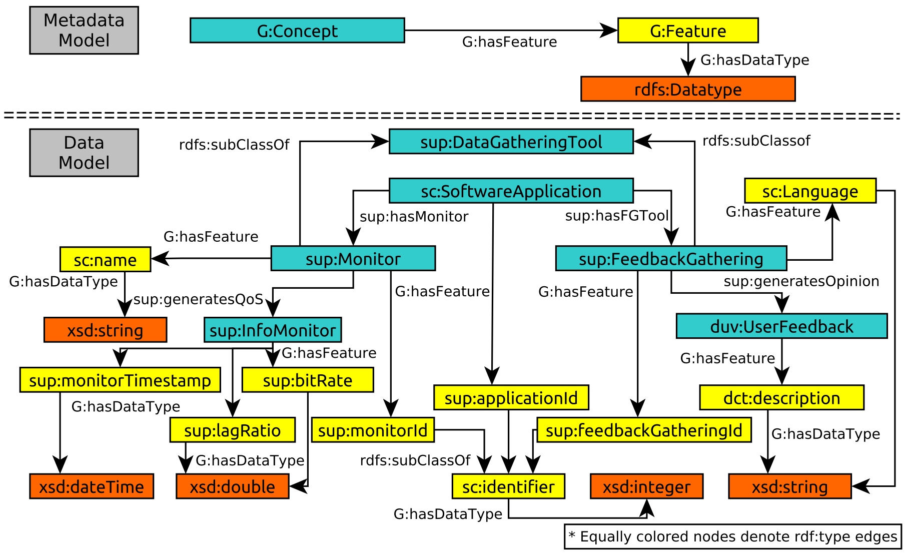


Fig. 3.3: RDF dataset of the metadata model and data model of $\mathcal{G}$ for the SUPERSEDE running example.

3.2 Source graph

The purpose of the Source graph $\mathcal{S}$ is to model the different wrappers and their provided schema. To this end, we define the metaconcept `S:DataSource` which models the different data sources (e.g., Twitter REST API). In $\mathcal{S}$, we additionally encode the necessary information for schema versioning, hence we define the metaconcept `S:Wrapper` which will model the different schema versions for a data source, which in turn consist of a representation of the projected attributes, modeled in the metaconcept `S:Attribute`. We embrace the reuse of attributes within wrappers of the same data source, as we assume the semantics do not differ across schema versions, however that assumption is not realistic among different data sources (e.g., not necessarily a timestamp has the same meaning in the VoD monitor and the Twitter API). Therefore, we encode in the attribute names the prefix of the data source they correspond to (e.g., for a data source D, its wrappers W and W' respectively provide the attributes {`D/a`,`D/b`} and {`D/a`,`D/c`}). Code 3.7 depicts the metadata model for $\mathcal{S}$ in Turtle RDF notation (using prefix `S`[10] as main namespace).

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix voaf: <http://purl.org/vocommons/voaf#> .
@prefix vann: <http://purl.org/vocab/vann/> .
```

[10] http://www.essi.upc.edu/~snadal/BDIOntology/Source

```
@prefix S: <http://www.essi.upc.edu/~snadal/BDIOntology/Source/> .

<http://www.essi.upc.edu/~snadal/BDIOntology/Source/> rdf:type voaf:Vocabulary ;
       vann:preferredNamespacePrefix "S";
       vann:preferredNamespaceUri "http://www.essi.upc.edu/~snadal/BDIOntology/Source";
       rdfs:label "The␣Source␣graph␣vocabulary" .

S:DataSource rdf:type rdfs:Class;
       rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Source/> .

S:Wrapper rdf:type rdfs:Class;
       rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Source/> .

S:Attribute rdf:type rdfs:Class;
       rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Source/> .

S:hasWrapper rdf:type rdf:Property ;
       rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Source/> ;
       rdfs:domain S:DataSource ;
       rdfs:range S:Wrapper .

S:hasAttribute rdf:type rdf:Property ;
       rdfs:isDefinedBy <http://www.essi.upc.edu/~snadal/BDIOntology/Source/> ;
       rdfs:domain S:Wrapper ;
       rdfs:range S:Attribute .
```

Code 3.7: Metadata model for $\mathcal{S}$ in Turtle notation

Example 3.2
Figure 3.4 shows the instantiation of $\mathcal{S}$ in SUPERSEDE. Red nodes depict the data sources that correspond to the three data sources introduced in Section 2.1. Then, orange and blue nodes depict the wrappers and attributes, respectively.

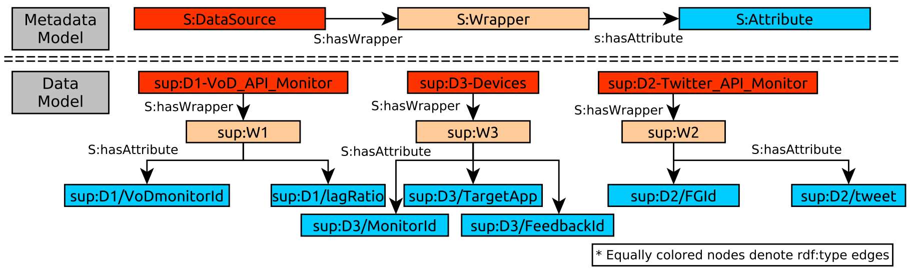


Fig. 3.4: RDF dataset of the metadata model and data model of $\mathcal{S}$.

3.3 Mapping graph

As previously discussed, we encode LAV mappings in the ontology. Recall that mappings are composed of (a) subgraphs of $\mathcal{G}$, one per wrapper, and

(b) the function F linking elements of type `S:Attribute` to elements of type `G:Feature`. We serialize such information in RDF in the Mapping graph $\mathcal{M}$. Subgraphs are represented using named graphs, which identify a subset of $\mathcal{G}$. Thus, each wrapper will have associated a named graph identifying which concepts and features it is providing information about. This will be represented using triples of the form $\langle w,$`M:mapping`$, G\rangle$, where w is an instance of `S:Wrapper` and G is a subgraph of $\mathcal{G}$. Regarding the function F, we represent it via the `owl:sameAs` property (i.e., triples of the form $\langle$`x`$,$`owl:sameAs`$,y\rangle$, where `x` and y are respectively instances of `S:Attribute` and `G:Feature`.

Example 3.3
In Figure 3.5 we depict the complete instantiation of the BDI ontology for the SUPERSEDE running example. To ensure readability, internal classes are omitted and only the core ones are shown. Named graphs are depicted using colored boxes, respectively red for w_1, blue for w_2 and green for w_3.

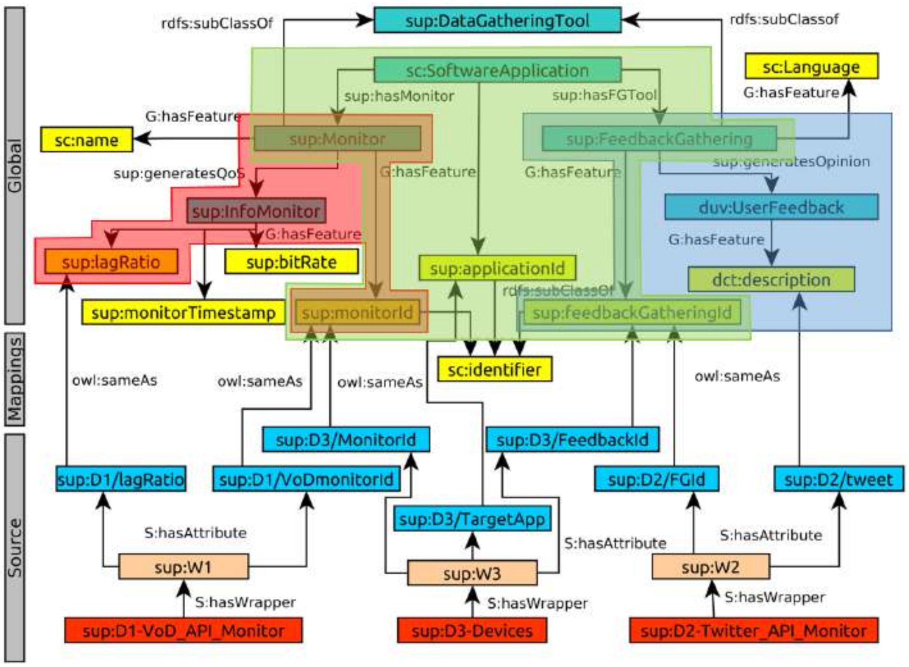


Fig. 3.5: RDF dataset of the metadata model and data model of the complete ontology for the SUPERSEDE running example.

The previous discussion sets the baseline to enable semi-automatic schema management in the data sources. Instantiating the metadata model, the data steward is capable of modeling the schema of the sources to be further linked to the wrappers and the data instances they provide. With such, in the rest of this chapter we will introduce techniques to adapt the ontology to schema evolution as well as query answering.

4 Handling Evolution

In this section, we present how the BDI ontology accomodates the evolution of situational data. Specific studies concerning REST API evolution [101, 162] have concluded that most of such changes occur in the structure of incoming events, thus our goal is to semi-automatically adapt the BDI ontology to such evolution. To this end, in the following subsections we present an algorithm to aid the data steward to enrich the ontology upon new releases.

4.1 Releases

In Section 2, we discussed the role of the data steward as the unique maintainer of the BDI ontology in order to make data management tasks transparent to data analysts. Now, the goal is to shield the analysts queries, so that they do not crash upon new API version releases. In other words, we need to adapt $\mathcal{S}$ to schema evolution in the data sources, so that $\mathcal{G}$ is not affected. To this end, we introduce the notion of *release*, the construct indicating the creation of a new wrapper, and how its elements link to features in $\mathcal{G}$. Thus, we formally define a release R as a 3-tuple $R = \langle w, G, F \rangle$, where w is a wrapper, G is a subgraph of $\mathcal{G}$ denoting the elements in $\mathcal{G}$ that the wrapper contributes to, and $F = a \mapsto V(G)$ a function where $a \in w.\overline{a_{ID}} \cup w.\overline{a_{nID}}$ and $V(G)$ vertices of type `G:Feature` in $\mathcal{G}$. R must be created by the data steward upon new releases. Several approaches can aid this process. For instance, to define the graph G, the user can be presented with subgraphs of $\mathcal{G}$ that cover all features. However, this raises the question of which is the most appropiate subgraph that the user is interested in. Regarding the definition of F, probabilistic methods to align and match RDF ontologies, such as PARIS [149], can be used. Note that the definition of wrappers (i.e., how to query an API) is beyond the scope of this thesis.

Example 4.1
Recall wrapper w_4 for data source D_1. Its associated release would be defined as $w_4(\mathsf{VoDmonitorId}, \mathsf{bufferingRatio})$, $G = \texttt{sup:lagRatio} \xleftarrow[\texttt{G:hasFeature}]{} \texttt{sup:InfoMonitor} \xleftarrow[\texttt{sup:generatesQoS}]{} \texttt{sup:Monitor} \xrightarrow[\texttt{G:hasFeature}]{} \texttt{sup:monitorId}$, and $F = \{\mathsf{VoDmonitorId} \mapsto \texttt{sup:monitorId}, \mathsf{bufferingRatio} \mapsto \texttt{sup:lagRatio}\}$.

4.2 Release-based ontology evolution

As mentioned above, changes in the source elements need to be reflected in the ontology to avoid queries to crash. Furthermore, the ultimate goal is to provide such adaptation in an automated way. To this end, Algorithm 1 applies the necessary changes to adapt the BDI ontology $\mathcal{T}$ w.r.t. a new release R. It starts registering the data source, in case it is new (line 4), and the new wrapper to further link them (lines 7 and 8). Then, for each attribute in the wrapper $R.w$, we check their existence in the current Source graph and register it, in case it is not present. Given the way URIs for attributes are constructed (i.e., they have the prefix of their source), we can ensure that only attributes from the same source will be reused within subsequent versions. This helps to maintain a low growth rate for $\mathcal{T}.\mathcal{S}$, as well as avoiding potential semantic differences. Next, the named graph is registered to the Mapping graph, to conclude with the serialization of function F (in $R.F$). The complexity of this algorithm is linearly bounded by the size of the parameters of R.

Algorithm 1 Adapt to Release

```
Input: $\mathcal{T}$ is the BDI ontology, $R$ new release
Output: $\mathcal{T}$ is adapted w.r.t. $R$
 1: function NewRelease($\mathcal{T}$, $R$)
 2:   $Source_{uri}$ = "S:DataSource/"+$source(R.w)$
 3:   if $Source_{uri} \notin$ SELECT ?ds FROM $\mathcal{T}$ WHERE ⟨?ds, "rdf:type", "S:DataSource"⟩ then
 4:     $\mathcal{T}.\mathcal{S} \cup=$ ⟨$Source_{uri}$, "rdf:type", "S:DataSource"⟩
 5:   $Wrapper_{uri}$ = "S:Wrapper/"+$R.w$
 6:   $\mathcal{T}.\mathcal{S} \cup=$ ⟨$Wrapper_{uri}$, "rdf:type", "S:Wrapper"⟩
 7:   $\mathcal{T}.\mathcal{S} \cup=$ ⟨$Source_{uri}$, "S:hasWrapper", $Wrapper_{uri}$⟩
 8:   for each $a \in (R.w.\overline{a_{ID}} \cup R.w.\overline{a_{nID}})$ do
 9:     $Attribute_{uri}$ = $Source_{uri}$+$a$
10:     if $Attribute_{uri} \notin$ SELECT ?a FROM $\mathcal{T}$ WHERE ⟨?a, "rdf:type", "S:Attribute"⟩ then
11:       $\mathcal{T}.\mathcal{S} \cup=$ ⟨$Attribute_{uri}$, "rdf:type", "S:Attribute"⟩
12:     $\mathcal{T}.\mathcal{S} \cup=$ ⟨$Wrapper_{uri}$, "S:hasAttribute", $Attribute_{uri}$⟩
13:   $\mathcal{T}.\mathcal{M} \cup=$ ⟨$Wrapper_{uri}$, "M:mapping", $R.G$⟩
14:   for each $(a, f) \in R.F$ do
15:     $a_{uri}$ = $Source_{uri}$+$a$
16:     $f_{uri}$ = "G:Feature/"+$f$
17:     $\mathcal{T}.\mathcal{M} \cup=$ ⟨$a_{uri}$, "owl:sameAs", $f_{uri}$⟩
```

Example 4.2
In Figure 3.6, we depict the resulting ontology $\mathcal{T}$ after executing Algorithm 1 with the release for wrapper w_4.

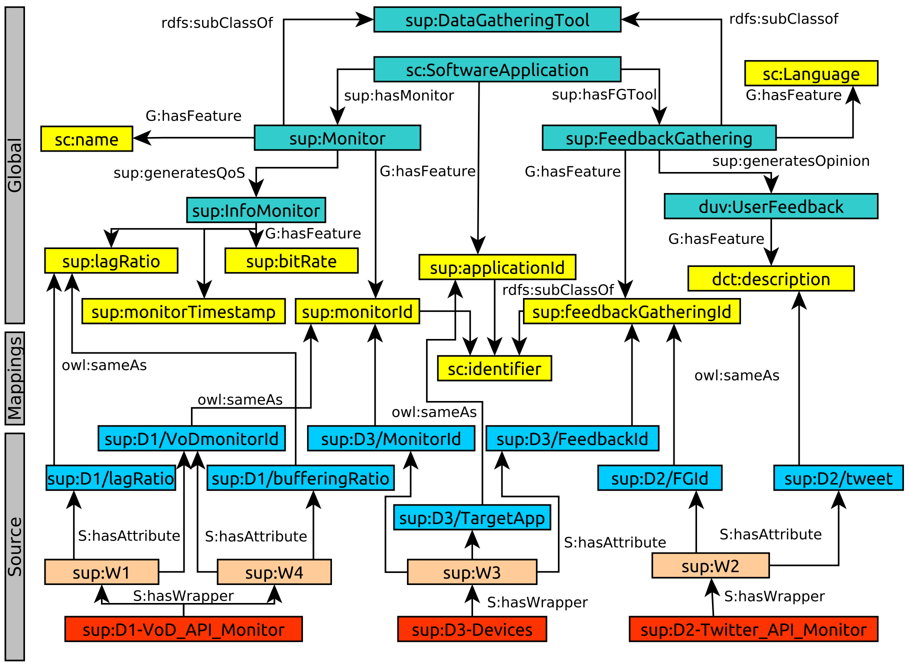


Fig. 3.6: RDF dataset for the evolved ontology $\mathcal{T}$ for the SUPERSEDE running example. Colored subgraphs are the same as Figure 3.5, the one for w_4 being the same than for w_1

5 Evaluation

In this section, we present the evaluation results of our approach. We provide three kinds of evaluations: a functional evaluation on evolution management, the industrial applicability of our approach and a study on the evolution of the ontology in a real-world API.

5.1 Functional evaluation

In order to evaluate the functionalities provided by the BDI ontology, we take the most recent study on structural evolution patterns in REST API [162]. Such work distinguishes changes at 3 different levels, those in (a) API-level, (b) method-level and (c) parameter-level. Our goal is to demostrate that our approach can semi-automatically accommodate such changes. To this end, it is necessary to make a distinction between those changes occurring in the data requests and those in the response. The former are handled by the wrapper's underlying query engine, which also needs to deal with other aspects such as authentication or HTTP query parametrization. The latter will be handled by the proposed ontology.

API-level changes

Those changes concern the whole of an API. They can be observed either because a new data source is incorporated (e.g., a new social network in the SUPERSEDE use case) or because all methods from a provider have been updated. Table 3.2 depicts the API-level change breakdown and the component responsible to handle it.

API-level Change	**Wrapper**	**BDI Ont.**
Add authentication model	✓	
Change resource URL	✓	
Change authentication model	✓	
Change rate limit	✓	
Delete response format		✓
Add response format		✓
Change response format		✓

Table 3.2: API-level changes dealt by wrappers or BDI ontology

Adding or changing a response format at API level consists of, for each wrapper querying it, registering a new release with this format. Regarding the deletion of a response format, it does not require actions, due to the fact that no further data on such format will arrive. However, in order to preserve historic backwards compatibility, no elements should be removed from $\mathcal{T}$.

Method-level changes

Those changes concern modifications on the current version of an operation. They occur either because a new functionality is released or because existing functionalities are modified. Table 3.3 summarizes the method-level change breakdown and the component responsible to handle it.

Those changes have more overlapping with the wrappers due to the fact that new methods require changes in both request and response. In the context of the BDI ontology, each method is an instance of `S:DataSource` and thus, adding a new one consists of declaring a new release and running Algorithm 1. Renaming a method requires renaming the data source instance. As before, a removal does not entail any action with the aim of preserving backwards historic compatibility.

Method-level Change	**Wrapper**	**BDI Ont.**
Add error code	✓	
Change rate limit	✓	
Change authentication model	✓	
Change domain URL	✓	
Add method	✓	✓
Delete method	✓	✓
Change method name	✓	✓
Change response format		✓

Table 3.3: Method-level changes dealt by wrappers or BDI ontology

Parameter-level changes

Such changes are those concerning schema evolution and are the most common on new API releases. Table 3.4 depicts such changes and the component in charge of handling it.

Parameter-level Change	**Wrapper**	**BDI Ont.**
Change rate limit	✓	
Change require type	✓	
Add parameter	✓	✓
Delete parameter	✓	✓
Rename response parameter		✓
Change format or type		✓

Table 3.4: Parameter-level changes dealt by wrappers or BDI ontology

Similarly to the previous level, some parameter-level changes are managed by both wrappers and the ontology. This is caused by the ambiguity of the change statements, and hence we might consider both URL query parameters and response parameters (i.e., attributes). Changing format of a parameter has a different meaning as before, and here entails a change of data type or structure. Any of the parameter-level changes identified can be automatically handled by the same process of creating a new release for the source at hand.

5.2 Industrial applicability

After functionally validating that the BDI ontology and wrappers can handle all types of API evolution, next we aim to study how these changes occur in real-world APIs. With this purpose, we study the results from [101] which presents 16 change patterns that frequently occur in the evolution of 5 widely used APIs (namely *Google Calendar*, *Google Gadgets*, *Amazon MWS*, *Twitter API* and *Sina Weibo*). With such information, we can show the number of changes per API that could be accommodated by the ontology. We summarize the results in Table 3.5. As before, we distinguish between changes concerning (a) the wrappers, (b) the ontology and (c) both wrappers and ontology. This enables us to measure the percentage of changes per API that can be partially accommodated by the ontology (changes also concerning the wrappers) and those fully accommodated (changes only concerning the ontology). Our results show that for all studied APIs, the BDI ontology could, on average, partially accommodate 48.84% of changes and fully accommodate 22.77% of changes. In other words, our semi-automatic approach allows to solve on average 71.62% of changes.

API Owner	#Changes Wrapper	#Changes Ontology	#Changes Wrapper&Ontology	Partially Accommodates	Fully Accommodates
Google Calendar	0	24	23	48.94%	51.06%
Google Gadgets	2	6	30	78.95%	15.79%
Amazon MWS	22	36	14	19.44%	50%
Twitter API	27	0	25	48.08%	0%
Sina Weibo	35	3	56	59.57%	3.19%

Table 3.5: Number of changes per API and percentage of partially and fully accommodated changes by $\mathcal{T}$

5.3 Ontology evolution

Now, we are concerned with performance aspects of using the ontology. Particularly, we will study its temporal growth w.r.t. the releases of a real-world API, namely Wordpress REST API[11]. This analysis is of special interest, considering that the size of the ontology may have a direct impact on the cost of querying and maintaining it. As a measure of growth, we count the number of triples in $\mathcal{S}$ after each new release, as it is the most prone to change. Given the high complexity of such APIs, we focus on a specific method and study its structural changes, namely the *GET Posts* API. By studying the changelog, we start from the currently deprecated version 1 evolving it to the next major version release 2. We further introduce 13 minor releases of version 2. (the details of the analysis can be found in [118]). We assume that a new wrapper providing all attributes is defined for each release.

The barcharts in Figure 3.7 depict the number of triples added to $\mathcal{S}$ per version release. As version 1 is the first occurrence of such endpoint, all elements must be added and thus carries a big overhead. Version 2 is a major release where few elements can be reused. Later, minor releases do not have many schema changes, with few attribute additions, deletions or renames. Thus, the largest batch of triples per minor release are edges of

[11]`https://wordpress.org/plugins/rest-api`

type `S:hasAttribute`. Each new version needs to identify which attributes it provides even though no change has been applied to it w.r.t. previous versions.

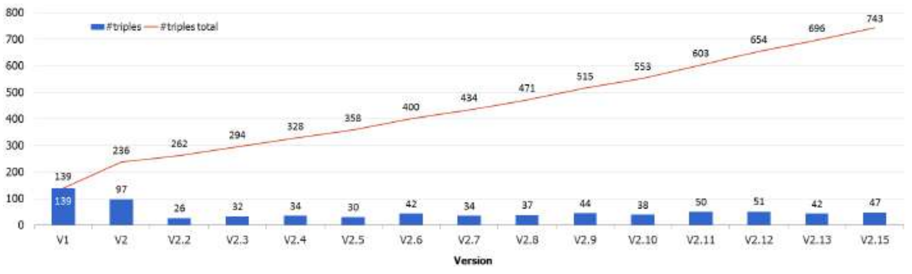


Fig. 3.7: Growth in number of triples for $\mathcal{S}$ per release in Wordpress API

With such analysis we conclude that major version changes entail a steep growth, however that is infrequent in the studied API. On the other hand, minor versions occur frequently but the growth in terms of triples has a steady linear growth. The red line depicts the cumulative number of triples after each release. For a practically stable amount of minor release versions, we obtain a linear, stable growth in $\mathcal{S}$. Notice also that $\mathcal{G}$ does not grow. Altogether guarantees that querying $\mathcal{T}$ in query answering will not impose a big overhead, ensuring a good performance of our approach across time. Nonetheless, other optimization techniques (e.g., caching) can be used to further reduce the query cost.

6 Related Work

In previous sections, we have cited relevant works on RESTful API evolution [162, 101]. They provide a catalog of changes, however they do not provide any approach to systematically deal with them. Other similar works, such as [171], empirically study API evolution aiming to detect its healthiness. If we look for approaches that automatically deal with such evolution, we must shift the focus to the area of database schemas, which are mostly focused on relational databases [144, 107]. They apply view cloning to accommodate changes while preserving old views. Such techniques rely on the capability of vetoing certain changes that might affect the overall integrity of the system. This is however an unrealistic approach to adopt in our setting, as schema changes are done by third party data providers.

Attention has also been paid to change management in the context of description logics (DLs). The definition of a DL that provides expresiveness to represent temporal changes in the ontology has been an interesting topic of study in the past years [105]. Relevant examples include [16], that defines the temporal DL *TQL*, providing temporal aspects at the conceptual model level, or [89] that delves on how to provide such temporal aspects for specific attributes in a conceptual model. It is known, however, that providing such temporal aspects to DLs entails a poor computational behaviour for CQ answering [105], for instance the previous examples are respectively coNP-*hard* and undecidable. Recent efforts are being put to overcome such issues and to provide tractable DLs and methods for rewritability of OMQs. For instance, [15] provides a temporal DL where the cost of first-order rewritability is polynomial, however that is only applicable for a restricted fragment of *DL-Lite*, and besides the notion of temporal attribute, which is key for management of schema evolution does not exist. Generally speaking, most of this approaches lack key characteristics for the management of schema evolution [122].

Regarding LAV schema mappings in data integration, few approaches strictly follow its definition. This is mostly due to the inherent complexity of query answering in LAV, which is reduced to the problem of answering queries using views [99]. Probably the most prominent data integration system that follows the LAV approach is Information Manifold [93]. To overcome the complexity posed by LAV query answering, combined approaches of GAV and LAV have been proposed, which are commonly referred as *both-as-view* (BAV) [112] or *global-and-local-as-view* (GLAV) [49]. Oppositely, we are capable of adopting a purely LAV approach by restricting the kind of allowed queries as well as how the mediated schema (i.e., ontology) has to be constructed.

7 Conclusions

In this chapter we have presented the building blocks to handle schema evolution using a vocabulary-based approach to OBDA. Thus, unlike current OBDA approaches, we restrict the language from generic knowledge representation ontology languages (such as DL-Lite) to ontologies based on RDF vocabularies. This enables us to adopt LAV mappings instead of the classical GAV. The proposed Big Data integration ontology aims to provide data analysts with an RDF-based conceptual model of the domain of interest, with the limitations that features cannot be reused among concepts. Data sources are accessed via wrappers, which must expose a relational schema in order to depict its RDF-based representation in the ontology and define LAV mappings, by means of named graphs and links from attributes to features. We have presented an algorithm to aid data stewards to systematically accommodate announced changes in the form of releases. Our evaluation results show that a great number of changes performed in real-world APIs could be semi-automatically handled by the wrappers and the ontology. We additionally have shown the feasability of our query answering algorithm.

Chapter 4

Answering Queries Using Views Under Semantic Heterogeneities and Evolution

This chapter is under submission for ACM SIGMOD/PODS International Conference on Management of Data (2019).
The layout of the paper has been revised.

Abstract

The data variety challenge refers to the complexity of managing and integrating a set of heterogeneous and evolving Big Data sources. Traditional data integration techniques, that focus on a rather static setting, fail to address such Big Data integration problems where a higher degree of autonomy is expected from the system. In this chapter, we address the problem of query processing in the presence of data variety. We precisely focus on processing queries that are agnostic of the semantic heterogeneities in the extensions of the sources as well as the evolution of data sources and their schemata. The former, is a predominant case for event data ingested at different granularity levels, while the latter occurs when queries access external sources that continuously change. To obtain both characteristics, we adopt a graph-based approach to represent the integration system with a restricted set of semantic annotations. The proposed query rewriting algorithm uses such annotations to resolve queries in the presence of the problems above. We theoretically and experimentally validate our approach showing its soundness and completeness.

1 Introduction

Big Data integration (BDI) refers to the wide set of techniques, methods and tools that go beyond traditional data integration to combine and unify Big Data sources [40]. Nowadays, for example organizations enhance their analytical pipelines by combining in-house and external data. These data are commonly ingested from third party providers, such as social networks, via REST APIs or other web protocols. Here, data are neither generated nor under control of the organization, thus it is paramount to provide a higher degree of autonomy to deal with semantic heterogeneities and unavoidable changes. For instance, Facebook's Graph API yields information related to users' posts[1] where the `created_at` attribute is encoded using a `datetime` format (i.e., millisecond granularity), however Facebook's marketing click tracking API[2] yields aggregated information for a specific day. Furthermore, in 2018 Facebook's Graph API suffered at least eleven breaking changes[3]. A natural desiderata in this setting is to enable data analysts to easily query such data disregarding the need of writing complex data transformations or safeguard mechanisms that avoid failure. This scenario requires providing on-demand integration of an heterogeneous and evolving set of data sources, which falls under the umbrella of the *data variety challenge*. This has been designated as a key success factor for Big Data projects [21]. The ultimate goal of the data variety challenge is to enable data analysts, with domain expertise but not proficiency in data management, to easily explore data sources, and thus democratize data access within an organization.

Classic data integration techniques have focused on the definition of a *global schema* that mediates queries over a static and generally structured set of data sources (i.e., the *source schemata*) [98]. This amounts to the problem of answering queries using views, whose goal is to rewrite queries posed over the global schema into an equivalent set of queries over the sources [69]. Yet, some of the assumptions from the classical integration setting are no longer valid and must be rethought for BDI [1, 56]. An observed recurring demand from data analysts in a number of BDI projects is to include into their query results all available data making transparent how it is originated in the sources. We identify two new major challenges, with respect to traditional data integration, that need to be dealt with in order to enable such autonomous and agnostic query processing:

- **Semantic heterogeneities.** In Big Data scenarios, event *event data* generated by sensors, monitors or logs are highly predominant [65]. Thus, it is common that different sources report data at different levels of gener-

[1]https://developers.facebook.com/docs/graph-api/reference/v3.1/post
[2]https://developers.facebook.com/docs/marketing-api/click-tags
[3]https://developers.facebook.com/docs/graph-api/changelog/breaking-changes

alization/specialization as well as aggregation/decomposition [125]. For instance, different sources might have varying sampling rates such as millisecond, second or minute. However, if the analyst is only interested in observing data at the hourly level, then the system should automatically aggregate all available data to this specific granularity.

- **Data source and schema evolution.** Due to the unprecedented growth in the number of data providers, the management of data source evolution requires to ensure queries are agnostic and gracefully adapt to the addition or removal of new data sources. Furthermore, given that such providers continuously evolve their services (e.g., new API releases), and consequently modify the schema or format provided in previous versions, we have to maintain coexisting schema versions in the integration system and automatically expand queries over them.

Consequently, in this chapter we focus on the problem of answering queries using views that are affected by semantic heterogeneities and evolution. Unfortunately, to our knowledge there is no related work that can help us address the problem easily and elegantly. There exist two major approaches to data integration; either physically materializing the content of the sources in an integrated repository, or virtualizing their schema and mediating queries over them. Online analytical processing (OLAP) tools are the most well known representatives of materialized integration systems. However, in BDI, where warehousing processes can be very expensive due to the high degree of autonomy in the sources, the predominant option is that of virtual integration across heterogeneous data models, coined as *polystores* [150]. For virtual integration, the kind of adopted schema mappings, either *global-as-view* (GAV) or *local-as-view* (LAV), will directly determine how the sources are modeled and queries processed [29]. One prominent family of virtual integration approaches are *ontology-based data access* (OBDA), which adopt ontologies, represented in a *description logic* (DL), on the target schema [130]. In OBDA, ontological assertions allow to enrich query results with extra knowledge (e.g., inheritance can be easily dealt with), allowing to find the certain answers of a query under the *open-world assumption* (OWA). OBDA generally adopts GAV mappings, characterizing concepts of the global schema in terms of queries over the sources. The task of query processing is reduced to mapping unfolding, however high variability in the data sources might entail continuous maintenance of mappings, making GAV not suitable.

Considering these premises, we propose a novel BDI approach to tackle query processing under semantic heterogeneities and evolution. Our main contribution is a query rewriting algorithm that transforms queries over the global schema to equivalent ones that include or discard semantically heterogeneous data sources. To deal with evolution, we advocate for the adoption of LAV mappings, that characterize elements of the source schemata in terms of a query over the global schema. They are inherently more suitable for dynamic environments like ours, where adding, removing or updating a source requires modifying only one mapping definition. We adopt a graph-based structure to represent the complete integration system. The benefits of using a graph formalism are twofold: first, all required metadata (i.e., global schema, mappings and source descriptions) are encoded in a single data structure, which simplifies the interoperability among them; second, graphs offer the flexibility to easily encode any required annotation for the rewriting process in the domains of vertices and edges. Precisely, the global schema (i.e., the *global graph*) encodes the domain of interest, as well as *semantic annotations* that cover the most relevant conceptual modeling constructs (i.e., association, specialization and aggregation). Data sources are accessed via wrappers, which act to us as views that hide the complexity of accessing the set of heterogeneous data sources [134]. The *source graph* is an accurate graph-based representation of wrappers and their attributes. To link the source and global graphs we encode LAV schema mappings as subgraphs. We precisely assume complete sources, an approach known as the *closed world assumption* (CWA).

The proposed algorithm deals with the semantic heterogeneities of specialization and aggregation by generating sets of queries that request data at lower granularity levels. These are rewritten to equivalent unions of conjunctive queries over the wrappers. The rewriting process is driven by semantic annotations in the global graph and the mappings, which permit to satisfy the properties of *minimally-soundness* and *minimally-completeness*. Next, we perform *implicit aggregations* on such rewritings to align available data with the granularity required in the original query. This is transparent to data analysts, who are not necessarily aware of the different granularity levels of data.

In particular, our main contributions are as follows.

- We present a novel BDI approach that encodes all required metadata (i.e., global schema, LAV mappings and wrapper descriptions) in a graph structure. The proposed structure maps to a subset of the BDI ontology presented in the previous chapter, here focusing on the necessary constructs for query rewriting.

- We introduce an algorithm that given a query, deals with the semantic heterogeneities of specialization and aggregation, by generating a set of equivalent queries.

- We present a query rewriting algorithm that resolves LAV mappings and translates graph-based queries to equivalent unions of conjunctive queries over the wrappers.

- We provide theoretical and experimental validations of our approach. First, proving the completeness and soundness of the rewriting algorithms, and later showing the practical efficiency in a real Big Data setting with high variety.

Outline. The rest of the chapter is structured as follows. We discuss additional related work in Section 2 and introduce background concepts in Section 3. Then, in Section 4 and 5 we respectively present the rewriting algorithms for CQs and CAQs. Next, in Section 6 we experimentally validate our approach. Finally, we present the conclusions in Section 7.

2 Related Work

The problem of answering queries using views has set the theoretical underpinnings for several data integration approaches [69]. Multiple research areas exist depending on the underlying assumptions made. We categorize these across three dimensions: first, whether the target schema is physically materialized or virtual; second, the type of mappings used to link target and source schemata (i.e., GAV, LAV or GLAV); and last how source incompleteness is dealt with (i.e., OWA or CWA). The most similar approaches to our setting are those performing virtual integration based on LAV mappings and assuming complete sources (i.e., CWA). There exist several algorithms for LAV mediation, with the *bucket algorithm* [100], the *inverse rules algorithm* [42] and the *MiniCon algorithm* [131] being the most prominent ones. All these algorithms are datalog-based to yield sets of maximally-contained query rewritings. To this end, conjuncts in the body of the query are considered subgoals that need to be isolately processed and further combined. How subgoals are resolved, and how rewritings are combined differs among the algorithms.

Data warehouses. A data warehouse (DW) is the best example of a materialized integration system where queries are evaluated under the CWA. Queries over a DW leverage a lattice structure such that data can be implicitly aggregated at multiple dimensional levels [72]. Related to our problem of interest, several extensions to DWs have been proposed for instance to include domain knowledge via ontologies [3]; or related to the management of evolution to detect and automatically fix inconsistencies upon changes [107].

Data exchange. The data exchange (DE) problem consists of materializing instances from a source schema S to a target schema T, such that the set of source-to-target constraints ($\Sigma_{s\text{-}t}$) and target constraints (Σ_t) are satisfied [12]. In general GLAV mappings are adopted, and query answering consists of computing the set of certain answers, generally assuming OWA, and evaluating the query over T. In practice, this is achieved by computing the *chase* over the instances in S, a method that systematically extends dependencies and generates new facts until all dependencies are satisfied. However, the scalability of this method is still one of its major drawbacks [23]. Answering aggregate queries has also been studied in this context, defining different semantics for aggregate queries in the presence of incompleteness (e.g., range semantics based on database repairs [13], or based on the endomorphic images of the canonical universal solution [4]).

Ontology-based data access. OBDA implements a virtual integration approach using ontologies. To this end, they adopt the *DL-Lite* family of DLs as foundation, a well-behaved fragment capturing a fair portion of conceptual modeling formalisms, and guarantee *first-order rewritability* of ontology-mediated queries [14]. The ontology can be leveraged to complement query results with further knowledge, thus being able to compute the certain answers under the OWA. Thanks to adopting GAV mappings, the query answering task is reduced to an unfolding process of mappings. Explicit aggregate queries have also been studied in an OBDA context. In [31] the authors propose epistemic semantics for aggregate queries, which aggregate only certainly known values. As pointed out in [96] such semantics might yield incorrect answers for *count* aggregates. To overcome this, the authors propose *aggregate certain answers* semantics which are more suitable when counting.

As conclusions of this succint related work study we acknowledge that, to the best of our knowledge, there is no work considering the intersection of our problems of interest (i.e., dealing with semantic heterogeneties to implicitly aggregate data and management of evolution) in a virtual data integration environment.

3 Preliminaries

3.1 Case study

As a case study, we take the SUPERSEDE[4] project, which will serve as running example throughout the chapter. SUPERSEDE aims to support decision-making in the evolution and adaptation of software services by exploiting end-user feedback and monitoring runtime data. This project is characterized by a high variety in the number and type of sources. One of its main technical challenges is to obtain aggregated quality metrics in a constantly evolving set of monitoring devices. This is an unattainable task for data analysts, whose expertise falls beyond that required to express complex queries joining multiple data sources and spanning over all schema versions. A desiderata is

[4]`https://www.supersede.eu`

to have a high-level representation of the domain of interest, agnostic of such technical details, so that data analysts can pose queries over it.

Figure 4.1 depicts the conceptual representation for SUPERSEDE. We use a UML class diagram as a representative modeling language, however any other would also suffice (e.g., entity/relationship). It supports multiple instances of *SoftwareApplication* (or just app), but for the sake of simplicity, we narrow the scope to a video broadcasting service for olympic games. There exist multiple device-specific implementations of the app, thus on execution different kinds of data collectors continuously obtain quality metrics. These can be categorized into *Monitors* or *FeedbackGathering*, respectively obtaining runtime data (i.e., *InfoMonitor*), and user feedback (i.e., *UserFeedback*). Such events are associated with their generation time (at the level of *Hour*).

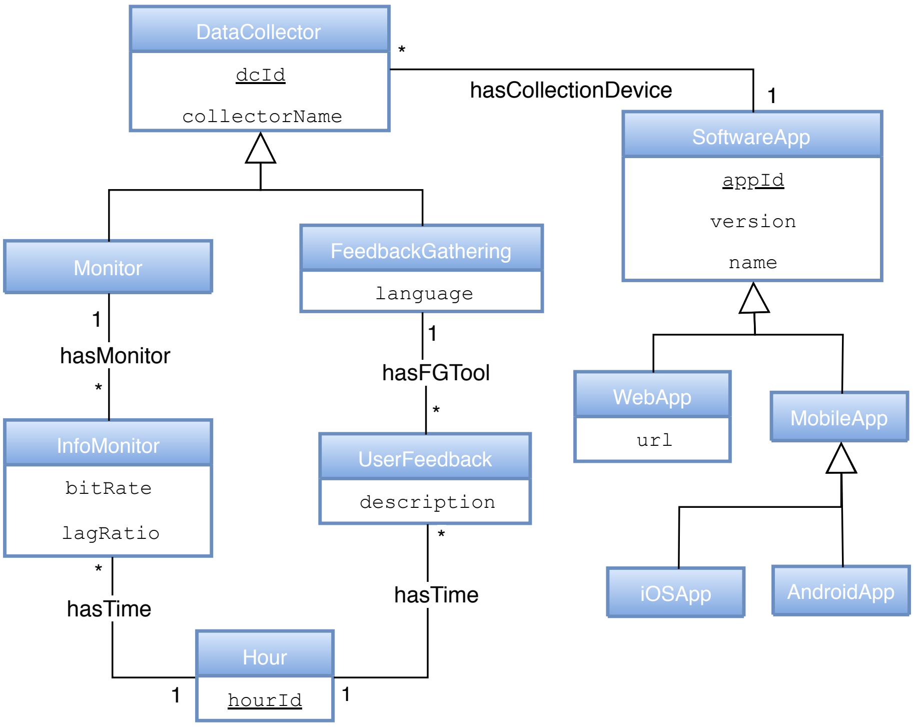


Fig. 4.1: Conceptual model for SUPERSEDE

Next, Figure 4.2 depicts an excerpt of the data sources, here represented as an homogeneous set of JSON documents to ease readability (albeit this is far from a realistic scenario). Precisely, from left to right, we encounter: metadata about the registered applications (e.g., the competition being broadcasted), metadata about the registered monitors (e.g., geographical area they cover), and the temporal aggregation hierarchy. Note that JSON keys do

```
{
"identifier": 4,
"name": "2018
     Winter
     Olympics",
"version": 2.3
}
```

```
{
"monitor":
     63,
"name": "
     Europe
      CDN"
"app": 4
}
```

```
{
"sId": "2018-02-09 07:
     34:25",
"mId": "2018-02-09 07:
     34",
"hId": "2018-02-09 07h
     "
}
```

Fig. 4.2: Sample metadata for *SoftwareApplications*, *Monitor* and *Hour*

```
{
"idMonitor": 44,
"bitRate": 15,
"watchTime": 2,
"waitTime": 4,
"hour": "2018-02-
     09 12h"
}
```

```
{
"idMonitor": 63
     ,
"bitRate": 18,
"lagRatio": 64,
"minute": "2018
     -02-09 10:2
     4"
}
```

```
{
"idMonitor": 28,
"bitRate": 12,
"lagRatio": 88,
"second": "2018-0
     2-09 09:33:2
     1"
}
```

Fig. 4.3: *InfoMonitor* data at different time granularity levels.

not necessarily conform the names in the conceptual model. We will later show this is acceptable in our approach. Then, Figure 4.3 depicts a fragment of the event data generated by monitors (i.e., *InfoMonitor*). Here, different monitors provide data at different time granularity levels, as well as different measurements (e.g, the lag ratio is measured as the fraction of wait and watch time). Also, let us assume the first two correspond to Android devices, while the latter to iOS.

Now, given this setting, the goal is to allow data analysts pose queries over a **graph-based representation** of the conceptual model in Figure 4.1. These should be automatically rewritten to equivalent queries over the sources (e.g., Figures 4.2 and 4.3). Throughout the chapter, we will exemplify our approach with the following query: *"app name and average hourly monitored lag ratio for Android-based apps"*.

3.2 Formal background

Throughout this subsection, we formalize the components that build up our approach. Note that, for the sake of consistency, we present again some of the concepts previously introduced in Chapter 3. Here, however, we put the focus on the used graph structure and components with the goal of performing query rewriting.

Data source model and queries

Relations and wrappers. A schema R is a finite nonempty set of relational symbols $\{r_1, \ldots, r_m\}$, where each r_i has a fixed arity n_i. Let A be a set of attribute names, then each $r_i \in R$ is associated to a tuple of attributes denoted by $att(r_i)$. Henceforth, we will assume that $\forall i, j : i \neq j \rightarrow att(r_i) \cap att(r_j) = \emptyset$ (i.e., relations do not share attribute names), which can be simply done prefixing attribute names with their relation name. Let D be a set of values, a tuple t in r_i is a function $t : att(r_i) \rightarrow D$. For any relation r_i, $tuples(r_i)$ denotes the set of all possible tuples for r_i. A wrapper w is an element in R with a function `exec()` that returns a set of tuples $T \subseteq tuples(w)$. The mechanism underlying `exec()` is transparent to our approach, only requiring its output to be a set of relational tuples. In practice, wrappers can be implemented via SQL queries, *Apache Spark* jobs or remote web service invocations, as long as there exists a mapping function from the specific data model to *first normal form* (1NF).

Conjunctive and conjunctive aggregate queries. A conjunctive query (CQ) is an expression of the form

$$Q = \pi_{\overline{y}}(w_1(\overline{x_1}) \times \ldots \times w_n(\overline{x_n}) \mid \bigwedge_{i=1}^{m} P_i(\overline{z_i}))$$

where $w_1, \ldots, w_n$ are distinct wrappers; $\overline{x_1}, \ldots, \overline{x_n}$ are sets of attributes such that $\overline{x_i} = att(w_i)$; $P_1, \ldots, P_n$ are equi join predicates respectively over $\overline{z_1}, \ldots \overline{z_n}$; and both $\bigcup_{i=1}^{m} \overline{z_i}$ and $\overline{y}$ are subsets of $\bigcup_{i=1}^{n} \overline{x_i}$. Here, $\overline{y}$ denotes output (or projected) attributes. Note that we have dropped predicate filters out of the definition of Q. Throughout the chapter, we might refer to a CQ as a 3-tuple $Q = \langle \pi, \bowtie, W \rangle$ respectively denoting the sets of projected attributes, join predicates and wrappers of Q. We also define the functions $att(Q)$, $wrap(Q)$ and $predatt(Q)$ respectively denoting the sets of projected attributes π, wrappers W and attributes contained in the equi join predicates $\bowtie$ of Q. We define the composition of two CQs ($Q = Q_1 \oplus Q_2$) as $Q = \langle att(Q_1) \cup att(Q_2), predatt(Q_1) \cup predatt(Q_2), wrap(Q_1) \cup wrap(Q_2) \rangle$.

Note the presented syntax of CQs does not include filters (e.g., $w_1.age > 30$). Without loss of generality, it is always possible to push down unary selection predicates on top of every wrapper.

A union of conjunctive queries (UCQ) is an expression of the form

$$\overline{Q} = Q_1 \cup \ldots \cup Q_n$$

where $Q_1, \ldots, Q_n$ are union-compatible CQs. Two CQs Q_1 and Q_2 are union-compatible if they have the same number of attributes (i.e., $|att(Q_1)| = |att(Q_2)|$). From now on, we will interpret a set of CQs as a UCQ.

A conjunctive aggregate query (CAQ) is an expression of the form

$$Q' = (\overline{x}, \overline{\alpha})\ Q$$

where Q is a CQ (or a UCQ); $\overline{x} \subseteq att(Q)$ is the *group-by* set; and each $\alpha \in \overline{\alpha}$ is an aggregate function defined over some attribute $a \in att(Q)$ for $tuples(Q)$. We require, $\overline{x}$ and $att(\overline{\alpha})$ to be disjoint (i.e., $\overline{x} \cap att(\overline{\alpha}) = \emptyset$), and its union to cover all attributes from Q (i.e., $\overline{x} \cup att(\overline{\alpha}) = att(Q)$).

Integration graph

In this thesis we adopt Lenzerini's data integration framework [98] into our graph-based integration setting. Thus, an integration graph $\mathcal{I}$ is formalized as a 3-tuple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, consisting of, respectively the global, source and mappings graphs. Next, we individually present each of these components.

Global graph. The global graph $\mathcal{G} = \langle V_{\mathcal{G}}, E_{\mathcal{G}} \rangle$ is an unweighted, directed, connected graph with no self loops. The vertex set $V_{\mathcal{G}}$ is partitioned into two disjoint sets C and F. We call the elements of C *concepts*, and the elements of F *features*. The set F itself is further partitioned into two disjoint subsets F_{id} and F_{id}^{-}, consisting of *id* features and *non-id* features, respectively. Next, labels in $E_{\mathcal{G}}$ contain the analyst's domain $\mathcal{L}$ as well as the set of *semantic annotations* $\mathcal{A}$. Semantic annotations are system specific labels and have a special treatment, for instance to drive the query rewriting process. Note that $\mathcal{A}$ and $\mathcal{L}$ must be disjoint. For now, we focus on the semantic annotations `hasFeature`, used to relate concepts and their features; and `subClass`, which allow to represent specialization (IS-A) relationships. Throughout the chapter, we will introduce further semantic annotations as required. Hence, we formalize the edge set $E_{\mathcal{G}}$ as the union of the following sets:

- $(C \times \mathcal{L} \times C)$, assigning labels in $\mathcal{L}$ between concepts;
- $(C \times \{\texttt{hasFeature}\} \times F)$, linking concepts and their features;
- $(C \times \{\texttt{subClass}\} \times C)$, creating inheritance relationships between concepts.

In the spirit of non-composite primary keys, we require concepts to have at most one ID feature. Moreover every ID feature can be linked to at most one concept.

Source graph. The definition of the source graph $\mathcal{S}$ is analogous to that of $\mathcal{G}$. However, here the vertex set $V_{\mathcal{S}}$ is composed of $(W \cup A)$, respectively the set of wrappers and attributes from the previous definition (recall that $\mathcal{S}$ is a graph-based representation of the wrappers and their attributes). We use

$wrap(\mathcal{S})$ to denote the set of wrappers in $V_{\mathcal{S}}$. Here, we introduce the semantic annotation `hasAttribute`, meant to connect a wrapper with its attributes. Thus, in $\mathcal{S}$ the edge set $E_{\mathcal{S}}$ is composed of $(W \times \{\texttt{hasAttribute}\} \times A)$. Note that, under the 1NF assumption, $\mathcal{S}$ can be seen as a set of stars. The semantics of a query over the source graph $Q_{\mathcal{S}}$ is that previously introduced of CQs over the wrappers.

Edge-restricted patterns. An *edge-restricted pattern* is a triple $p = \langle s, \ell, t \rangle$, where s and t are constants in $V_{\mathcal{G}}$; and $\ell \in (\mathcal{L} \cup \mathcal{A})$ is an edge label. From now on we are going to consider sets of edge-restricted patterns as graphs. Furthermore, hereinafter we will assume that the considered sets of edge-restricted patterns are *connected*.

Schema mappings. A LAV schema mapping for a wrapper w is a pair $\mathcal{M}(w) = \langle \mathcal{F}, \varphi \rangle$, where $\mathcal{F}$ is an injective function $\mathcal{F} : att(w) \rightarrow F$; and φ is a set of edge-restricted patterns. Consequently, we define the functions $map(\mathcal{M}(w))$ and $patt(\mathcal{M}(w))$ respectively denoting, for $\mathcal{M}(w)$, the mapping from attributes to features $\mathcal{F}$ and the set of edge-restricted patterns φ. Recall that we encode mappings as part of the graph, precisely $\mathcal{M}$ contains $\mathcal{F}$ and φ. Thus, to encode $\mathcal{F}$ we extend the set of semantic annotations $\mathcal{A}$ with the `sameAs` label, linking attributes in $\mathcal{S}$ to features in $\mathcal{G}$. For φ, we encode it as a subgraph of $\mathcal{G}$ (i.e., a *named graph*), which intuitively identifies the fragment of $\mathcal{G}$ covered by w. From now on, we will assume consistency between $\mathcal{F}$ and φ, in the sense that all features mapped by $\mathcal{F}$ are mentioned in t for some edge-restricted pattern in φ.

Before moving on with the formalization, let us reflect on the relationship between the proposed mappings and the customary form found in literature. Commonly, schema mappings are represented by *source-to-target tuple generating dependencies* (*s-t tgds*) [46]. These are logical expressions of the form $\forall \overline{x}(\phi_S(\overline{x}) \rightarrow \exists \overline{y} \psi_T(\overline{x}, \overline{y}))$, where $\phi_S(\overline{x})$ and $\psi_T(\overline{x}, \overline{y})$ are respectively conjunctions of first-order formulas over the source and target schemas. LAV mappings are a special case of them, with the form $\forall \overline{x}(R_S(\overline{x}) \rightarrow \exists y \psi_T(\overline{x}, \overline{y}))$, where $R_S(\overline{x})$ is a source relational symbol. Note there exists a direct relation between the graph-based and this logical form for LAV mappings. Specifically, for each $\mathcal{M}(w)$ we would have a formula of the form $\forall \overline{x_{\mathcal{S}}}(w(\overline{x_{\mathcal{S}}}) \rightarrow \exists \overline{y_{\mathcal{G}}}\ \varphi(\overline{x_{\mathcal{G}}}, \overline{y_{\mathcal{G}}}))$, where $\overline{x_{\mathcal{S}}} \subseteq att(w)$; $\overline{x_{\mathcal{G}}}, \overline{y_{\mathcal{G}}} \subseteq F$, and φ is a connected set of edge-restricted patterns. Unlike traditional mapping definitions, the set of variables $\overline{x_{\mathcal{S}}}$ does not appear in the body. This is aligned with the fact that attribute names (externally defined in the wrappers) might differ from feature names (defined in the global graph). Such link is made by the injective function $\mathcal{F}$.

Querying the sources via the integration graph

Lastly, we now formalize the constructs used to query the data sources via the integration graph. Thus, from now on we assume all operations are applied over a fixed instance of $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$. In order not to overload notation, we might independently refer to the components of $\mathcal{G}$, $\mathcal{S}$ or $\mathcal{M}$ avoiding mentioning $\mathcal{I}$.

Global queries. A global query $Q_{\mathcal{G}}$ is a pair of the form $\langle \pi, \varphi \rangle$ where $\pi \subseteq F$ is a set of projected features; and $\varphi = \{p_1, \ldots, p_n\}$ is a set of edge-restricted patterns. We hence define the functions $proj(Q_{\mathcal{G}})$, returning the set of projected features, and $patt(Q_{\mathcal{G}})$, returning the set of edge-restricted patterns. $Q_{\mathcal{G}}$ is *legal* if composing the elements of φ yields a graph that is acyclic and connected if we disregard edge directions. We do not further specify the semantics on how to evaluate $Q_{\mathcal{G}}$, as queries posed on such language are only meant to be rewritten, a process we later describe and will fix its semantics.

Definition 1 (Covering$_{\mathcal{I}}(W, \varphi)$)
A set of wrappers $W \subset wrap(\mathcal{S})$ covers the set of edge-restricted patterns $\varphi \subseteq \mathcal{G}$, denoted $\textsc{covering}_{\mathcal{I}}(W, \varphi)$, if the union of LAV mappings of wrappers in W subsumes φ. This is formally defined as $\forall w \in W : \bigcup patt(\mathcal{M}(w)) \supseteq \varphi$.

Definition 2 (Minimal$_{\mathcal{I}}(W, \varphi)$)
A set of wrappers $W \subset \mathcal{S}$ is minimal w.r.t. the set of edge-restricted patterns $\varphi \subseteq \mathcal{G}$ if removing any wrapper in W yields a non-covering set of wrappers. This is formally defined as $\nexists w \in W : \textsc{covering}_{\mathcal{I}}(W \backslash w, \varphi)$.

Rewritings. A CQ $Q_{\mathcal{S}}$ is a *rewriting* of a global query $Q_{\mathcal{G}}$ if the wrappers in the rewriting cover the set of edge-restricted patterns, formally $\textsc{covering}_{\mathcal{I}}(wrap(Q_{\mathcal{S}}), patt(Q_{\mathcal{G}}))$; and attributes participating in equi join predicates in the rewriting use only ID features, formally $\forall p \in pred(Q_{\mathcal{S}}) \exists w \in wrap(Q_{\mathcal{S}}) : map(\mathcal{M}(w))(p) \in F_{id}$.

Proposition 1
Given two disjoint sets of edge-restricted patterns φ, φ', and two distinct rewritings $Q_{\mathcal{S}}$, $Q'_{\mathcal{S}}$. If $\textsc{minimal}_{\mathcal{I}}(wrap(Q_{\mathcal{S}}), \varphi)$ and $\textsc{minimal}_{\mathcal{I}}(wrap(Q'_{\mathcal{S}}), \varphi')$, then $\textsc{minimal}_{\mathcal{I}}(wrap(Q_{\mathcal{S}} \oplus Q'_{\mathcal{S}}), \varphi \cup \varphi')$ if $wrap(Q_{\mathcal{S}}) \cap wrap(Q'_{\mathcal{S}}) = \emptyset$.

Proof. The proof can be straightforwardly obtained from Definition 2. $\square$

Rewriting algorithm. A rewriting algorithm is a function $\mathcal{R}_{\mathcal{I}} : \mathbb{Q}_{\mathcal{G}} \to \overline{\mathbb{Q}_{\mathcal{S}}}$ from the set $\mathbb{Q}_{\mathcal{G}}$ of all legal global queries to the set $\overline{\mathbb{Q}_{\mathcal{S}}}$ of UCQs, such that $\forall Q_{\mathcal{G}} \in \mathbb{Q}_{\mathcal{G}}$, $\mathcal{R}_{\mathcal{I}}(Q_{\mathcal{G}})$ consists only of rewritings of $Q_{\mathcal{G}}$. We define the notions of *minimally-sound* and *minimally-complete* rewriting algorithms. Informally, the former depicting that all rewritings provided by $\mathcal{R}$ are minimal, and the latter depicting that $\mathcal{R}$ yields all possible minimal rewritings. The formal definitions are as follows:

Definition 3 (Minimally-sound($\mathcal{R}_\mathcal{I}$))
A rewriting algorithm $\mathcal{R}_\mathcal{I}$ is minimally-sound if $\forall Q_\mathcal{G} \in \mathbb{Q}_\mathcal{G}$ and $\forall Q_\mathcal{S} \in \mathcal{R}_\mathcal{I}(Q_\mathcal{G})$ we have $\textsc{minimal}_\mathcal{I}(wrap(Q_\mathcal{S}), patt(Q_\mathcal{G}))$.

Definition 4 (Minimally-complete($\mathcal{R}_\mathcal{I}$))
A rewriting algorithm $\mathcal{R}_\mathcal{I}$ is minimally-complete if $\forall_{Q_\mathcal{G} \in \mathbb{Q}_\mathcal{G}}$ and every $Q_\mathcal{S}$ such that it is a rewriting of $Q_\mathcal{G}$ and satisfies $\textsc{minimal}_\mathcal{I}(wrap(Q_\mathcal{S}), patt(Q_\mathcal{G}))$, it holds that $Q_\mathcal{S} \in \mathcal{R}_\mathcal{I}(Q_\mathcal{G})$.

Problem statement We recall that in this chapter we are interested in studying the problem of answering queries $Q_\mathcal{G}$ posed over an integration graph $\mathcal{I}$. Hence, this reduces to finding a rewriting algorithm $\mathcal{R}_\mathcal{I} : Q_\mathcal{G} \rightarrow \overline{Q_\mathcal{S}}$ where it holds that $\textsc{minimally-sound}(\mathcal{R}_\mathcal{I})$ and $\textsc{minimally-complete}(\mathcal{R}_\mathcal{I})$.

3.3 Case study (cont.)

Going back to the case study presented in Section 3.1, we now exemplify it using the introduced formalization. Figure 4.4 depicts the graphical representation of the global graph $\mathcal{G}$.

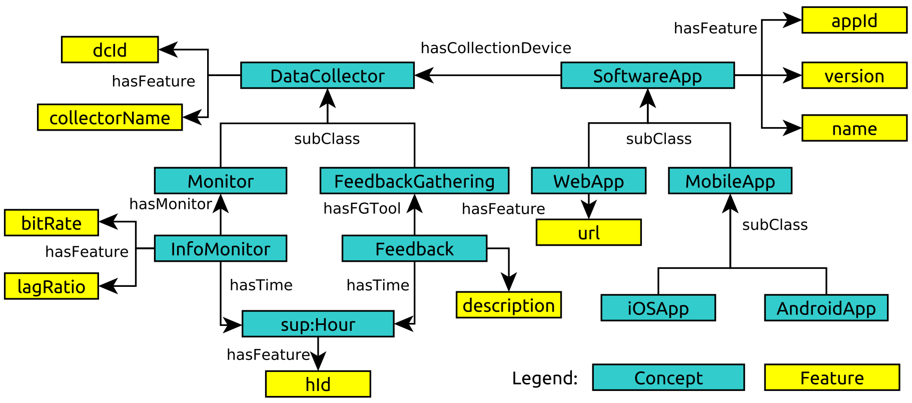


Fig. 4.4: Global graph for SUPERSEDE

Then, let us assume $\mathcal{S}$ contains a set of wrappers covering the previously introduced data sources. Precisely, we have $w_{apps}(idApp, name, version)$, $w_{mon}(idMon, nameMon, app)$ and $w_{time}(sID, mID, hID)$, respectively providing data for apps, monitors and time as depicted in Figure 4.2. Next, we have the wrappers for event data from Figure 4.3. We define $w_1(idMonitor, bitRate, lagRatio, time)$ as the wrapper querying the leftmost JSON document (i.e., info monitor for Android devices at the hour level), which does not provide the lag ratio metric but the watch and wait times. Note the wrapper's schema and that of the JSON document differ, which is caused by the wrapper's implementation (i.e., the `exec()` function) as defined in Code 4.1. Similarly, we define w_2 and w_3 with the same schema, respectively querying info monitor for Android devices at minute level and for iOS devices at second level.

```
sparkContext.read.json("hdfs://...")
  .withColumn("lagRatio",col("watchTime")/col("waitTime"))
  .withColumnRenamed("hour","time")
  .select("idMonitor","bitRate","lagRatio","time")
  .collect
```

Code 4.1: Spark-based implementation of w_1 in Scala

Next, the global and source graphs are linked via LAV mappings. Figure 4.5 depicts a graphical representation of the complete integration system $\mathcal{I}$, as well as the LAV mappings of wrappers w_{apps}, w_{mon}, w_{time} and w_1 (note we do not include those that involve aggregation techniques, precisely w_2 and w_3, which will be introduced later in the chapter).

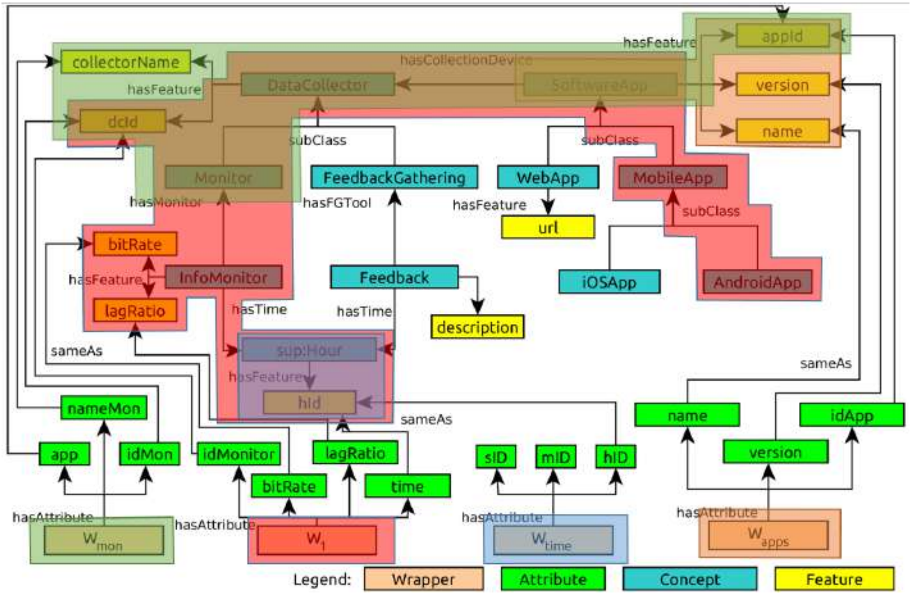


Fig. 4.5: Integration system for SUPERSEDE. Each wrapper is colored as its corresponding LAV mapping

Finally, recall the query *"app name and average hourly monitored lag ratio for Android-based apps"*, whose global query $Q_{\mathcal{G}}$ version corresponds to the following expression.

- $\pi = \{name, hId, lagRatio\}$

- $\varphi = \{\langle Hour, hasFeature, hId\rangle \wedge \langle InfoMonitor, hasTime, Hour\rangle \wedge \langle InfoMonitor, hasFeature, lagRatio\rangle \wedge \langle InfoMonitor, hasMonitor, Monitor\rangle \wedge \langle Monitor, subClass, DataCollector\rangle \wedge \langle DataCollector, hasApp, SoftwareApp\rangle \wedge \langle SoftwareApp, hasFeature, name\rangle \wedge \langle MobileApp, subClass, SoftwareApp\rangle \wedge \langle AndroidApp, subClass, MobileApp\rangle\}$

A legal rewriting for it would be the following expression (we omit the wrapper's sets of attributes for brevity in this example). Note we have not included w_2 in the rewriting, as its attribute *time* is at the minute level and should be aggregated. The process involving implicit aggregation and CAQ generation is described in Section 5.

$$\pi_{w_{apps}.name, w_{time}.hId, w_1.lagRatio}(w_1 \times w_{mon} \times w_{apps} \times w_{time})|$$
$$w_1.idMonitor = w_{mon}.idMon \wedge w_{mon}.app = w_{apps}.idApp \wedge w_1.time = w_{time}.hId)$$

4 Rewriting Conjunctive Queries

The core of our method is RewriteCQ, a rewriting algorithm that given a query $Q_{\mathcal{G}} = \langle \pi, \varphi \rangle$ automatically resolves the LAV mappings and discovers how to join wrappers to yield a UCQ $\overline{Q_{\mathcal{S}}}$. RewriteCQ is inspired by the *bucket algorithm* for LAV mediation [100]. As presented, the main idea of the bucket algorithm is first to individually find rewritings for each subgoal in the query, and store them in buckets. Then, the algorithm finds a set of conjunctive queries such that each of them contains one conjunct from every bucket. In our case, concepts are analogous to buckets. Hence, we will first separately find those wrappers that cover the requested concepts in φ (i.e., *intra-concept generation*) to later find all ways to join their combinations that yield covering and minimal rewritings (i.e., *inter-concept generation*).

An additional feature of our approach is that we leverage specialization relationships to select wrappers. Precisely, if the set of edge-restricted patterns φ from $Q_{\mathcal{G}}$ includes the set of concepts $c_1, \ldots, c_n$, then we should only consider in the rewriting process those wrappers w such that $c_i \in patt(\mathcal{M}(w))$ (for example, in the exemplary query w_3 should not be considered as its LAV mapping does not cover *AndroidApp* but *iOSApp*). This feature allows to reduce the set of candidate wrappers in the resulting $Q_{\mathcal{S}}$.

4.1 Preliminaries

The proposed rewriting algorithm extensively performs subgraph matching tasks, here implemented via *conjunctive regular path queries* (CRPQs) applied over the global and source graphs. A CRPQ retrieves sequences of nodes and edges in the graph such that these nodes and edges are connected in the graph by a path conforming to certain regular expressions.

For the sake of simplicity, here, we will distinguish when a CRPQ yields a unique value to a variable (i.e., denoting the variable as z_i) or a set of values (i.e., denoting the variable as $\overline{z_i}$).

Example 4.1
The following query, $\overline{z} \leftarrow \langle x, y, ?z \rangle(\mathcal{G})$ matches all those triples with $s = x$ and $\ell = y$ (where x and y are constants) and a variable z from the graph G. Furthermore, denoting the head as $\overline{z}$ we indicate the query yields a set of matches.

Example 4.2
The query, $(\overline{?c, ?f}) \leftarrow \langle ?c, \texttt{hasFeature}, ?f \rangle \wedge \langle ?f, \texttt{subClass}, \texttt{ID} \rangle(G)$ returns pairs of concept and their ID (obtained as a subclass of the constant ID) from $\mathcal{G}$.

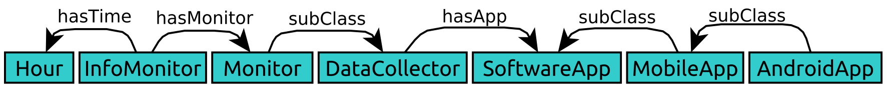
We refer the reader to the literature on queries over graphs for a full formal syntax and semantics of CRPQs [11, 19, 167].

4.2 Rewriting algorithm

In this and the following subsections we present REWRITECQ at a high abstraction level. The specific details of the algorithms, together with their implementation based on CRPQs, can be found in Appendix A.

Algorithm 2 depicts REWRITECQ. First, we construct the graph of query related concepts G as an analogy to empty buckets. Precisely, the graph G consists only of concept vertices (i.e., from C) that φ refers to and their relationships.

Example 4.3
In the exemplary query (i.e., *"app name and average hourly monitored lag ratio for Android-based apps"*), the graph of query related concepts G would be that depicted in the following figure.



Algorithm 2 RewriteCQ

```
Input: $\mathcal{I}$ is an integration graph, $Q_{\mathcal{G}} = \langle \pi, \varphi \rangle$ is a global query
Output: $\overline{Q_{\mathcal{S}}}$ is a UCQ
 1: function REWRITECQ($\mathcal{I}, Q_{\mathcal{G}}$)
 2:   let $G \leftarrow \emptyset$ be the graph of query related concepts
 3:   for each triple $p \in \varphi$ do
 4:     if $p$ connects two concepts then
 5:       $G \cup= p$
 6:   $partialCQsGraph \leftarrow$ INTRACONCEPTGENERATION($Q_{\mathcal{G}}, G$)
 7:   $\overline{Q_{\mathcal{S}}} \leftarrow$ INTERCONCEPTGENERATION($partialCQsGraph$)
 8:   return $\overline{Q_{\mathcal{S}}}$
```

4.3 Intra-concept generation

This phase (see Algorithm 3) receives as input a global query $Q_{\mathcal{G}}$ and the graph of query related concepts to generate a graph of *partial rewritings* (or *partial CQs*) per concept. Partial rewritings cover a specific concept and its features that have been stated in φ. These will be obtained resolving the LAV mappings for each concept to find which wrappers provide their features. Thus, for each concept c, as a first step, we identify all those wrappers that cover some of the queried features (i.e., the set of candidate CQs). We also select those wrappers that cover featureless concepts in the query. Next, we systematically generate combinations of such queries such that they cover c and all its queried features.

Algorithm 3 Intra-concept generation

```
Input: $Q_{\mathcal{G}} = \langle \pi, \varphi \rangle$ is a global query, $G$ is the graph of query related concepts
Output: $partialCQsGraph$ is the graph of partial CQs per concept
 1: function INTRACONCEPTGENERATION($\langle \pi, \varphi \rangle, G$)
 2:   let $partialCQsGraph$ be an empty graph where vertices are pairs <Concept,$\overline{CQ}$>
 3:   for each concept $c$ in the graph of query related concepts $G$ do
 4:     let $attsPerWrapper$ be a map structure where keys are wrappers ($W$) and values sets of
        covered attributes ($\overline{A}$)
 5:     let $\overline{F}$ be the set of queried features for concept $c$
 6:     if no features in $c$ have been queried ($\overline{F} = \emptyset$) then
 7:       add $\emptyset$ to $attsPerWrapper$ for each wrapper covering the concept $c$
 8:     for $f \in \overline{F}$ do
 9:       let $\overline{W'}$ be the set of wrappers covering the feature $f$ of $c$
10:       for $w \in \overline{W'}$ do
11:         add to $w$ in $attsPerWrapper$ the attribute $a$ corresponding to $f$ covered by $w$
12:     let $candidateCQs$, be a set of queries generated from each pair $\langle w, \overline{A} \rangle$ in $attsPerWrapper$
13:     let $coveringCQs \leftarrow \emptyset$, be the set of CQs fully covering the queried features for $c$
14:     while $candidateCQs \neq \emptyset$ do
15:       let $Q$ be a query removed from the set of candidates
16:       let $I$ be the graph induced by $c$ and its queried features $\overline{F}$
17:       $coveringCQs \cup=$ COVERINGCQs($I, c, Q, candidateCQs$)
```

```
18:     add vertex $c$ to $partialCQsGraph$ with the set of covering queries $coveringCQs$
19:   add edges to $partialCQsGraph$ preserving the connectivity in $G$
20:   **return** $partialCQsGraph$
```

Example 4.4
The output of Algorithm 3 in the example would be a graph (for the sake of simplicity here we show the vertex set) with the following pairs $\langle c, \overline{CQ} \rangle$. Note that, due to the succinctness of the running example, no combinations of CQ have been generated here. For the sake of simplicity we omit set notation when sets contain only one element.

- $Hour$ – $\langle hID, \emptyset, w_{time} \rangle, \langle time, \emptyset, w_1 \rangle$
- $InfoMonitor$ – $\langle lagRatio, \emptyset, w_1 \rangle$, $\langle lagRatio, \emptyset, w_2 \rangle$, $\langle lagRatio, \emptyset, w_3 \rangle$
- $Monitor$ – $\langle \emptyset, \emptyset, w_{mon} \rangle$, $\langle \emptyset, \emptyset, w_1 \rangle$, $\langle \emptyset, \emptyset, w_2 \rangle$, $\langle \emptyset, \emptyset, w_3 \rangle$
- $DataCollector$ – $\langle idMon, \emptyset, w_{mon} \rangle$, $\langle idMonitor, \emptyset, w_1 \rangle$, $\langle idMonitor, \emptyset, w_2 \rangle$, $\langle idMonitor, \emptyset, w_3 \rangle$
- $SoftwareApp$ – $\langle \{idApp, name\}, \emptyset, w_{apps} \rangle$
- $MobileApp$ – $\langle \emptyset, \emptyset, w_1 \rangle$, $\langle \emptyset, \emptyset, w_2 \rangle$, $\langle \emptyset, \emptyset, w_3 \rangle$
- $AndroidApp$ – $\langle \emptyset, \emptyset, w_1 \rangle$

Generating covering CQs

The process of generating covering CQs (see Algorithm 4) is a recursive task that given an input query Q and a set of candidate CQs incrementally generates covering combinations. Ultimately, each of this generated combinations must cover the graph G. Here, G represents the graph induced by the concept c and its queried features. Note that we do not move on with the process if adding a query does not contribute with new features, which ensures *minimality*. Generating the combination of two CQs might entail discovering join conditions among them, this process is depicted in the inter-concept generation (see method COMBINECQ in Algorithm 6).

Algorithm 4 Get covering CQs

```
**Input:** $G$ is the graph to check coverage, $c$ is the concept at hand, $currentCQ$ is a CQ, $candidateCQs$
    is a set of CQs
**Output:** the set $candidateCQs$ is empty, all potential combinations of covering CQs with respect
    to $G$ are in $coveringCQs$
 1: **function** COVERINGCQS($G$, $c$, $currentCQ$, $candidateCQs$)
 2:   let $coveringCQs \leftarrow \emptyset$ be the set of generated covering CQs
 3:   if $currentCQ$ covers $G$ then
 4:     add $currentCQ$ to the set $coveringCQs$
 5:   else if $candidateCQs \neq \emptyset$ then
 6:     for $CQ \in candidateCQs$ do
 7:       if $currentCQ \oplus CQ$ provides more features than $currentCQ$ itself then
 8:         let $Q'$ be the query resulting from calling combineCQ($currentCQ$, $CQ$, $c$, $c$)
 9:         recursively call coveringCQs using $Q'$ as $currentCQ$ and removing $CQ$ from $candidateCQs$
10:   return $coveringCQs$
```

Example 4.5
Additionally to the wrappers presented in the case study, let us assume two new wrappers $w'_{apps}(idApp, name)$ and $w''_{apps}(idApp, version)$ covering *SoftwareApp* and the respective features. Thus, when processing the concept *SoftwareApp* (for the sake of this example, let us assume *version* is also covered by φ), Algorithm 4 would generate the following two covering CQs:

- $\langle\{idApp, name, version\}, \emptyset, w_{apps}\rangle$
- $\langle\{w'_{apps}.idApp, w'_{apps}.name, w''_{apps}version\}, \{w'_{apps}.idApp = w''_{apps}.idApp\}, \{w'_{apps}, w''_{apps}\}\rangle$

4.4 Inter-concept generation

This phase deals with the combination of queries covering connected concepts. It receives as input the *partialCQsGraph* and systematically compacts edges from the graph generating new sets of minimal CQs. At each iteration, we generate a new synthetic node as a result of compacting the source and target nodes of the selected edge, thus the algorithm terminates when the graph has no edges. Note method chooseEdge, which can range from a purely random selection to an informed heuristic based decision that prioritizes early pruning. In our implementation, chooseEdge is based on the least number of wrappers on both ends. Defining complex heuristics is out of the scope of this chapter. Next, we discuss the high level specification of the algorithm.

Algorithm 5 Inter-concept generation

```
Input: $partialCQsGraph$ is the graph of partial CQs per concept
Output: $UCQs$ is a set of CQs (i.e., a union of CQs)
1: function InterConceptGeneration($partialCQsGraph$)
2:   while $partialCQsGraph$ has edges do
3:     $e \leftarrow$ chooseEdge($partialCQsGraph$)
4:     let $s$ and $t$ be the source and target concepts of $e$, and $\overline{CQ_s}$ and $\overline{CQ_t}$ be the sets of CQs
       respectively covering $s$ and $t$
5:     let $\overline{CQ}$ be the resulting set of calling combineCQ($\overline{CQ_s}$, $\overline{CQ_t}$, $s$, $t$)
6:     Remove $s$, $t$ from $partialCQsGraph$, add a new vertex $s + t$ with $\overline{CQ}$ preserving connectivity.
7:   return aVertexFrom($partialCQsGraph$)          ▷ $partcialCQsGraph$ has a single vertex
```

Example 4.6
Using the input from the previous phase (see Example 4.4, the output from Algorithm 5 would be a set containing the following expression:

$$\pi_{w_{apps}.name, w_{time}.hId, w_1.lagRatio}(w_1 \times w_{mon} \times w_{apps} \times w_{time})|$$
$$w_1.idMonitor = w_{mon}.idMon \wedge w_{mon}.app = w_{apps}.idApp \wedge w_1.time = w_{time}.hId)$$

Combining sets of CQs

Method COMBINECQs (see Algorithm 6) deals with the generation of legal combinations of the queries in the sets $\overline{CQ_s}$ and $\overline{CQ_t}$. This method is split in two steps, first combining those CQs that share a wrapper and then combining those that do not share wrappers. Two CQs that share a wrapper can be merged (i.e., using operator $\oplus$) if the resulting query is minimal with no further action. Merging CQs that do not share wrappers involve discovering equi joins among their participating wrappers. To this end, we filter out those queries from $\overline{CQ_s}$ and $\overline{CQ_t}$ that cover the identifier of the participating concepts c_s and c_t (i.e., yielding the four sets of queries $\overline{CQ_{s-ID_s}}$, $\overline{CQ_{s-ID_t}}$, $\overline{CQ_{t-ID_s}}$ and $\overline{CQ_{t-ID_t}}$). Then, by computing the cartesian product of those sets of queries covering the same ID we call method FINDJOINS.

Algorithm 6 Combine sets of CQs

```
Input: $\overline{CQ_s}$ and $\overline{CQ_t}$ are sets of CQs, $c_s$ and $c_t$ are concepts respectively covered by $\overline{CQ_s}$ and $\overline{CQ_t}$,
    $e$ is the edge connecting $c_s$ and $c_t$
Output: $\overline{CQ}$ is a set with all valid combinations of $\overline{CQ_s}$ and $\overline{CQ_t}$
 1: function COMBINECQs($\overline{CQ_s}$, $\overline{CQ_t}$, $c_s$, $c_t$, $e$)
 2:   let $\overline{W_{shared}}$ be the set of wrappers that appear in both $\overline{CQ_s}$ and $\overline{CQ_t}$, and cover edge $e$
 3:   for $w \in \overline{W_{shared}}$ do
 4:     for each pair $q_s$,$q_t$ from the cartesian product $\overline{CQ_s}$ and $\overline{CQ_t}$ that contain $w$ do
 5:       if $q_s \oplus q_t$ is minimal w.r.t. the graph induced by $c_s$, $c_t$ and the edge connecting them then
 6:         add to $\overline{CQ}$ the query $q_s \oplus q_t$
 7:   define $\overline{CQ_{s-ID_s}}$, $\overline{CQ_{s-ID_t}}$, $\overline{CQ_{t-ID_s}}$ and $\overline{CQ_{t-ID_t}}$ as the subsets of respectively $\overline{CQ_s}$ or $\overline{CQ_t}$
      that cover $e$ and the identifiers of $c_s$ or $c_t$, and do not have wrappers in $\overline{W_{shared}}$
 8:   for each pair $q_s, q_t \in \overline{CQ_{s-ID_s}} \times \overline{CQ_{t-ID_s}}$ do
 9:     add to $\overline{CQ}$ the result of FINDJOINS($q_s, q_t, ID_s$)
10:   for each pair $q_s, q_t \in \overline{CQ_{s-ID_t}} \times \overline{CQ_{t-ID_t}}$ do
11:     add to $\overline{CQ}$ the result of FINDJOINS($q_s, q_t, ID_t$)
12:   return $\overline{CQ}$
```

Discovering joins for two CQs.

Given two CQs CQ_s and CQ_t, method FINDJOINS (see Algorithm 7) performs the process of finding equi join predicates among them using the identifier ID. This process finds all wrappers covering ID from CQ_s and CQ_t, to compute their cartesian product. For each combination of wrappers, we look the attribute corresponding to ID and generate a new equi join predicate.

Algorithm 7 Find joins

```
Input: $Q_s$ and $Q_t$ are CQs, $ID$ is an identifier feature
Output: $CQ$ is a combination of $Q_s$ and $Q_t$ with equi join predicates
1: function FINDJOINS($Q_s$, $Q_t$, $ID$)
2:   let $\overline{W}_s$ be the wrappers from $Q_s$ covering $ID$
3:   let $\overline{W}_t$ be the wrappers from $Q_t$ covering $ID$
4:   $CQ \leftarrow Q_s \oplus Q_t$
5:   for each pair $w_s, w_t \in \overline{W}_s \times \overline{W}_t$ do
6:     let $a_s$ be the attribute corresponding to $ID$ in $w_s$
7:     let $a_t$ be the attribute corresponding to $ID$ in $w_t$
8:     add $a_s = a_t$ as new equi join predicate to $CQ$
9:   return $CQ$
```

4.5 Discussion

In this subsection, we discuss the computational complexity of REWRITECQ as well as its soundness and completeness.

Computational complexity

We start this discussion classifying REWRITECQ to its complexity class.

Theorem 1. *Rewriting a query Q_G to a UCQs $\overline{Q_S}$ using* REWRITECQ *is NP-hard.*

Proof. The previous theorem can be easily proved by reduction from *Set Cover* [88]. The optimization/search version of set cover is a well-known NP-hard problem. Shortly, the set cover problem is defined as: given a set S of n points and $\mathcal{F} = \{S_1, S_2, \ldots, S_m\}$ a collection of subsets of S, select as few as possible subsets from $\mathcal{F}$ such that every point in S is contained in at least one of the subsets. The reduction works as follows. From the set of points S, let us consider a global query Q_G, where for each point in S we generate a triple $p_i = \langle s, \ell, t \rangle \in \varphi$ (note graph edges can be disregarded and checked at the end). Then, from the set $\{S_1, \ldots, S_m\}$ we consider the set of all wrappers covering some point in S. It is straightforward to see that finding combinations of subsets is equivalent to finding combinations of wrappers such that the complete set of attributes in the query is covered. Furthermore, the set cover problem seeks as few as possible subsets, which is equivalent to our definition of a minimal rewriting. As a matter of fact, we are interested in enumerating all possible solutions of the problem, while in some instances of set cover finding one is enough. □

Next, after classifying REWRITECQ in the class of NP-hard problems we want to get an accurate cost formula. Let W be the average number of wrappers covering each concept (not including those concepts that are not covered by any wrapper), F be the number of features in a query pattern φ, and C be the number of concepts covered in the query pattern φ. To start with, recall that Algorithm 3 first generates all covering combinations of wrappers per concept. This is achieved incrementally by obtaining all different ways to perform equi joins among them. Next, Algorithm 5 further finds all combinations of queries among different wrappers. From the previous rationale, we can conclude that the complexity of REWRITECQ is $\binom{W}{F}^C$. Its worst case corresponds to the scenario where each wrapper only contributes to one queried feature, and thus all possible combinations are covering. Note that the processing of inheritance relationships does not incur additional cost, oppositely it might prune the set of considered wrappers (and thus the set of covering rewritings).

Minimally-soundness and minimally-completeness

Our aim now is to show that REWRITECQ is a minimally-sound (see Definition 3) and minimally-complete (see Definition 4) rewriting algorithm. Precisely, we discuss the following invariants that hold: *(a)* $\overline{Q_S}$ does not contain any non-minimal CQ, and *(b)* $\overline{Q_S}$ contains all minimal CQs.

Proof. The trivial case occurs when a single concept C is covered by the query pattern φ. Here, only Algorithm 3 will be executed. We can easily see that the set of candidate CQs (line 12) contains all CQs that cover C and some of its queried features. Then, Algorithm 4 systematically combines CQs to later generate combinations of covering CQs (Algorithm 6). As previously explained, this process only generates minimal covering queries (as any combination not contributing with new features is discarded), which guarantees the first invariant. Note also that concepts involved in specialization relationships are also considered by default if they are part of φ. Regarding the second invariant, it is guaranteed by Algorithm 6 which performs a cartesian product when generating combinations of CQs (i.e., finds all possible equi join conditions).

Querying more than one concept involves Algorithm 5. We assume a graph of partial CQs G with vertices $C_1, \dots, C_n$ each with its respective set of minimal CQs. This can be seen as the instantiation of the trivial case for each C_i. Given an edge in G, we systematically generating all possible combinations of CQs from the source and target of it. We show that all minimal CQs are obtained by reductio ad absurdum, thus let us assume the output of Algorithm 5 does not contain a minimal query Q'_S. Recall that generating such combination is performed in two disjoint steps, first processing those queries that share wrappers and then those that do not. For the former, Algorithm 5 makes an explicit check in line 5 to validate that all generated queries are minimal, thus guaranteeing the first invariant. For the latter, the first invariant is guaranteed by Proposition 1, which states that combining two minimal queries that do not share wrappers yields a minimal query. Then, the second invariant is guaranteed by the fact that all generations are computed from cartesian products (i.e., see lines 4, 8 and 10) that cover the complete search space for the sets of queries at hand. Thus, Q'_S has necessarily been generated in one of this three cartesian products (and thus added to the resulting set $\overline{CQ}$), which contradicts the assumption and shows the second invariant is guaranteed. □

5 Rewriting Conjunctive Aggregate Queries

Up to now, we have assumed the global graph $\mathcal{G}$ exclusively contains the analyst's domain of interest. This implicitly determines the granularity at which data should be presented. Our aim now, is to offer the possibility to automatically include any data provided at finer granularity levels into the query results. This requires the definition of a rewriting algorithm performing implicit aggregation operations (i.e., CAQ queries). In this section, we first introduce the required constructs and additional semantic annotations used to drive the rewriting algorithm, and later present its details.

5.1 The aggregation graph

The definition of data structures defining aggregation relationships is a natural task in OLAP. In such settings, a multidimensional lattice represents different aggregations that can be performed from data materialized at lower levels of granularity. Specific data and query models have been proposed to address OLAP on graphs [174]. Here, we aim to adopt such ideas into our graph-based structure to benefit from them in the rewriting process. To this end, we introduce the aggregation graph $\mathcal{G}_{agg}$, defined as a copy of $\mathcal{G}$ where relevant aggregation relationships are materialized (i.e., hierarchies and their levels).

For our purposes, a hierarchy $\mathcal{H}$ is composed of a strict totally ordered set of levels $\mathcal{L}$. We identify the top level of a hierarchy as $top(\mathcal{H})$. We consider concepts in $\mathcal{G}$ as the top levels of their hierarchies. Then, in $\mathcal{G}_{agg}$ we include all those levels such that there exists a wrapper providing data at a lower granularity level. Thus, we define the vertex set of $\mathcal{G}_{agg}$ as $V_{\mathcal{G}_{agg}} = V_{\mathcal{G}} \cup L \cup AF$, where L is the set of *levels* and AF is the set of *aggregation functions*. We assume the set of commutative and associative aggregate functions (i.e., *sum*, *min* and *max*). Next, in the edge set, we extend the set of semantic annotations $\mathcal{A}$, to include the following labels.

Aggregation hierarchies. An edge labeled `partOf` defines a partial order between two levels. Ultimately, the root of a hierarchy must be a concept c in the global graph (i.e., $c \in \mathcal{G}$).

Aggregable features. A `hasAggregationFunction` edge creates a link from features to aggregation functions. A feature is aggregable whenever we can apply an aggregation function to modify its granularity level. It is important to define only those functions that are semantically correct for each feature. For instance, we could link *lagRatio* with *sum*, but it would be meaningless to link such function with *description*. We also assume ID features are not aggregable.

$\mathcal{G}_{agg}$ is constructed in a way that levels preserve the same connectivity properties as their top concept c has in $\mathcal{G}$. Formally, given a triple $p = \langle s, \ell, t \rangle \in \mathcal{G}$ such that s and t are concepts, and s has features where there exist wrappers providing them at granularity levels $\{l_1, \ldots, l_n\}$, then for each level l_i the graph $\mathcal{G}_{agg}$ contains the following triples:

- $p_1 = \langle x, \texttt{partOf}, y \rangle$; where x is a fresh level name, and y is another fresh level name or s itself.
- $p_2 = \langle l_i, \texttt{partOf}, l_j \rangle$, where l_j is another level in the aggregation hierarchy such that $l_i > l_j$, or $l_j = t$.
- $p_3 = \langle x, \ell, l_i \rangle$, which links the freshly defined level name x with l_i using the same edge label ℓ that originally connects s and t in $\mathcal{G}$.

Regarding wrappers and LAV mappings, we expect them to be defined over $\mathcal{G}_{agg}$ covering the levels in the graph for which they provide data. We assume the same mapping mechanism as described in Section 3. The definition of such lattice structure, as well as mapping creation, is a necessary task that needs to be done by the data steward (i.e., administrator), similar to the situation in data warehouses.

Example 5.1
Figure 4.6 depicts the fragment of $\mathcal{G}_{agg}$ for SUPERSEDE. For the sake of space, we only show the part corresponding to concepts *InfoMonitor* and *Hour*. Note the graph now contains the time aggregation hierarchy, including *Second* and *Minute*, where the sets of triples p_1, p_2, p_3 have been materialized for each of the levels. This entailed the definition of *InfoMonitorSec* and *InfoMonitorMin*.

Recall the set of wrappers introduced in Section 3.3. Figure 4.7 depicts the subgraphs defined by $patt(\mathcal{M}(w))$ for wrappers w_{time}, w_1, w_2 and w_3.

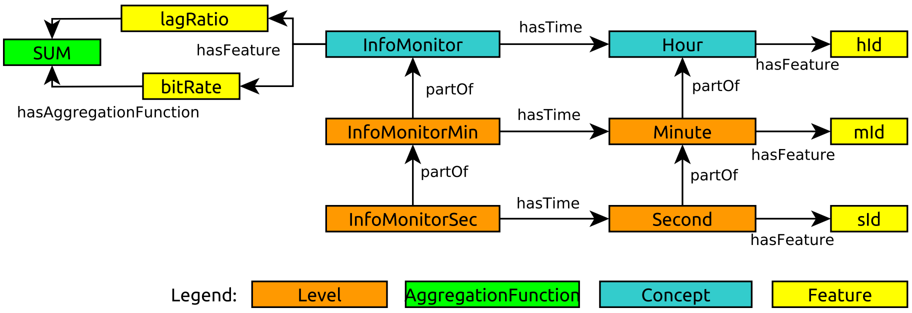


Fig. 4.6: Fragment of $\mathcal{G}_{agg}$ for SUPERSEDE

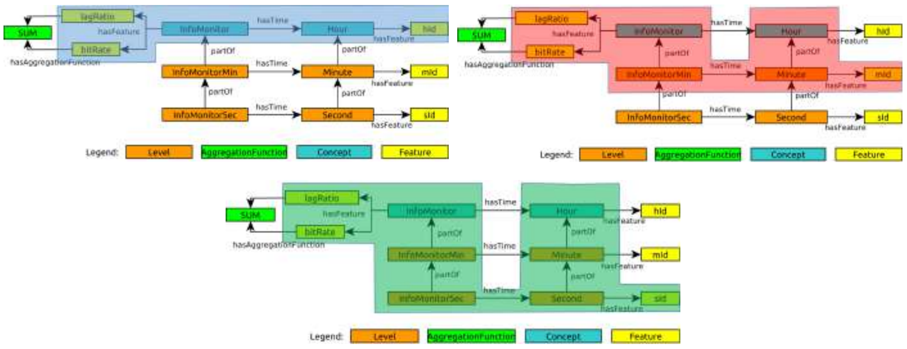


Fig. 4.7: LAV mappings, respectively, for w_1, w_2 and w_3 in $\mathcal{G}_{agg}$ specifying different levels of granularity

5.2 Generating CAQs

Here, we describe the algorithm that rewrites a query $Q_{\mathcal{G}}$ to CAQs over the wrappers. This is achieved leveraging the previously presented algorithm REWRITECQ. The aggregation graph $\mathcal{G}_{agg}$ allows us to rephrase a query posed over $\mathcal{G}$, where no aggregates have been explicitly stated, to a set of equivalent queries over $\mathcal{G}_{agg}$. Intuitively, we define a virtual graph (i.e., $\mathcal{G}_{virtual}$), as a copy of $\mathcal{G}$, where implicit aggregate queries will be considered as new (virtual) wrappers. Once all virtual wrappers have been defined, we can evaluate $Q_{\mathcal{G}}$ over $\mathcal{G}_{virtual}$ to obtain a resulting UCQs, where all data have been aggregated at the requested granularity. Algorithm 8 depicts the main procedure given an input query $Q_{\mathcal{G}}$.

Algorithm 8 is concept-centric, thus we define a mapping function *featuresPerConcept* that links a concept c to its set of queried features $\overline{f}$ (i.e., those present in π). For each pair $\langle c, \overline{f} \rangle$ in the map, we identify their adjacent hierarchies (i.e., sets of levels such that their top concept is connected to c), as well as their regular neighbors (i.e., neighboring concepts that do not

Algorithm 8 Rewrite CAQ

```
Input: $Q_\mathcal{G} = \langle \pi, \varphi \rangle$ is a global query
Output: $\overline{Q_\mathcal{S}}$ is a UCQs where all data provided at lower granularity levels have been
    implicitly aggregated
 1: function REWRITECAQ($Q_\mathcal{G}$)
 2:   $featuresPerConcept \leftarrow$ Map(key:$c \rightarrow$ val:$\overline{f}$)
 3:   for $p = \langle s, l, t \rangle \in patt(Q_\mathcal{G})$ do
 4:     if $p.\ell =$ hasFeature $\wedge$ $p.t \in proj(Q_\mathcal{G})$ then
 5:       $featuresPerConcept[p.s] \cup= p.t$
 6:   $\mathcal{G}_{virtual} \leftarrow \mathcal{G}$
 7:   for $\langle c, \overline{f} \rangle \in featuresPerConcept$ do
 8:     $\overline{\mathcal{H}} \leftarrow$ ADJACENTHIERARCHIES($c, \mathcal{G}_{agg}$)
 9:     $\overline{R} \leftarrow$ REGULARNEIGHBORS($c, \mathcal{G}_{agg}, \overline{\mathcal{H}}$)
10:     for $\overline{L} = \langle l_1, \ldots, l_n \rangle \in \mathcal{H}_1 \times \ldots \times \mathcal{H}_n$ do
11:       $\varphi_m \leftarrow$ MATCHQUERY($c, \overline{f}, \overline{L}, \overline{R}$)
12:       $\overline{Q_\mathcal{S}} \leftarrow$ REWRITECQ($\langle \varnothing, \varphi_m \rangle, \mathcal{G}_{agg}$)
13:       for $Q_\mathcal{S} \in \overline{Q_\mathcal{S}}$ do
14:         $Q'_\mathcal{S} \leftarrow$ GENERATECAQ($Q_\mathcal{S}, \pi, \mathcal{G}_{agg}$)
15:         $w \leftarrow$ Wrapper($att(Q'_\mathcal{S}), Q'_\mathcal{S} \rightarrow$ exec())
16:         Add the wrapper $w$ and its mapping $\mathcal{M}(w)$ to $\mathcal{G}_{virtual}$
17:   $\overline{Q_\mathcal{S}} \leftarrow$ REWRITECQ($Q_\mathcal{G}, \mathcal{G}_{virtual}$)
18:   return $\overline{Q_\mathcal{S}}$
```

conform an aggregation hierarchy). These are used to generate queries that combine levels in the aggregation lattice (i.e., match queries φ_m). A match query represents an equivalent query as $Q_\mathcal{G}$ at a lower granularity level, which is subsequently rewritten to its equivalent UCQs. Each of such resulting CQs is converted to a CAQ, where we distinguish among aggregable attributes (i.e., with aggregate functions) and those in the group-by set. This yields a new wrapper, now providing data at the same granularity level as that specified in $\mathcal{G}$, which is registered to $\mathcal{G}_{virtual}$. Finally, we execute the original query $Q_\mathcal{G}$ over $\mathcal{G}_{virtual}$. Next, we provide details on each of the specific methods used in REWRITECAQ.

Identify adjacent hierarchies. Algorithm 9 identifies the set of hierarchies $\overline{\mathcal{H}}$ that are adjacent to c in $Q_\mathcal{G}$. $\overline{\mathcal{H}}$ is composed of sets of levels that are linked via `partOf` edges such that $top(\mathcal{H})$ is connected to c. This is easily obtained using the Kleene closure in the pattern at line 5.

Example 5.2
For the fragment of $\mathcal{G}_{agg}$ depicted in Figure 4.6 and the exemplary query, the set $\overline{\mathcal{H}}$ would consist of a single hierarchy $\{Second \rightarrow Minute \rightarrow Hour\}$.

Algorithm 9 Adjacent hierarchies

```
Input: $c$ is a concept in $\mathcal{G}_{agg}$
Output: $\overline{\mathcal{H}}$ is the set of hierarchies adjacent to $c$
1: function IDENTIFYHIERARCHIES($c, \mathcal{G}_{agg}$)
2:   $\overline{\mathcal{H}} \leftarrow \varnothing$
3:   $\bar{t} \leftarrow \langle c, ?\ell, ?t \rangle(\mathcal{G}_{agg})$
4:   for $t \in \bar{t}$ do
5:     if $\exists\ \bar{l} \mid \langle ?l, \texttt{partOf+}, t \rangle(\mathcal{G}_{agg})$ then
6:       $\overline{\mathcal{H}} \cup= \bar{l}$
7:   return $\overline{\mathcal{H}}$
```

Identify adjacent regular neighbors. Algorithm 10 complements the previous step identifying all those concepts adjacent to *c* in the query (and their subclasses) that do not conform an aggregation hierarchy.

Example 5.3
From the previous input, the set $\overline{R}$ would contain the concept *Monitor*.

Algorithm 10 Adjacent regular neighbors

```
Input: $c$ is a concept in $\mathcal{G}_{agg}$, $\overline{\mathcal{H}}$ is the set of adjacent hierarchies
Output: $\overline{R}$ is the set of concepts adjacent to $c$ and their subclasses
1: function REGULARNEIGHBORS($c, \mathcal{G}_{agg}, \overline{\mathcal{H}}$)
2:   $\overline{R} \leftarrow \varnothing$
3:   for $t \in \langle c, ?\ell, ?t \rangle(\mathcal{G}_{agg})$ do
4:     if $t \notin \bigcup_{H \in \overline{\mathcal{H}}} top(H)$ then
5:       $\overline{R} \cup= t$
6:       $\overline{R} \cup= \{\exists\ sc \mid \langle sc, \texttt{subClass+}, t \rangle(\mathcal{G}_{agg})$
7:   return $\overline{R}$
```

Generate match query. Algorithm 11 depicts how we generate a match query for a given combination of levels. We generate a subgraph of $\mathcal{G}_{agg}$ such that it contains for the concept at hand its queried features and its regular neighboring concepts. Then, for each level *l* we include all levels in the hierarchy from *l* to the top, as well as their counterpart in the generated hierarchy of *c*.

Example 5.4
Following the previous example, queries φ_m would be generated, all covering the same set of features and regular neighbors. Regarding hierarchies

Q_1 would cover *InfoMonitor* $\rightarrow_{hasDate}$ *Hour*. Then, Q_2 would cover *InfoMonitorMin* $\rightarrow_{hasDate}$ *Minute* as well as all levels until the top of both vertices. Likewise, Q_3 would cover *InfoMonitorSec* $\rightarrow_{hasDate}$ *Second* and all levels until the top of both vertices.

Algorithm 11 Generate match query

Input: c is a concept, $\overline{f}$ is a set of features, $\overline{L}$ is a combination of adjacent levels, and $\overline{R}$ is its set of regular neighbors
Output: φ_m is a match query

```
1:  function MATCHQUERY($c, \overline{f}, \overline{L}, \overline{R}$)
2:    $\varphi_m \leftarrow \varnothing$
3:    for $f \in \overline{f}$ do
4:      $\varphi_m \cup= \langle c, \texttt{hasFeature}, f \rangle$
5:    for $R \in \overline{R}$ do
6:      $\overline{\ell} \leftarrow \langle c, ?\ell, R \rangle(\mathcal{G}_{agg})$
7:      for $\ell \in \overline{\ell}$ do
8:        $\varphi_m \cup= \langle c, \ell, R \rangle$
9:    for $l \in \overline{L}$ do
10:     $\langle x, \ell \rangle \leftarrow \langle ?x, \texttt{partOf+}, c \rangle \wedge \langle ?x, ?\ell, l \rangle(\mathcal{G}_{agg})$
11:     $\varphi_m \cup= \langle x, \ell, l \rangle$
12:     while $x \neq c$ do
13:       $top \leftarrow \langle x, \texttt{partOf}, ?top \rangle(\mathcal{G}_{agg})$
14:       $\varphi_m \cup= \langle x, \texttt{partOf}, top \rangle$
15:       $x \leftarrow top$
16:     while $\ell \neq l$ do
17:       $top \leftarrow \langle \ell, \texttt{partOf}, ?top \rangle(\mathcal{G}_{agg})$
18:       $\varphi_m \cup= \langle \ell, \texttt{partOf}, top \rangle$
19:       $\ell \leftarrow top$
20:   return $\varphi_m$
```

Generate CAQ. For each resulting CQ from the previous step, we now generate a CAQ that performs implicit aggregations (see Algorithm 12). We distinguish whether an attribute is aggregable (we need apply an aggregation function over it) or not (we include it in the group-by set).

Example 5.5
Each of the previously generated rewritings would yield one CAQ. Precisely aggregating *lagRatio* and using *hID* in the group-by set.

Algorithm 12 Generate CAQ

```
Input: $Q_{\mathcal{S}}$ is a CQ over the wrappers, $\pi$ is the set of queried features, $\mathcal{G}$ is the global graph
Output: $Q'_{\mathcal{S}}$ is a CAQ defined over $Q_{\mathcal{S}}$
1: function GENERATECAQ($Q_{\mathcal{S}}, \pi, \mathcal{G}$)
2:   for $a \in att(Q_{\mathcal{S}})$ do
3:     $agg \leftarrow \emptyset, groupBy \leftarrow \emptyset$
4:     if $\exists AF, f | \langle a, \texttt{sameAs}, ?f\rangle \wedge \langle ?f, \texttt{hasAggFunc}, AF\rangle(\mathcal{G}_{agg}) \wedge f \in \pi$ then
5:       $agg \cup= AF(a)$
6:     else
7:       $groupBy \cup= a$
8:   return $\texttt{CAQ}(agg, groupBy, Q_{\mathcal{S}})$
```

5.3 Discussion

In this subsection we first provide a cost formula for the computational complexity of REWRITECAQ, and later prove its soundness and completeness.

Computational complexity

Recall that Algorithm 8, for each concept in the query, exhaustively explores all combinations of levels for its adjacent hierarchies. Let C be the number of concepts covered by the query pattern φ, then we define H as the average number of adjacent hierarchies per concept and L their average number of levels. Thus, from the previous discussion we can easily see that the cost of rewriting a query to a set of CAQs is CL^H. Additionally, for each of the generated combinations we need to account for the cost of REWRITECQ, as presented in Section 4.5. Note that, in practice, few of the generated combinations will have covering wrappers. In this case, as we show in the experimental results, the cost of REWRITECQ is considerably smaller than when there exists covering wrappers.

Soundness and completeness

Here, we show that Algorithm 8 is sound and complete given a query $Q_{\mathcal{G}} = \langle \pi, \varphi \rangle$. Let us assume the set C with all covered concepts in φ, hence, here we refer to soundness and completeness in the sense that the following invariants hold: *(a)* $\mathcal{G}_{virtual}$ includes all concepts in C, *(b)* $\mathcal{G}_{virtual}$ includes all concepts that are (transitively) descendants of C (i.e., its levels), and *(c)* $\mathcal{G}_{virtual}$ includes all applicable mappings for the concepts in C and their levels. As preconditions, we assume $\mathcal{G}_{agg}$ is correctly instantiated, and REWRITECQ is minimally-sound and minimally-complete.

Proof. In the trivial case, no concepts in C have descendant levels, which entails that no wrapper provides data at a finer granularity than that specified in $Q_{\mathcal{G}}$. Then, no aggregation is required and no adjacent hierarchies will be identified in line 8. Thus, the resulting output from line 17 (i.e., REWRITECQ($\mathcal{Q}, \mathcal{G}_{virtual}$) is equivalent to REWRITECQ($\mathcal{Q}, \mathcal{G}$) which, from the preconditions, yields a minimally-sound and minimally-complete solution.

When some $C_i \in C$ has descendant levels, there exists at least a wrapper providing data at finer granularity. Then, Algorithm 9 would find all levels in $\mathcal{G}_{agg}$ (note the closure *partOf*+). Then, all combinations of participating levels are generated to compose match queries φ_m. As we are invoking REWRITECQ for each φ_m, and REWRITECQ is minimally-sound and minimally-complete, we would be obtaining all possible CQs for each combination of levels, satisfying the first two invariants. Next, the resulting rewritings are aggregated and added as new wrappers in $\mathcal{G}_{virtual}$ (together with their LAV mappings), satisfying the third invariant. As the three invariants hold we conclude that Algorithm 8 is sound and complete. □

6 Experimental evaluation

In this section we experimentally measure the performance of the proposed rewriting algorithms. Specifically, we aim to show how REWRITECQ behaves in realistic scenarios with respect to the theoretical complexity.

6.1 Experimental setting

For evaluation purposes, we systematically generate experimental runs (i.e., executions of REWRITECQ) with different characteristics. Specifically, we have the following six experimental variables that define an execution of REWRITECQ:

- Number of features per concept ($|F|$)
- Number of edges covered by a query ($|E_Q|$)
- Overall number of wrappers ($|W|$)
- Number of edges covered by a wrapper ($|E_W|$)
- Fraction of features in a concept covered by a query ($Frac_Q$)
- Fraction of features in a concept covered by a wrapper ($Frac_W$)

Table 4.1 depicts the domain for each variable, using realistic values similar to those found in the literature.

Variable	Domain
$\lvert F\rvert$	$\{5, 10, 20\}$
$\lvert E_Q\rvert$	$\{2, 4, 6, 8, 10, 12\}$
$\lvert W\rvert$	$\{2, 4, 8, 16, 32, 64, 128\}$
$\lvert E_W\rvert$	$\{2, 4, 6, 8, 10, 12\}$
$Frac_Q$	$\{0.3, 0.6, 0.9\}$
$Frac_W$	$\{0.3, 0.6, 0.9\}$

Table 4.1: Domain for each experimental variable

The process of generating an experimental run (i.e., a query and a set of covering wrappers) consists of obtaining random subgraphs of a large enough clique playing the role of $\mathcal{G}$, which guarantees the desired randomness. Algorithm 13 depicts the process of generating experimental runs based on the previously introduced variables.

Algorithm 13 Generate an experimental run (query and wrappers)

```
Input: $\mathcal{G}$ is the global graph (here a clique), $|F|$, $|E_Q|$, $|W|$, $|E_W|$, $Frac_Q$, $Frac_W$
Output: $Q_\mathcal{G}$ is a global query, $W$ is a set of wrappers covering $Q_\mathcal{G}$
1: function generateExperimentalRun($\mathcal{G}$, $|F|$, $|E_Q|$, $|W|$, $|E_W|$, $Frac_Q$, $Frac_W$)
2:   $Q_\mathcal{G} \leftarrow$ connected random subgraph of $\mathcal{G}$ with $|E_Q|$ edges
3:   $Q'_\mathcal{G} \leftarrow$ with a probability $Frac_Q$ of appearing, expand $Q_\mathcal{G}$ with up to $|F|$ features
4:   $W \leftarrow \varnothing$
5:   for $i \leftarrow 1$ to $|W|$ do
6:     $w \leftarrow$ connected random subgraph of $Q_\mathcal{G}$ with $|E_W|$ edges
7:     $w' \leftarrow$ with a probability $Frac_W$ of appearing, expand $Q_\mathcal{G}$ with up to $|F|$ features
8:     $W \cup= w'$
9:   return $\langle Q'_\mathcal{G}, W\rangle$
```

For each combination of variables in the domain, we generate an experimental run and invoke rewriteCQ. For each experimental run, we measure the size of the resulting UCQs (U) and the processing time (R) in milliseconds. To account for variability, we generate experimental runs using the same parameters three times and keep the median of R. The experiments are performed on a machine running GNU/Linux with an Intel Core i5 processor running at 3.5 GHz and with 16GB of RAM memory.

We implemented a prototype of the rewriting algorithms[5]. The implementation is based on SPARQL, where each construct is represented as a *resource description framework* (RDF) graph. The Jena library is used to manipulate the graphs and queries.

[5]https://github.com/serginf/MDM

6.2 Experimental results

From the obtained results, we observe that U and R are highly correlated (i.e., Pearson correlation of $\rho = 0.997$), thus we only report on R. The previously presented theoretical complexity is always an upper bound, which is not plotted in the figures to avoid hindering their visualization. In this subsection we report on the most significant experimental results that show the trend of our approach. In Appendix B, we exhaustively report all results and provide a more elaborated discussion on them.

Evolution of response time based on wrappers

We first analyse how the response time evolves based on the number of wrappers. To this end we plot the evolution of R for different values of $|W|$. As depicted in Figure 4.8, in general there is an exponential trend for R as the number of sources (i.e., wrappers) grow. Nonetheless, we can see our approach can efficiently deal with a large number of sources (i.e., 128) while the number of edges in the query is relatively small. With an increased number of covered edges in φ the cost also exponentially grows.

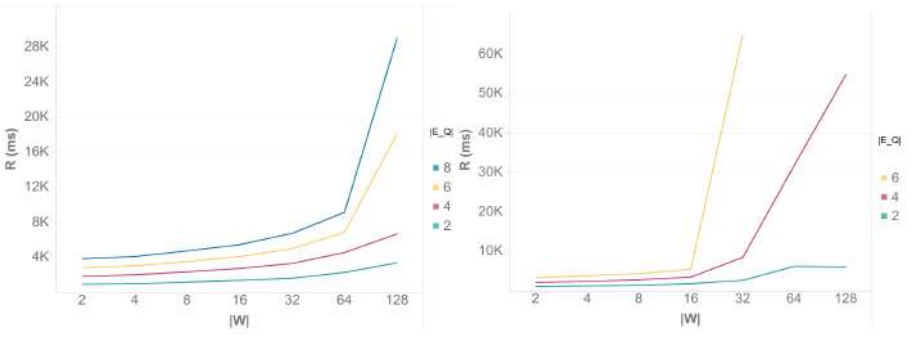


Fig. 4.8: Evolution of R w.r.t. $|W|$ for $|F| = 5$ and $|F| = 20$

Evolution of response time based on edges in the query.

In the second experimental analysis, we are concerned with studying the impact of the size of the query on the time to perform a rewriting. To this end, we plot the evolution of R for different values of $|E_Q|$. As depicted in Figure 4.9, the cost of rewriting is almost linear regardless of $|E_Q|$ for low values of $|E_W|$. This is not a surprising result, as we can expect a large pruning of candidate solutions in the intra-concept generation phase. As the number of covered edges by wrappers grows, we start seeing variability and a more exponential trend.

Note we have filtered out $|W| = 128$ due to the high variability yield in the results caused by failures in the rewriting process due to the size of intermediate results, which hindered the visual analysis.

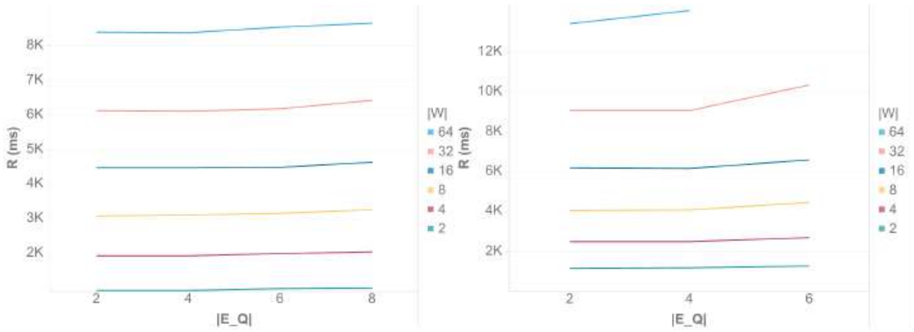


Fig. 4.9: Evolution of R w.r.t. $|E_Q|$ for $|F| = 5$ and $|F| = 20$

7 Conclusions

We have presented an approach to tackle the problem of answering queries using views under semantic heterogeneities and evolution. We have proposed a purely graph-based data integration system, which allows us to represent all integration constructs (i.e., global schema, LAV mappings and source descriptions) as well as the necessary semantic annotations to drive the query rewriting process. We have presented a minimally-sound and minimally-complete rewriting algorithm for an input global query. Next, we deal with semantic heterogeneities by automatically generating combinations of queries (and rewriting them) at different levels of granularity, thus performing implicit aggregations. We have presented experimental results showing the efficiency in practice of the proposed methods.

Acknowledgements. This work has been partly supported by the H2020 SUPERSEDE project, funded by the EU Information and Communication Technologies Programme (num. 644018), and the GENESIS project, funded by the Spanish Ministerio de Ciencia e Innovación (num. TIN2016-79269-R).

Chapter 5

SLA-driven Selection of Intermediate Results to Materialize

This chapter has been published as a paper in Fundamenta Informaticae, 163(2): 111-138 (2018). The layout of the paper has been revised.
DOI: https://doi.org/10.3233/FI-2018-1734

Co-authoring declaration. This work has been done together with the PhD student Rana Faisal Munir, with an overall equal contribution from both. Precisely, the introduction (Section 1), problem formulation (Section 2) and definition of cost model for intermediate result materialization selection (Section 3) were done jointly with equal contribution. The state space search algorithm (Section 4) was done by Sergi Nadal, while a data format selection approach (not included in this thesis) was done by Rana Faisal Munir. The experimental evaluation (Section 5) was jointly developed, with focus from Sergi Nadal on the intermediate result selection evaluation.

Abstract

Data-intensive flows deploy a variety of complex data transformations to build information pipelines from data sources to different end users. As data are processed, these workflows generate large intermediate results, typically pipelined from one operator to the following ones. Materializing intermediate results, shared among multiple flows, brings benefits not only in terms of performance but also in resource usage and consistency. Similar ideas have been proposed in the context of data warehouses, which are studied under the materialized view selection problem. With the rise of Big Data systems, new challenges emerge due to new quality metrics captured by service level agreements which must be taken into account. In this chapter, we propose a novel approach for automatic selection of multi-objective materialization of intermediate results in data-intensive flows, which can tackle multiple and conflicting quality objectives. The experimental results show that our approach provides 40% better average speedup with respect to the current state-of-the-art.

1 Introduction

Nowadays, many organizations are shifting their business strategy towards data analytics in order to guarantee their success. In the past, the vast majority of analysed data was transactional, however the emergence of Big Data systems allows a new range of data analytics, by replacing traditional extract-transform-load (ETL) process with much richer data-intensive flows (DIFs) [82]. This new range of data analytics is supported by the Hadoop[1] ecosystem which has a distributed storage system (Hadoop Distributed File System - HDFS[2]) to store large scale data and a processing engine (i.e., MapReduce [38]) to execute DIFs. It works on a distributed cluster of commodity hardware which provides competitive advantage to organizations by reducing their hardware costs. In addition, many modern cloud providers offer pay-per-use services to organizations by implementing the big data systems under service level agreements (SLAs).

An in-depth study of analytical workloads, in Big Data systems across seven enterprises, shows that user workloads have high temporal locality, as 80% of them will be reused by different stakeholders on the range of minutes to hours [35]. Thus, providing partial materialization of results in shared flows can clearly bring benefits by saving computational resources. However, the aforementioned study raises the question of *"what intermediate results to materialize?"*. This boils down to the traditional data management problem of *materialized view selection* [72], which is well-known to be NP-hard [67].

This question is not easily addressable, despite the efforts of the research community. Some works [123, 44, 161] have tackled the problem of finding the optimal partial materialization in DIFs, however all of them are specific to the MapReduce framework and only aim at optimizing the system performance-wise by ignoring other relevant SLAs (such as freshness, reliability, scalability, etc.[143]). Moreover, the aforementioned solutions do not consider different characteristics associated with different SLAs. For instance, in some organizations, they allow to get results from a stale materialized node (i.e., to allow low freshness) for a certain time period to reduce the loading cost. These characteristics can be expressed separately and the optimal value should be chosen for each materialized node. These shortcomings of existing solutions are addressed by our proposed approach, which is a technology independent materialization solution and can take into consideration generic quantifiable SLAs with their associated characteristics.

1.1 Motivational example

sec:motexample To motivate our work, we present a DIF, shown in Figure 5.1,

[1]https://hadoop.apache.org

[2]https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

which depicts a high-level representation containing relational operations and User Defined Functions (UDFs). It uses five input sources and serves three queries. Each data source and data operator is labeled by its estimated processing cost (i.e., consumed resources, in seconds) and storage cost (in GB). Note that data processing entails extracting and loading data from the sources into the data processing system.

For the sake of this example, let us suppose that all the sources update once per day, except *Source 1* and *Source 3* that have a update frequency of 6 and 4 times per day, respectively. *Query 1*, *Query 2* and *Query 3* have a frequency of 2, 20, and 10 times per day, respectively. In addition, let us assume that we allow stale materialized results and it is provided as a characteristic vector (given as number of updates per time unit $[1, 2, ..., n]$).

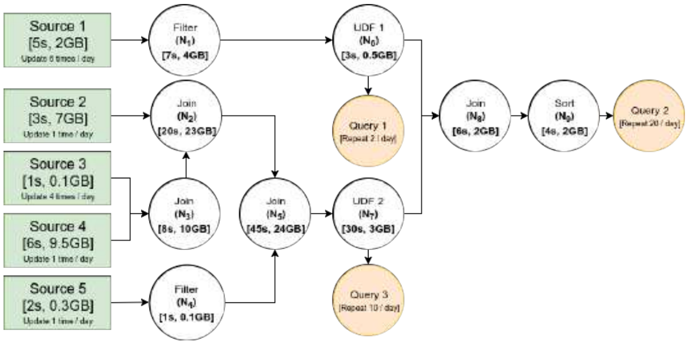


Fig. 5.1: An Example of a DIF

In this example, we focus on optimizing four SLAs (i.e., time to load, time to query, space needed to store intermediate results, and freshness). *Loading time* is measured by the sum of processing cost from the sources to the partial materializations, *query time* is measured by the sum of execution cost from the partial materializations to the user's output, *storage space* is measured by the sum of storage cost for the selected partial materializations, and *freshness* is estimated using the cost function presented in Section 3.3.

Several of the existing intermediate result materialization approaches from Big Data Systems can be used, as discussed in Section 6. Let us focus on one of them, namely ReStore [44], which uses two kinds of heuristics (i.e., conservative and aggressive) in order to choose DIF nodes for materialization. Conservative heuristics materialize the output of those operators that reduce the input size (i.e., project and filter). The aggressive heuristics materialize the output of those operators which produce large outputs and those known to be computationally expensive (i.e., join, group and cogroup). Table 5.1 shows the selected nodes and a quadruple with the costs (i.e., loading time, query time, storage space, and freshness). It should be noted that *query time* represents the sum of times of all three queries. The first two columns show both ReStore's heuristics, while the rightmost shows a pareto-optimal solution (i.e., a solution that cannot be improved further in the presence of multiple conflicting SLAs) in boldface.

	ReStore Cons.	ReStore Agg.	Pareto-optimal
Nodes	N_1, N_4	N_2, N_3, N_5, N_8	$\boldsymbol{N_7}$
Cost	$\langle$75 s, 257 s, 4.1 GB, 1$\rangle$	$\langle$336 s, 115 s, 59 GB, 1$\rangle$	$\langle$**101 s, 100 s, 3 GB, 0.78**$\rangle$

Table 5.1: Selected intermediate nodes and cost for the four SLAs (load, query, store, freshness)

As shown in Table 5.1, ReStore conservative heuristics choose N_1 and N_4. They take more time in loading, due to *Source 1*, which updates very frequently and effects the loading cost of N_1. We calculate the total load time of a node by multiplying its load time with the update frequency of its input sources. Furthermore, both N_1 and N_4 do not provide good speedup because they have high query time. ReStore aggressive heuristics choose N_2, N_3, N_5, and N_8. These nodes help to reduce the query time, but require more space to store and more time to load. It should be noted that ReStore does not support freshness. If the input sources change, it deletes all their dependent materialized nodes. This pull-strategy provides a fixed degree of freshness and thus, we set it to 1 in our quadruple.

Finally, the pareto-optimal solution considers four SLAs (i.e., loading time, query time, storage space, and freshness) together by assigning them the same weight, and based on them, it chooses only node N_7, which provides better speedup compared to ReStore's heuristics. Even though, N_7 depends on *Source 3* , which has a high loading cost due to its high update frequency, it is still worth to materialize because it is reused more often by repetitive queries (i.e., Query 2, Query 3). Moreover, the loading cost can be improved by choosing the optimal value of refresh frequency for N_7. The possible values are $[1, 2, 3, 4]$, where 4 will provide the maximum freshness. The pareto-optimal solution chooses 3 as the refresh frequency for N_7, which helps to improve the load time and it also provides a good degree of freshness.

The above given example shows that the state-of-the-art solutions produce suboptimal results in the case of different SLAs. To address this problem, we revisit the traditional frameworks for materialized view selection [153] and analyse its applicability and extensions in the context of DIFs for Big Data systems. As a result, we present an approach to automatically select the optimal materialization of intermediate results driven by multiple quality objectives represented as quantifiable SLAs with their associated characteristics. This is achieved by implementing a multi-objective optimization technique (discuss in Section 4) which efficiently tackles multiple and conflicting objectives to select materialized nodes. Moreover, for each to-be-materialized node.

Contributions. The main contributions of our work are as follows:

- We propose a novel cost model for multi-objective selection of optimal partial data materialization for DIFs.
- We present a local search algorithm that, driven by SLAs, probabilistically selects a set of near-optimal intermediate results to materialize.
- We assess our method and show its performance gain by using the TPC-H benchmarking suit.

Outline. The rest of the chapter is structured as follows. Section 2 presents the theoretical building blocks and formalizes our approach. Sections 3 and 4 present the cost model for intermediate result selection and the algorithm to explore the search space. In Section 5, we present the experimental results. Section 6 discusses related work and Section 7 concludes the chapter.

2 Formal Building Blocks and Problem Statement

2.1 Multiquery AND/OR DAGs and data-intensive flows

The general framework for materialized view selection [153] relies on multiquery AND/OR Directed Acyclic Graphs (DAGs). As defined in [155], a query DAG is a bipartite graph $\mathcal{G}$, where AND nodes (or *operational nodes*) are labeled by a relational algebra operator, and OR nodes (or *view nodes*) are labeled by a relational algebra expression. Moreover, given a set of queries Q defined over a set of source relations R, a multiquery DAG $\mathcal{G}$ is a query DAG, which may have multiple sink (query) nodes. Roughly speaking, the materialized view selection problem can be expressed as a search space based problem over the multiquery DAG $\mathcal{G}$. Additionally, [154] formalizes the output of such problem as a data warehouse (DW) configuration $C = \langle V, Q^V \rangle$, where Q^V is the set of queries in the query set Q rewritten over the view set V. Note that there exist two special DW configurations: $\langle Q, Q^Q \rangle$ which represents a materialization of the query set Q and $\langle R, Q \rangle$ which represents a complete materialization of the source data stores R.

However, the multiquery AND/OR DAGs fail to capture the complex semantics present in DIFs operators, as they solely rely on relational operators. To this end, in this chapter we build upon the ideas from the aforementioned frameworks and adapt them for the case of DIFs, which consider more complex data transformations [85]. It is straightforward to see that any multiquery DAG $\mathcal{G}$ can be represented as a DIF, however the opposite does not hold due to the fact that AND/OR DAGs are solely based on relational operators, while DIFs are extended with more complex operations. Thus, in this chapter, we extend the notion of DW configuration to **Big Data system (BDS) configuration** for the case of DIFs. Hereinafter, we will depict a BDS configuration as a set of nodes from the DIF to materialize $B = \{b_1, \ldots, b_n\}$. In the following sections, we describe the specific components for the problem in-hand, and reformulate the materialized view selection problem in the context of BDS.

2.2 Components

Data-Intensive Flow. In this chapter, we adopt the notation from [83], hence we define a DIF D as a DAG (V, E) where its nodes (V) are the flows' operational nodes, and its edges (E) represent the directed data flow. Operational nodes are defined as $o = \langle \mathbb{I}, \mathbb{O}, \mathbb{S}, V_{pre} \rangle$, where $\mathbb{I}$ and $\mathbb{O}$ are sets of respectively input and output schemata, where each schema is defined as a finite set of attributes, $\mathbb{S}$ expresses operator's semantics, and V_{pre} a subset of attributes of the input schemata ($V_{pre} \subseteq \bigcup_{I \in \mathbb{I}} I$) whose values are used by o.

Design Goal (DG). $\mathbb{DG}$ represents a set of design goals, where each (DG_i) characterizes an SLA. It can be specified as either a minimization or a maximization of an objective cost function, or alternatively as a boundary that must not be surpassed in such cost function. Formally:

- *Min/Max*: From a set of BDS configurations $\mathbb{B}$, it returns the minimal B by means of evaluating the cost function CF, defined as $DG_{min}(\mathbb{B}) = \min_{B \in \mathbb{B}} (CF(B))$. Note that maximizing the cost function is equivalent to the negation of minimization.
- *Constraints*: For a BDS configuration, it checks whether the evaluation of the cost function CF fulfills the constraint K, formally $DG(B) = [CF(B) \leqslant K]$. Note that the constraint can express an arbitrary logical predicate (e.g., $<, >, \geqslant$). It is important to note that DG(C), where C is constraints, in this case is binary true/false and it differs from the above which gets the value obtained from the cost function.

Cost Function (CF). Given a BDS configuration, $\mathbb{CF}$ represents a set of cost functions where each (CF_i) is the estimation of an SLA for B. Hence, we define $CF(B) = \sum_{b \in B} E(b)$ (where $E(b)$ is the cost estimation of an element $b \in B$).

Characteristics Vector (CV). Some costs are determined once a node is chosen, but for SLAs, we can select arbitrary values for the features that impact them. For instance, in some organizations, they allow stale data for some period of time, which can be defined as a refresh frequency for every materialized node. Thus, the refresh frequency should be chosen to maximize the overall benefit. We keep a vector of such choices. Each position of the vector represents a characteristic affecting some SLAs. These characteristics will impact on the associated cost function CF.

Utility Function. In the context of multi-objective optimization (MOO) [108], it is common to aggregate all objectives $DG_1, \ldots, DG_n$ into a single value to obtain the global utility. Such function, known as the *utility function* U as it measures benefit, is formally defined as $U(\mathbb{DG}) = U(CF_1(B), \ldots, CF_n(B))$. Each $CF_i(B)$ provides a quantitative evaluation of B (it can be seen as individual utility functions for each cost function) for its associated DG_i, in the case of min/max design goals, or 0, and $+\infty$ for satisfied and non-satisfied constraints, respectively. Generally in MOO higher utilities are preferred. However, in our context there are some CF where we aim for minimal utilities (e.g., query time).

2.3 Problem statement

We state the problem of intermediate results materialization selection in DIFs as: given a data-intensive flow D, a set of design goals $\mathbb{DG}$, a set of cost functions $\mathbb{CF}$, a characteristics vector $\mathbb{CV}$, a utility function $U(\mathbb{DG})$, and a cost model represented by a set of estimators over D calculated by means of statistical information from sources, return a BDS configuration B', such that, $U(\mathbb{DG})$ is minimal, each $b \in B'$ with its optimal characteristics values for $\mathbb{CV}$.

3 Cost Model for Intermediate Results Materialization Selection

In this section, firstly we present our approach to estimate statistics for each operation of a DIF. We assume that the statistics of each input source are available. Secondly, we discuss the metrics (i.e., execution and storage) that we consider in this chapter. It should be noted that we choose to ignore the CPU cost, and focus only on the I/O operations. Also, regardless of being executed in parallel, the overall execution cost of the flow will remain the same (only time span would be reduced). Finally, we use the proposed metrics to estimate the cost of SLAs. In this chapter, we present the cost functions for four SLAs (loading, query, storage, and freshness).

3.1 Data-intensive flow statistics

As previously mentioned, cost functions are computed from estimators. Every operational node in a data-intensive flow D might have several estimators, each assessing a single SLA (e.g., an execution cost), where they perform a cost based estimation according to the operator's semantics. In order to devise more accurate metrics, some essential statistics must be obtained from the input data stores and propagated across D. By topologically traversing D, we can propagate such statistics at each node, based on the specific semantics of operators. In [68], the authors describe a complete set of statistics which are necessary to perform cost based estimations for DIFs. Here we focus on the following subset: selectivity factor $sel_P(R)$, number of distinct values per attribute $V(R.a)$ and cardinality $T(R)$. R denotes an input data store, while $R.a$ is an attribute of R. Note that statistics only consider logical properties of the flow, hence they are independent of the underlying engine where the flow is executed.

Example 3.1
JOIN operator Let us assume a *JOIN* operator $R' = R \bowtie S$ (e.g., N_6 in Figure 5.1), with input schemata $R(a, b)$, $S(c, d)$ and semantics $P_{R.a=S.c}$. Inspired by the work in [53], we propose measures for the above-mentioned statistics for this *JOIN* operator (we have done likewise for the rest of operators) as:

$$sel'_P = \begin{cases} \dfrac{1}{V(R.a)}, & \mathit{if\ domain}(S.c) \subseteq \mathit{domain}(R.a) \\[2ex] \dfrac{V(R.a \cap S.c)}{V(R.a) \cdot V(S.c)}, & \mathit{otherwise} \end{cases}$$

$$V(R'.att_i) = V(R.att_i) \cdot (1 - sel_P)^{\frac{T(R)}{V(R.att_i)}} \qquad T(R') = sel'_P(R \bowtie S) \cdot T(R) \cdot T(S)$$

The selectivity factor is obtained as the fraction of the number of shared values in the *JOIN* attributes, when the domain of the right-hand side is contained in the domain of the left-hand side (i.e., analogously to Primary Key(PK)-Foreign Key(FK) relations), otherwise an estimation is made as a fraction of shared values and its Cartesian product. Regarding the number of distinct values for an attribute, it is estimated as the input number of distinct values, multiplied by the probability that no repetitions of a value are selected. Finally, the cardinality is measured likewise the relational case.

3.2 Metrics

Once statistics for D have been calculated, they can act as building blocks for metrics. Here, we focus on estimating both performance-wise ($Execution_{estimator}$) and space-wise ($Space_{estimator}$) metrics. Performance metrics are measured by means of estimated disk I/O (in blocks) and space metrics by the number of disk blocks occupied by the intermediate results materialization. It is worth noting that in terms of execution, the CPU cost is negligible as opposed to

I/O cost [6], hence we ignore CPU cost and focus on the I/O cost of operators. Therefore, non-blocking operational nodes (acting as pipelines) will not incur any cost for such $Execution_{estimator}$.

To devise metrics, certain characteristics of the underlying engine are required. We focus on the following subset: the size of a disk block B, the number of main memory buffers available M, and the size in bytes that each attribute occupies $sizeOf(att_i)$. For instance, in the Oracle relational database the block size is approximately of 8KB, while in Hadoop's HDFS it is 64MB or 128MB. The incurred space of intermediate results is measured by means of the estimated number of blocks generated. However, this will vary according to the underlying schema that such results have and therefore, we need to make this calculation based on the record length, that is $sizeOf(att_i)$ (including the corresponding control information). Thus, the specific number of blocks for an input R is measured as:

$$B(R) = \left\lceil \frac{T(R)}{\left\lfloor \frac{B}{\sum sizeOf(att_i)} \right\rfloor} \right\rceil$$

Example 3.2
JOIN operator cont. Given the *JOIN* operation from example 3.1, one implementation of such operator is based on the block-nested loop algorithm, which scans S for every block of R using $M-2$ memory buffers (as the remaining two are used to perform tuple comparisons and output results), thus the estimation for execution and space costs is as follows:

$$Execution_{estimator} = B(S) + B(R) \cdot \left\lceil \frac{B(S)}{M-2} \right\rceil \qquad Space_{estimator} = B(R')$$

However, in a MapReduce environment, execution cost is dominated by data transfers (i.e., communication cost over the network) that occurs during the data shuffling phase between mapper and reducer nodes [5]. In such case, the natural implementation of a $JOIN$ is using the hash join algorithm, where the hash function maps keys to k buckets and data is shipped to k reducers. Assuming no data skewness, each reducer receives a fraction of $\frac{T(R)}{k}$ and $\frac{T(S)}{k}$. Having c as a constant representing the incurred network overhead per transferred HDFS block, the cost estimations of the *JOIN* are:

$$Execution_{estimator} = \frac{B(R) + B(S)}{k} \cdot c \qquad Space_{estimator} = B(R')$$

Note that the presented metrics can be highly variable within D. For instance, not surprisingly, *JOIN* nodes are the operations that consume the most time and space in order to generate intermediate results. Additionally, modern DIFs make heavy usage of User Defined Functions (UDFs) which consists of ad-hoc operations, difficulting the estimation of their I/O cost. An approach to solve this problem is to rely on static analysis of code to estimate the I/O cost for UDFs [76]. Finally, it is worth noting that other approaches exist to measure the presented metrics, for instance [142] proposes a method based on micro-benchmarking. On the contrary, our approach does not require any execution of the flow which however, impacts the quality of the estimation.

3.3 Cost functions

In this section, we present a set of cost functions to evaluate a BDS configuration, based on metrics from the materialized operational nodes of D. In our experiments, we focus on traditional metrics used in multi-query optimization namely loading cost, query cost, storage cost and freshness. However, note that our approach is extensible to other types of metrics such as monetary aspects [121], energy consumption [135], etc. For instance to estimate monetary cost, the pay-per-use cloud services charge based on the resources used, which can be estimated by our cost model. Further, the estimated resource utilization can be used to calculate the cost of renting machines on different cloud providers. Regarding storage, here we are not concerned with the layout to be used as this is assessed once the selection of nodes to be materialized has been found.

First, we must present some auxiliary methods over BDS configurations in which cost functions are based on. Let $Pre(b)$ and $Suc(b)$ be respectively the input and output subgraphs for a node b, recursively defined as $Pre(b) = \{b\} \cup \forall b_i \in predecessors(b)\ Pre(b_i)$ and $Suc(b) = \{b\} \cup \forall b_i \in successors(b)\ Suc(b_i)$, and respectively finishing when the indegree and outdegree of b is 0. Hence, we can define the input and output subgraphs of a BDS configuration B as $I(B) = \bigcup_{b \in B} Pre(v)$ and $O(C) = \bigcup_{v \in C} Suc(v)$. Specifically, the former is a subgraph where its source nodes are the sources in a D and sink nodes are all the elements $b \in B$. The latter is a subgraph where its source nodes are all elements $b \in B$ and sink nodes are the final nodes in D. Additionally, $sources(b)$ gives the input sources of a node b and $sinks(b)$ provides the queries over a node b.

Loading Cost. The cost of loading a set of intermediate results CF_{LT} is the sum of processing source data and propagating them to the intermediate results in B. Our approach is valid for both maintenance and update of intermediate results, as long as source statistics are properly updated. From a BDS configuration B, the estimated loading cost is intuitively the cost of executing the operations of D, loading the intermediate results for each node $b \in B$ (i.e., $I(B)$), and the cost of writing such results to the disk. Thus, we define $CF_{LT} = \sum_{b \in B}[\sum_{b_i \in I(b)} Execution_{estimator}(b_i) * RF(b_i)] + \sum_{b \in B} Space_{estimator}(b)$. Here, RF represents the refresh frequency of materialized nodes which is fixed in the characteristics vector CV of each node. The unit of refresh frequency is *total number of updates per time unit*. It should be noted that we are talking about sequential files that do not provide random access, so only full update is possible (no incremental).

Query Cost. The query cost CF_{QT} is the sum of querying the intermediate results, transform the data and deliver results to the user. From a BDS configuration B, the estimated query cost is computed as the sum of execution costs of successor nodes for each node $b \in B$ (i.e., $O(C)$). However, note that the cost of processing the operations of the nodes in B should not be taken into account as it is already evaluated in CF_{LT}. Therefore, it is necessary to consider only nodes in the set $O(B)\backslash B$, denoted $O^+(B)$. Finally, it is necessary to consider the cost of reading such intermediate results from the disk. Hence, we define $CF_{QT} = \sum_{b \in B}[\sum_{b_i \in O^+(b)}(Execution_{estimator}(b_i) * (\sum_{s_i \in sinks(bi)} QF(s_i))] + \sum_{b \in B} Space_{estimator}(b)$. Here, QF represents the frequency of queries. The query frequency can be expressed per day, hour or minute. QF helps to select the most reused node. Queries with high frequencies benefit more from the re-usage. Hence, a node which is used in highly repetitive queries will be given more weight during selection.

Storage Cost. The storage cost function CF_S concerns the storage space needed to store intermediate results. It is computed as the sum of estimated space for storing the results of each node in B, and it can be seen as the estimated space require to accommodate the deployed BDS configuration. It is defined as $CF_S = \sum_{b \in B} Space_{estimator}(b)$. Notice that $Space_{estimator}$ can be used for estimating the costs of reading and loading intermediate results, showed in CF_{LT} and CF_{QT}, as well as to estimate the occupied space for the case of minimizing or constraining its value.

Freshness. The freshness cost function estimates the freshness of the results of a query, which are obtained using materialized nodes, denoted as B'. For the freshness function, we build on the formula from [138]. The variable *age* tells how old data are in a materialized result with regard to the current data in the input sources. In our case, age cannot be known as it is not possible to foresee when materialized results are going to be used. It should be noted that update frequency of a node is calculated based on its input sources. Whereas, the refresh frequency is given in the characteristics vector CV of each node. We calculate the update frequency of a materialized node b as an average of the input frequencies of its input sources $UF(b) = Average_{s_i \in sources(b)} UF(s_i)$. Then, we can approximate *age* of b as the mid point between two refreshments $Age(b) = RF(b)^{-1}/2$. With such, we can measure the freshness of a node b as $Freshness(b) = (1 + UF(b) * Age(b))^{-1}$. Furthermore, $Freshness(b)$ is used to calculate the freshness of the results of a query Q as $CF_{Freshness}(Q_{results}) = Average_{b' \in B'} Freshness(b')$. This cost function helps to choose a node for materialization which provides up-to-date results to the queries.

4 State Space Search Algorithm

As previously mentioned, the problem of intermediate results materialization in DIFs can be reduced to the general materialized view selection problem, known to be NP-hard. Hence, we must avoid exhaustive algorithms and rely on informed search algorithms. Furthermore, in this particular case, purely greedy algorithms will not provide near-optimal results as the proposed cost functions are not monotonic. In classic artificial intelligence, a state space search problem is usually represented with the following components: *(i) initial state* where to start the search; *(ii) set of actions* available from a particular state; *(iii) transition model* describing what each action does and what are the derived results from it; *(iv) goal test* which determines whether the evaluated state is the goal state (i.e., the optimal state); and *(v) path cost* function to assign cost to the actions path.

In our context, we see a state as any BDS configuration B over which action functions are applied. It is noteworthy to mention that in such problem we are not interested in the set of actions that have led to a solution, but in the solution itself, which is initially unknown. Additionally, as any state B is a valid solution, we drop the component of *goal state*. Furthermore, the path cost is substituted by the definition of a *heuristic function*, which will guide the search. In the following subsections, we present the particularities of our specific problem for the remaining components.

4.1 Actions

For a BDS configuration B, we can compute actions (navigations over the graph), yielding new BDSs B'. First, we define the generic navigation operation $B' = Nav(b_{origin}, b_{destination})$, with $b_{origin}, b_{destination} \in D$ and semantics $Nav(b_{origin}, b_{destination}) = \left(B \backslash \{b_{origin}\}\right) \cup \{b_{destination}\}$. We then define three specific actions applied over nodes in B:

1. *Forward* ($F(b, b') = Nav(b, b')$): characterizing a forward movement from b to b' in D, applicable when $b' \in successors(v)$.

2. *Backward* ($B(b, b') = Nav(b', b)$): characterizing a backward movement from b' to b in D, applicable when $b' \in predecessors(b)$.
3. *Stay* ($N(b) = \emptyset$): always applicable, as it does not perform any movement. Such operator is only useful when other nodes b' are combined with M or U, so that a new state is generated where b remains selected.
4. *Increment* ($Inc(b, i)$): Increases the value of a characteristic (identified by position i_{th} of the vector CV) for a node b.
5. *Decrement* ($Dec(b, i)$): On the contrary, it helps to decrease the value of a characteristic for node b at i_{th} position of the vector CV.

From the previous definitions, for each node b, we define the set $Actions(b)$ as:

$$\bigcup_{b_i \in successors(b)} \{F(b, b_i)\} \cup \bigcup_{b_i \in predecessors(b)} \{B(b, b_i)\} \cup \{N(b)\} \cup \{Inc(b, i)\} \cup \{Dec(b, i)\}$$

Finally, we obtain all possible actions from a BDS configuration B by computing the Cartesian product of the power set of each $Actions(b_i)$ (note, empty sets are removed from each power set but this is not depicted for readability) as $\mathcal{P}(Actions(b_1)) \times \ldots \times \mathcal{P}(Actions(b_n))$. The rationale behind this operation is to generate, for each node b, all combinations of movements. The usage of a power set is relevant for the cases when the input or output schemata of a node is not unary (e.g., a $JOIN$). Then, such different combinations are furtherer combined with the rest of nodes via a Cartesian product. Note that such set can be extremely large for complex Ds, however it is easy to see that many combinations generate invalid BDS configurations. To this end, we define the two essential conditions that a BDS configuration must fulfill in order to be valid, namely answerability and non-dominance.

Answerability of all queries. Ensuring that all queries (sink nodes) can be answered from materialized results. It can be checked by guaranteeing there is at least one materialized node for each path in D. Formally, $\forall b \in sources(D) \forall p_i \in Paths_{b,sinks(D)} \exists node \in p_i : node \in B$. For instance, in Figure 5.2a, we can see that the green-colored BDS configuration does not satisfy answerability as the path from N_2 to N_9 does not contain any materialized node.

Non-dominance of nodes. The purpose of our approach is to minimize the number of nodes to materialize by avoiding unnecessary materializations. For instance, if it is decided to materialize all sink nodes then it is unnecessary to materialize any intermediate node. In graph theory, a node m *dominates* n if all paths from the source node to n must pass through m. We extend this definition for the case of multiple nodes, and thus we test non-dominance of a set of nodes by checking that, for each node b there is at least one path from b to sink nodes where b is the only selected node. This is formally defined as $\forall b \in B \exists p_i \in Paths_{b,sinks(D)} : |\{\forall node \in p_i : node \in B\}| = 1$. For instance, Figure 5.2b, shows that the green-colored BDS configuration does not satisfy non-dominance, as nodes N_6 and N_7 dominate N_5.

Besides the two essential conditions, it is necessary to maintain a set of visited nodes to check whether a state B has not been already visited, and thus avoid unnecessary expansions in the search space. Figure 5.2c, depicts the valid BDS configurations obtained by applying actions to the BDS configuration {3,4}. Experimenting with the DIF in Figure 5.1, it has been observed that on average eliminating states that do not fulfill such conditions makes a reduction on the search space by 88%. Next, we generate increment and decrement actions for the current node to move vertically by using different index positions of CV. This helps to find the best possible values for given characteristics vector CV for each to-be-materialized node.

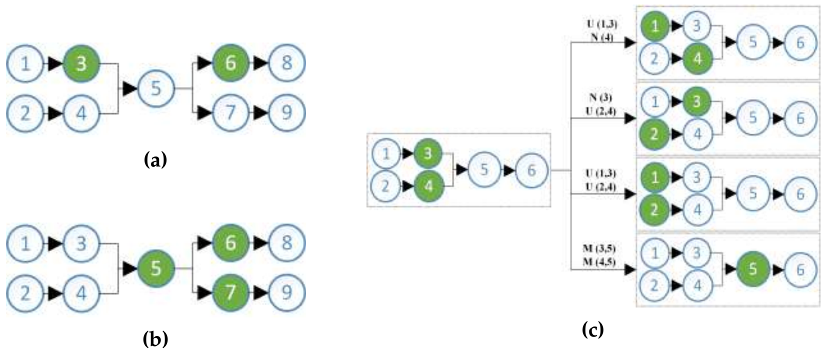


Fig. 5.2: (a) depicts a BDS configuration where answerability is not satisfied, (b) depicts a configuration where non-dominance is not satisfied, and (c) depicts the valid actions for configuration {3,4}

4.2 Initial state

As previously mentioned, the search space contains many local optimum points due to the non-monotonicity of cost functions, therefore the obtained solution might vary depending on the initial state. Three possible initial states have been devised aiming to cover all search space varieties:

- Materialize all source nodes, representing the BDS configuration $B = sources(D)$.
- Materialize all sink nodes, representing the BDS configuration $B = sinks(D)$.

- Random selection of nodes, guaranteeing a valid initial state where answerability and non-dominance are satisfied. Further, for the random selected nodes, we also randomly choose values in all positions of the characteristics vector CV.

Note that the two former are special cases of the third, thus this is the strategy that has been chosen to generate initial states (we provide a more thorough discussion on this in Section 4.4).

4.3 Heuristic

Provided that values of the different objectives lay in very different ranges, and in order to provide a consistent comparison, it is necessary to make use of a non-dimensional utility function normalizing all objectives. There exist a vast number of different normalization strategies [61]. For our purpose, and given the nature of the problem, we make use of the *normalized weighted sum* as utility function, defined as:

$$h(B) = U(\{CF_1, \dots, CF_n\}, B) = \sum_{i=1}^{n} w_i \cdot CF_i^{trans}(B)$$

$CF_i^{trans}(B)$ stands for the evaluation of the transformed cost function for B, being $CF_i(B)$ is evaluation CF_i (see Section 2.2), CF_i^o the *utopia* point (i.e., minimal BDS for CF_i), and CF_i^{max} the maximal BDS:

$$CF_i^{trans}(B) = \frac{CF_i(B) - CF_i^o}{CF_i^{max} - CF_i^o}$$

Such approach yields values between zero and one, depending on the accuracy of both CF_i^o and CF_i^{max} computation. However, it is mostly unattainable to get their exact values, and for that we have to rely on estimations. To achieve this, we compute estimations of utopian BDSs for all cost functions as the union of all minimum nodes for each path from source to sink nodes. Maximum points are obtained by following the similar approach, in this case obtaining maximum nodes for each path from source to sink nodes. Note that, if design goals with constraints are presented, then it is possible to use such constraint value K as maximum point by dismissing the need of estimations.

4.4 Searching the solution space

Local search algorithms consist of the systematic modification of a given state, by means of *Action* functions, in order to derive an improved state. Many complex techniques do exist for such approach (e.g., simulated annealing or genetic algorithms). The intricacy of these algorithms consists of their parametrization, which is also their key performance aspect at the same time . In this chapter, we focus on *hill-climbing*, a non-parametrized search algorithm which can be seen as a local search by always following the path that yields higher heuristic values. Since the used cost functions are highly variable due to their non-monotonicity, hill-climbing might provide different outputs depending on the initial state. In order to overcome such problem, we adopt a variant named *Shotgun hill-climbing* which consists of a hill-climbing with restarts (see Algorithm 14). After certain number of iterations, we can keep the best solution. Such approach of hill-climbing with restarts is surprisingly effective, specially when considering random initial states.

Algorithm 14 Shotgun Hill-Climbing

```
Input $D, i$:                                          ▷ DIF, number of iterations
Output $solution$                                      ▷ Solution BDS configuration
 1: $solution$ = null
 2: do
 3:   $B$ = randomInitialState($D$); $finished = false$
 4:   while !$finished$ do
 5:     $neighbors$ = ResultsFromActions($B$)
 6:     $B'$ = stateWithSmallestHeuristic($neighbors$)
 7:     if h($B'$) < h($B$) then
 8:       $B = B'$
 9:     else
10:       $finished = true$
11:   if h($B$) < h($solution$) then
12:     $solution = B$
13:   $--i$
14: while $i > 0$
15: return $solution$
```

5 Experiments

In this section, we report our experimental findings. Our experiments are performed on an 8-machine cluster. Each machine has a Xeon E5-2630L v2 @2.40GHz CPU, 128GB of main memory and 1TB SATA-3 of hard disk. Each machine runs Hadoop 2.6.0 and Pig 0.15.0[3] on Ubuntu 14.04 (64 bit). We have dedicated one machine for the name node and the remaining seven machines for data nodes. We use TPC-H[4] benchmarking tool to generate datasets and queries. These queries have been converted to Apache Pig which is a procedural language of the big data systems. It has support for user defined functions which is a key feature of modern DIFs. In order to create a complex DIF, we use CoAl [83], which in this case, combines six TPC-H

[3]https://pig.apache.org

[4]http://www.tpc.org/tpch/

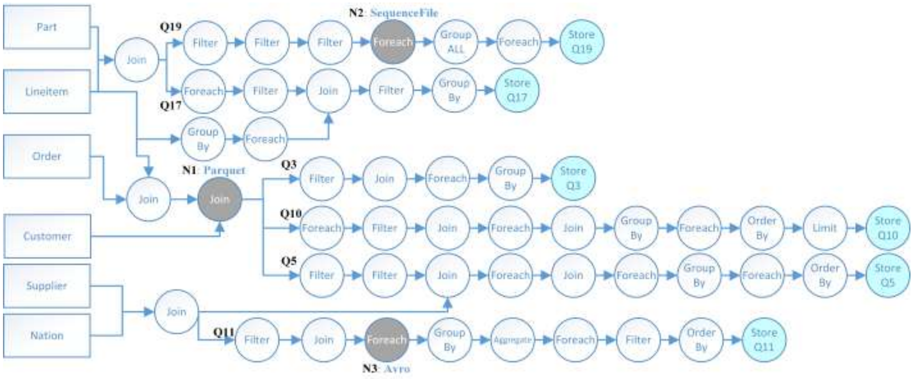


Fig. 5.3: DIF of six TPC-H queries

queries into one integrated DIF as shown in Figure 5.3. The DIF size is chosen with the goal of representing a realistic data pipeline, however being still tractable for validation with an exhaustive search.

5.1 Intermediate results selection evaluation

In this section, we evaluate our approach to validate its two properties: one is convergence and second is the quality of the obtained solutions. We also compare our approach with an existing state of the art solution to show its effectiveness.

Evaluation of convergence and quality of the obtained solutions

The goal of this experiment is to evaluate convergence and quality of the obtained solutions in Algorithm 14. For the sake of experiments, we assign update frequency to each table of TPC-H as shown in Table 5.2. We assume that *supplier* and *nation* tables never update and hence, they have 0 update frequency. Further, *part* and *customer* tables do not update very often and their changes can be applied every 6 hours. That is why, we assign them 4 per day update frequency. Finally, *orders* and *lineitem* tables are frequently updated and they update together whenever there is a new order. We assume that their changes are synchronized every 1 hour and thus, their update frequencies are 24 per day.

To evaluate the convergence of solutions, we systematically executed single shots of our approach (i.e., one iteration) until the number of obtained solutions converged and no new solutions were obtained. With such information, and using the different frequencies, we can provide an estimation of the probability to obtain a solution B_K, formally defined as:

Table Name	UF
Part	4 / day
Lineitem	24 / day
Orders	24 / day
Customer	4 / day
Supplier	0 / day
Nation	0 / day

Table 5.2: Update Frequency (UF) of TPC-H tables

$$P(B_K) = \frac{freq(B_K)}{\sum\limits_{j=1}^{n} freq(B_j)} \tag{5.1}$$

We aim to provide an estimation of the probability of the running Algorithm 14 with i iterations, a solution B_K will be found. To this end, we introduce Equation (5.2) measuring the probability to obtain such solution at position K ($1 \leqslant K \leqslant n$) by running i iterations. It should be noted that B_1 has been confirmed to be the optimal after performing a breadth first search.

$$P(B_K, i) = P(B_K, i-1) \sum_{j=K}^{n} P(B_j, 1) + P(B_K, 1) \sum_{j=K+1}^{n} P(B_j, i-1) \tag{5.2}$$

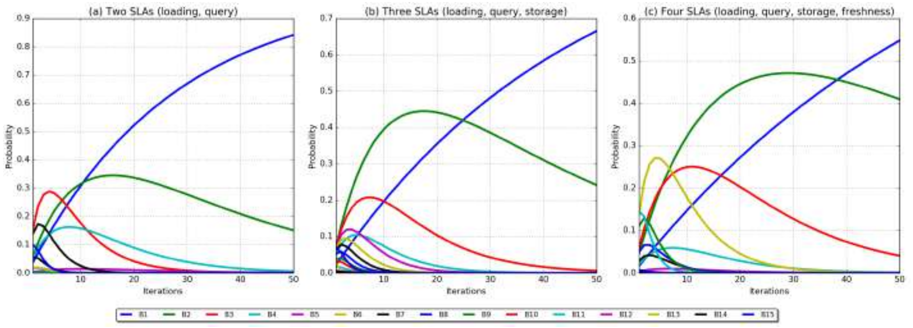


Fig. 5.4: Evolution of probabilities per number of iterations for each different solution

The above mentioned formula is a recursive formula where the base case (i.e., $P(B_K, 1)$) corresponds to the previously defined $P(B_K)$ (i.e., the probability to find solution B_K in one iteration). The rationale behind the recursive case is that after each iteration the one with the lowest heuristic value is kept. Thus, we measure the probability that the solution at position K (i.e., B_K) remains chosen after i iterations. This is achieved by adding (a) the probability that in the previous $i-1$ iterations, B_K was chosen and in the ith iteration an equal or worst solution is chosen (i.e., $P(B_K, i-1) \sum_{j=K}^{n} P(B_j, 1)$); and (b) the probability that in the previous $i-1$ iterations a worst solution was chosen and in the ith iteration B_K is chosen (i.e., $P(B_K, 1) \sum_{j=K+1}^{n} P(B_j, i-1)$). Intuitively, increasing the number of iterations, those with smallest heuristics will have a higher probability to be found regardless of the initial probability being low. With such basis, we can provide an estimation of the evolution of the probability to find a solution B_K by applying the aforementioned formula.

Based on the above mentioned formula, we experiment with the trade-off between different SLAs. We perform evaluation with the following settings: (1) two SLAs (i.e., load time, query time), equally weighted to 50%, (2) three SLAs (i.e., load time, query time, storage space), equally weighted to 33%, and (3) four SLAs (i.e., load time, query time, storage space, freshness), equally weighted to 25%. Our experiments show that the number of iterations to *converge* gradually increases with the number of considered SLAs. As shown in Figure 5.4, our approach takes 11 iterations, 26 iterations, and 39 iterations to converge (i.e., to be certain with a probability of 80%, that the obtained solutions will be one in the top three) for two, three, and four considered SLAs, respectively. In addition, we measure the average execution time of an iteration in different settings. Our approach takes 55.45 seconds, 58.68 seconds, and 183.34 seconds for two, three and four SLAs, respectively. For four SLAs, it takes more time because it has larger search space, due to the conflicting SLAs and the characteristics vector (i.e., refresh frequency). As all considered scenarios follow the same convergence trend as shown in Figure 5.4, let us focus on the most complex scenario involving the trade-off of four SLAs. For four SLAs, we obtained $n = 22$ different solutions across 90 executions. It can be seen that after 39 iterations, it is almost certain (i.e., >90% probability) that the obtained solutions will be one in the top three.

From such results, we conclude that the problem of finding optimal solutions by using hill-climbing indicates the issues with local optimums, known for greedy multi-objective optimization algorithms, and opens the challenge of applying more complex (i.e., parametrized solutions). However, the approach of shotgun hill-climbing, quickly yields near-optimal results after few iterations with high probability.

Comparison with an existing solution

Several intermediate materialization approaches for Big Data systems can be found in the literature, as discussed in Section 6. However, all of them focus on improving the query execution time without considering others SLAs (such as freshness). In order to show the effectiveness of our approach, we compare against the best representative solution (i.e., ReStore).

Query Name	Start Time	Repeated	When to Repeat
Q3	0	Yes	6 / hour
Q5	1	No	-
Q10	2	Yes	2 / hour
Q11	3	Yes	1 / hour
Q17	0	Yes	14 / hour
Q19	2	No	-

Table 5.3: Sample workload based on TPC-H

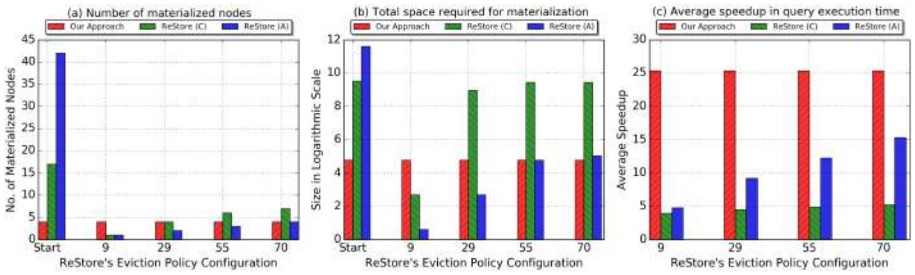


Fig. 5.5: Comparison of our approach, ReStore (C) conservative heuristics and ReStore (A) aggressive heuristics

As mentioned in [35], 80% of queries are repeated in the range of minutes to hours. Thus, we created a sample workload by utilizing six TPC-H queries, based on the aforementioned work. We set four out of six queries as repetitive and two out of six as non-repetitive queries. In addition, we set their query frequencies in the range of minutes to hours as shown in Table 5.3. Moreover, ReStore has a configuration parameter for applying its eviction policies (to delete unused materialized nodes). For experiments, we chose different configuration values such as 9, 29, 55, and 70 in minutes to compare with all the possible behaviors of ReStore.

Figure 5.5 depicts three charts to show different metrics for comparison. In Figure 5.5a, we compare *the total number of materialized nodes*, in Figure 5.5b, we show the total space required, and in Figure 5.5c, we show the average speedup gain in the repetitive queries. When executing the queries for the first time as shown in Figure 5.5a and Figure 5.5b, ReStore materializes each operator matching the heuristics and thus, it materializes more nodes and takes more space. Whereas, our approach uses the cost model to materialize only those nodes which satisfy all the four objectives (i.e., loading time, query time, storage space, and freshness).

When we configured *9 minutes* for applying ReStore's eviction policies, it perceives only Q17 as a repetitive query because it is repeated before applying the eviction policies. Hence, it deletes all the materialized nodes except those

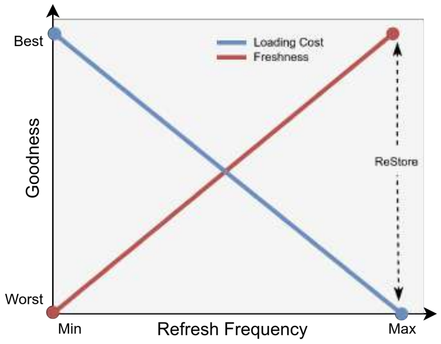


Fig. 5.6: Effect of Refresh Frequency on Loading Cost and Freshness

which are used in Q17. Similarly, when we chose *29 minutes*, now it assumes that Q3 and Q17 are repetitive queries and keeps only their materialized nodes. This decision helps to reduce the occupied space but it also decreases the average speedup as shown in Figure 5.5c. Likewise, when we configured *55 minutes*, ReStore notices three queries (i.e., Q3, Q10, and Q17) as repetitive and keeps only the associated materialized nodes. As a consequence, it deletes all other materialized nodes which also reduces the average speedup. Finally, when we configured *70 minutes*, now it detects all possible repetitive queries and manages to keep all the required materialized nodes. However, still ReStore (C) keeps more nodes and takes more space compared to our approach as shown in Figure 5.5a and Figure 5.5b. On the other hand, ReStore (A) keeps a similar number of materialized nodes to our approach, but provides less average speedup. In general, our approach considers query frequency which helps to choose only the required materialized nodes from the start and provides better speedup than ReStore's heuristics.

In our experiments, we also evaluated our approach based on the characteristics vector (i.e. refresh frequency) to find the trade-off between loading cost and freshness. As shown in Figure 5.6, ReStore does not have support to balance them. It always deletes a materialized node as soon as any of its input source is updated. Thus, it always provides maximum freshness (only affected by the time to materialize new nodes). Consequently, it worsens the loading cost for materialized nodes, which may have highly variable input sources. Oppositely, our approach takes refresh frequency as an input and based on this, it tries to balance loading cost and freshness, by choosing the optimal value for each to-be-materialized node.

From these experiments, we conclude that our approach provides better solutions for materialization than ReStore. In addition, it can consider different SLAs as discussed in Section 3.3, which are not an option in the existing materialization solutions. Moreover, our example shows that we can also accept refresh frequency as a characteristic to find the balance between freshness and loading cost which is not possible in the existing materialized solutions.

6 Related Work

In this chapter, we discuss related work on selections of intermediate results to materialize. Our discussion encompasses four different research lines, namely: *materialized view selection in relational databases*, *materializing intermediate results in BDS*, *multi-query optimization*, and *sharing computation in BDS*. In this section, we separately present their related work.

Materialized Views Selection. The materialized view selection problem has been extensively studied in the context of data warehouses [69]. Most commercial database systems now include a physical design advisor that automatically recommends materialized views by analysing a sample workload of queries (e.g. [8]). According to the survey [69], most view selection methods follow the approach of balancing the trade-off between query processing and view maintenance cost, and dismiss other relevant SLAs (such as freshness, etc.). Whereas, our approach is generic, thus it can consider any type of SLAs that are quantifiable. In addition, our approach takes a characteristics vector (such as refresh frequency) as an input for different SLAs and based on it, it chooses the optimal characteristic value for each to-be-materialized node. This feature is not an option in the existing materialized views solutions.

Materializing Intermediate Results. There exist several approaches that aim to identify potential materialization of intermediate results for future reuse in Big Data systems. ReStore [44] is a heuristic based materialized solution which is implemented for Apache Pig. It has two heuristics (i.e., conservative and aggressive) to decide which operator's output to materialize. Similarly, m2r2 [86] also helps in choosing output of different operators for materialization. However, both solutions are tightly coupled with the technology and in addition, they do not consider different SLAs. In [133], a similar approach to ours is presented, however it focuses on a performance-oriented approach aimed to cloud environments, while we tackle any generic SLAs that can be represented with cost functions.

Multi-query Optimization. Similar to materialized view selection, there have been detailed work done on multi-query optimization in relational databases [136, 139]. Multi-query optimization focuses on improving the performance of concurrent running queries, while our approach focuses on recurrent queries which have redundant parts. They keep the redundant parts in the memory to use them in the currently running queries. On the contrary, we materialize the redundant parts to use them in the future queries.

Sharing Computations. Similar to multi-query optimization, there are few techniques proposed for Big Data systems. MRShare [123] proposes a cost model to group similar queries and optimizes re-usage of redundant parts in Hadoop. Similarly, [161] presents an approach for concurrent running jobs in Hadoop, by reusing existing multi-query optimization techniques. In general, their goal is to avoid redundant work of concurrent running jobs, whereas our work focuses on recurrent jobs.

7 Conclusions

In this chapter, we have presented an approach for the selection of intermediate results from data-intensive flows. We have built upon the general framework for materialized view selection by giving it additionally a multi-objective perspective. Moreover, we have provided a set of three cost functions with its building blocks (i.e., engine-independent statistics and engine-dependent metrics), and a representation of the approach as a state space search problem. Experimental results have showed that our approach is highly efficient in terms of performance, while providing near-optimal results.

Acknowledgements. This research has been partly funded by the European Commission through the Erasmus Mundus Joint Doctorate "Information Technologies for Business Intelligence - Doctoral College" (IT4BI-DC) and the GENESIS project, funded by the Spanish Ministerio de Ciencia e Innovación (num. TIN2016-79269-R).

Chapter 6

Conclusions and Future Directions

Abstract

In this chapter, we summarize the main results of this PhD thesis, presented in Chapters $2-5$*. We additionally, present several interesting future directions arising from this thesis work.*

1 Conclusions

In this thesis dissertation, we have presented our approach for managing the different activities composing the data integration process. The main goal of this thesis is to provide a novel framework for data integration in the context of data-intensive ecosystems. To this end, we introduced our vision for data integration as a sequence of activities encompassing both stewardship and data exploitation perspectives. The former represented by the activities of deployment of an architecture and populating metadata, and the latter by the activities of virtual and materialized integration. In such vision, metadata plays a key role to couple the different activities and offers the possibility to partially automate them. With the focus on the requirements posed by the three Vs (i.e., volume, variety and velocity), we have proposed novel contributions towards each of the integration activities. In what follows, we first summarize the contributions presented in Chapters $2-5$, and finally conclude the thesis.

Chapter 2 studied the problem of defining a semantic-aware data-intensive integration architecture including predefined flows of metadata to support the automation of data exploitation. The chapter begins with the definition of a set of requirements sought for a data-intensive architecture, which were based on the study of the literature and past experience in industrial projects. Driven by such requirements, we studied the literature related to architectural solutions for data-intensive ecosystems. This study lead to conclude that there nowadays exists two main families of architectures that partially cover the sought requirements. Starting from this premise, we proposed *Bolster* a software reference architecture that combines ideas from the two previous families and satisfies all requirements. A distinguishing feature of *Bolster* is that it provides semantic-awareness in data-intensive ecosystems. These are system implementations that have components to simplify data definition and exploitation. In particular, they leverage metadata (i.e., data describing data) to enable (partial) automation of data exploitation and to aid the user in their decision making processes. This simplification supports the differentiation of responsibilities into cohesive roles enhancing data governance. Our contributions were complemented by an exemplary case study illustrating how the components in *Bolster* interact. Finally, we presented a framework to support the instantiation of components from a stack of open source tools, followed by a description of industrial experiences where *Bolster* was successfully adopted.

Chapter 3 concerned the problem of providing new metadata artifacts that allow to represent variety and variability in the sources, while maintaining simplicity in schema mappings leveraging on semantic graphs and their formalisms. To this end, we proposed an integration-oriented ontological vocabulary (i.e., a metadata model) that leverages well-known data integration foundations. With the proposed structure, we are capable of formalizing all integration constructs (i.e., global schema, source schemata and mappings) in a single graph-based structure. Furthermore, we advocate for the adoption of local-as-view schema mappings, which are well-suited for the dynamic setting of interest despite potentially generating performance problems. To semi-automatically deal with evolution, we proposed an algorithm that, upon a new change, updates the graph-based metadata structure. Our evaluation results showed that the proposed approach is capable of handling up to 71.62% of the kind of structural changes occurring in widely used external data sources.

Chapter 4 presented a novel approach to the problem of answering queries using views under semantic heterogeneities as well as data source and schema evolution. Specifically, we leveraged the introduced graph-based metadata model and proposed query rewriting algorithms that transformed an input graph-based query into a set of equivalent queries over the sources such that include or discard semantically heterogeneous data sources, as well as perform implicit aggregations of data. We showed that, under the closed-world assumption, the proposed method yields the set of certain answers as it is sound and complete. Albeit the cost of rewriting a query falls in the NP-hard complexity class, we show by means of an extensive set of experiments that the search space is still manageable in realistic scenarios, even without any pruning or search heuristic. This allows us to efficiently handle query rewriting over a magnitude of hundreds of sources.

Finally, Chapter 5 focused on the problem of selecting the optimal set of intermediate results to be reused from data-intensive flows driven by metadata and service-level agreements. To this end, we revisited the traditional techniques for materialized view selection and extended them for the case of data-intensive flows. Precisely, we adopted multi-objective optimization methods to assess multiple and conflicting objectives (represented by design goals and cost functions). Employing a heuristic-based local search algorithm (i.e., *shotgun hill climbing*), we efficiently explore the search space and find solutions. The experimental results show that, with high probability, the obtained solution turns out to be the optimal with a number of iterations in the order of tenths.

Overall, this thesis proposed contributions to each of the activities in the data integration process (see Figure 1.9 for its graphical representation). We contend that this work is a step towards the definition and implementation of an end-to-end data integration framework in the context of data-intensive ecosystems.

2 Future directions

The proposed framework in this thesis for data-intensive integration opens the door to many interesting future directions to extend our current work.

We foresee the extension of the proposed graph vocabulary with richer semantic annotations (e.g., service-level agreements). These can serve as drivers to perform a heuristic-based query rewriting process, with the goal of reducing the result set (e.g., providing only data of the most recent source). Another relevant line of future work is to explore new classes of queries beyond the proposed sets of edge-restricted patterns, for instance with regular querise. Alternatively, we could relax the necessity of specifying the complete query pattern but provide only the concepts of interest for the analyst. This would require automatically discovering candidate paths to resolve the query. Another interesting line of research would be that of extending the core graph model (e.g., adopt hypergraphs), or any of its constructs (e.g., the mappings graph to represent global-local-as-view schema mappings) to increase their expressiveness (e.g., considering derived information).

As future work, resulting from the overall framework, we devise the coupling of the results of this thesis with Quarry [84], a platform for the automatic deployment of data-intensive flows from input information requirements. Hence, Quarry maps to the materialized activity in the proposed data integration process. Combining our approach for virtual integration with Quarry could lead to the implementation and deployment of the envisioned end-to-end data integration system.

Appendices

Appendix A

Detailed Algorithms for Rewriting Conjunctive Queries

1 Preliminaries

In this appendix we present a detailed version of the different phases involving REWRITECQ.

2 Intra-concept generation

In this subsection we first depict the main algorithm for intra-concept generation (see Algorithm 15). Later we detail each of the performed steps together with examples from the case study introduced in Section 3.1 of Chapter 4.

Algorithm 15 Intra-concept generation

```
Input  $Q_{\mathcal{G}} = \langle \pi, \varphi \rangle$ is a global query, $G$ is the graph of query related concepts
Output  $partialCQsGraph$ is the graph of partial CQs per concept
 1: function INTRACONCEPTGENERATION($\langle \pi, \varphi \rangle, G$)
 2:   $partialCQsGraph \leftarrow \emptyset$                    ▷ Graph with vertices <String,$\overline{CQ}$>
 3:   for $c \in G$.vertexSet() do
 4:     $attsPerWrapper \leftarrow$ Map(key:$W \rightarrow$ val:$\overline{A}$)
 5:     $\overline{F} \leftarrow \langle c,$hasFeature$,?F\rangle(\varphi)$
 6:     if $\overline{F} = \emptyset$ then
 7:       $\overline{W'} \leftarrow \{w \in W \mid c \in patt(\mathcal{M}(w))$.vertexSet()$\}$
 8:       for $w \in \overline{W'}$ do
 9:         $attsPerWrapper[w] \leftarrow \emptyset$
10:     for $f \in \overline{F}$ do
11:       $\overline{W'} \leftarrow \{w \in W \mid \langle c,$hasFeature$,f\rangle \in patt(\mathcal{M}(w))\}$
```

```
12:     for $w \in \overline{W'}$ do
13:         $a \leftarrow \langle ?a, \texttt{sameAs}, f\rangle \wedge \langle w, \texttt{hasAttribute}, ?a\rangle(\mathcal{G})$
14:         $attsPerWrapper[w] \leftarrow attsPerWrapper[w] \cup \{a\}$
15:   $candidateCQs \leftarrow \emptyset$
16:   for $w \in attsPerWrapper$ do
17:     $Q \leftarrow \texttt{CQ}(\langle attsPerWrapper[w], \emptyset, \{w\}\rangle)$
18:     $candidateCQs \cup= Q$
19:   $coveringCQs \leftarrow \emptyset$
20:   while $candidateCQs \neq \emptyset$ do
21:     $Q \leftarrow \textsc{any}(candidateCQs)$
22:     $candidateCQs \setminus= Q$
23:     $\textsc{coveringCQs}(c \times \{\texttt{hasFeature}\} \times \overline{F}, c, Q, candidateCQs, coveringCQs)$
24:   $partialCQsGraph.\texttt{findVertex}(c) \leftarrow coveringCQs$
25: return $partialCQsGraph$
```

Identify queried features (lin. 3-5). The intra-concept generation performs the same logic for each concept c in the graph of query related concepts. We first define a function that maps wrappers to sets of attributes (i.e., the map $attsPerWrapper$). This identifies the attributes that each wrapper covers for the queried features in c. Next, the pattern in line 5 stores the set of features $\overline{F}$ for c specified by the user's query φ.

Example 2.1
Focusing on the concept *InfoMonitor*, the set $\overline{F}$ would contain $\{lagRatio\}$.

Process featureless concepts (lin. 6-9). Next, we process the case where the set of requested features for c is empty. This is commonly the case of specialization concepts, or other intermediate concepts that provide semantics to $\mathcal{G}$ but no data. In this case, we need to generate partial CQs identifying its participating wrappers but with no projected attributes. Hence, we look for the set of wrappers $\overline{W'}$ whose LAV subgraph cover c. Each of this wrappers is added as a new key in the map $attsPerWrapper$ with an empty set of attributes.

Example 2.2
If we focus on the *Monitor* concept (which is featureless), here we would add the keys w_{mon}, w_1, w_2 and w_3 to $attsPerWrapper$, all with empty sets of attributes.

Resolve LAV mappings (lin. 10-14). Alternatively, when concepts have queried features, we iterate on each feature in $\overline{F}$ looking for the set of wrappers

$\overline{W'}$ such that its LAV subgraph covers f. For each $w \in \overline{W'}$, we follow its `sameAs` semantic annotation to identify its respective attribute name a in $\mathcal{S}$. Thus, such attribute name a is added to the set of attributes for w in the map $attsPerWrapper$.

Example 2.3
Going back to *InfoMonitor*, we would add new entries for w_1, w_2 and w_3 all with the set $\{lagRatio\}$.

Generate covering CQs (lin. 16-24). Once we have populated the mapping function $attsPerWrapper$ for a concept c, we can generate combinations of CQs that are covering. Initially, for each wrapper w we generate a new CQ (depicted by the constructor `CQ`$(\langle attsPerWrapper[w], \emptyset, \{w\}\rangle)$), which is added to the set of *candidateCQs* (i.e., the set of candidate queries with one wrapper and the requested attributes they contribute to). While the set of candidate queries is not empty, we pick a query Q from this set, which can be done randomly or in order of most covered features for c (i.e., selecting first the most covering features can ensure more pruning of the set of candidate queries). Next, we systematically find covering CQs calling the recursive algorithm COVERINGCQs (see Algorithm 16).

Example 2.4
The output of Algorithm 15 in the example would be a graph (for the sake of simplicity here we show the vertex set) with the following pairs $\langle c, \overline{CQ}\rangle$ (note that, due to the succinctness of the running example, no combinations of CQ have been generated here).

- *Hour* – $\langle hID, \emptyset, w_{time}\rangle, \langle time, \emptyset, w_1\rangle$
- *InfoMonitor* – $\langle lagRatio, \emptyset, w_1\rangle$, $\langle lagRatio, \emptyset, w_2\rangle$, $\langle lagRatio, \emptyset, w_3\rangle$
- *Monitor* – $\langle \emptyset, \emptyset, w_{mon}\rangle$, $\langle \emptyset, \emptyset, w_1\rangle$, $\langle \emptyset, \emptyset, w_2\rangle$, $\langle \emptyset, \emptyset, w_3\rangle$
- *DataCollector* – $\langle idMon, \emptyset, w_{mon}\rangle$, $\langle idMonitor, \emptyset, w_1\rangle$, $\langle idMonitor, \emptyset, w_2\rangle$, $\langle idMonitor, \emptyset, w_3\rangle$
- *SoftwareApp* – $\langle idApp, name, \emptyset, w_{apps}\rangle$
- *MobileApp* – $\langle \emptyset, \emptyset, w_1\rangle$, $\langle \emptyset, \emptyset, w_2\rangle$, $\langle \emptyset, \emptyset, w_3\rangle$
- *AndroidApp* – $\langle \emptyset, \emptyset, w_1\rangle$

Finding covering CQs

Finding a set of covering CQs for a concept consists of recursively generating valid combinations from the set of candidate CQs. We only consider those combinations such that adding a wrapper contributes with new features, otherwise it will not be minimal. In this case, we look for all valid combinations of both CQs (i.e., ways to join the wrappers) calling method COMBINECQs (see Algorithm 18). We recursively call COVERINGCQs with each legal combination. If any of such combinations is covering, it is added to the set of resulting *coveringCQs*.

Algorithm 16 Get covering CQs

```
Input  $G$ is the graph to check coverage, $c$ is the concept at hand, $currentCQ$ is a CQ, $candidateCQs$
    is a set of CQs, $coveringCQs$ is empty
Output  the set $candidateCQs$ is empty, all potential combinations of covering CQs with respect
    to $G$ are in $coveringCQs$
 1: function COVERINGCQs($G$, $c$, $currentCQ$, $candidateCQs$)
 2:   $coveringCQs \leftarrow \varnothing$
 3:   if COVERING($currentCQ, G$) then
 4:     $coveringCQs \leftarrow currentCQ$
 5:   else if $candidateCQs \neq \varnothing$ then
 6:     for $CQ \in candidateCQs$ do
 7:       $currentFeatures \leftarrow \varnothing$
 8:       for $a \in atts(currentCQ)$ do
 9:         $f \leftarrow \langle a, \texttt{sameAs}, ?f\rangle, \langle c, \texttt{hasFeature}, ?f\rangle$
10:         $currentFeatures \cup= f$
11:       $contributedFeatures \leftarrow \varnothing$
12:       for $a \in atts(currentCQ \oplus CQ)$ do
13:         $f \leftarrow \langle a, \texttt{sameAs}, ?f\rangle, \langle c, \texttt{hasFeature}, ?f\rangle$
14:         $contributedFeatures \cup= f$
15:       if $currentFeatures \subset contributedFeatures$ then
16:         $\overline{CQs} \leftarrow$ COMBINECQs($currentCQ, CQ, c, c$)
17:         $coveringCQs \cup=$ COVERINGCQs($G, c, Q', candidateCQs \setminus CQ$)
18:   return $coveringCQs$
```

3 Inter-concept generation

In this section, we first depict the main algorithm for inter-concept generation (see Algorithm 17). The algorithm systematically chooses an edge e in the graph of partial CQs in order to find valid combinations among the queries in both ends of e using method COMBINECQ. The algorithm finishes when there are no more edges in the graph.

Algorithm 17 Inter-concept generation

```
Input  $partialCQsGraph$ is the graph of partial CQs per concept
Output  $UCQ$ is a set of CQs (i.e., a union of CQs)
 1: function INTERCONCEPTGENERATION($partialCQsGraph$)
 2:   while $partialCQsGraph$.edgeSet() $\neq \varnothing$ do
```

```
3:    $e \leftarrow$ CHOOSEEDGE($partialCQs$)
4:    $s \leftarrow$ SOURCE($e, partialCQsGraph$), $t \leftarrow$ TARGET($e, partialCQsGraph$)
5:    $\overline{CQ_s} \leftarrow s.\_2$, $\overline{CQ_t} \leftarrow t.\_2$
6:    $\overline{UCQ} \leftarrow$ COMBINECQs($CQ_s, CQ_t, s, e, t$)
7:    Remove $s$ and $t$ from $partialCQsGraph$, and add a new vertex $(s+t)$ with the set $\overline{CQ}$
      preserving the graph structure.
8:  return ANYVERTEX($partialCQs$)
```

Example 3.1
Using the input from the previous phase, the output from Algorithm 17 would be a set containing the following expression:

$$\pi_{w_{apps}.name, w_{time}.hId, w_1.lagRatio}(w_1 \times w_{mon} \times w_{apps} \times w_{time})|$$
$$w_1.idMonitor = w_{mon}.idMon \land w_{mon}.app = w_{apps}.idApp \land w_1.time = w_{time}.hId)$$

Combining sets of CQs

Combining two sets of CQs ($\overline{CQ_s}$ and $\overline{CQ_t}$) consists of finding all possible combinations amongst the cartesian product of both sets. However, such sets can be initially pruned by splitting the combination in two phases. First, we can process those queries that share wrappers, because they do not require discovering equi join conditions. In this case the problem is reduced to merging the cartesian product of all minimal queries (lines 3-20). Otherwise, we need to find joins which entails looking for all pairs of queries whose wrappers cover any of the available IDs (i.e., from c_s or c_t) as explained in Section 4.4 (lines 21-39).

Algorithm 18 Combine CQs

```
Input  $\overline{CQ_s}$ and $\overline{CQ_t}$ are sets of CQs, $c_s$ and $c_s$ are the concepts ($e$ their edge) covered respectively
       by $CQ_s$ and $CQ_t$
Output  $\overline{CQ}$ is a set of valid combinations of $\overline{CQ_s}$ and $\overline{CQ_t}$
 1: function COMBINECQs($\overline{CQ_s}$, $\overline{CQ_t}$, $c_s$, $e$, $c_t$)
 2:   $\overline{CQ} \leftarrow \emptyset$                    ▷ This set will hold all valid combinations
 3:   $wrappers_s \leftarrow \emptyset$
 4:   for $CQ \in \overline{CQ_s}$ do
 5:     $wrappers_s \cup= wrap(CQ)$
 6:   $wrappers_t \leftarrow \emptyset$
 7:   for $CQ \in \overline{CQ_t}$ do
 8:     $wrappers_t \cup= wrap(CQ)$
 9:   $wrappersInBothSides \leftarrow wrappers_s \cap wrappers_t$
10:   $\overline{CQ_{s_{shared}}} \leftarrow \emptyset$
11:   for $CQ \in \overline{CQ_s}$ do
12:     if $wrap(CQ) \cap wrappersInBothSides \neq \emptyset$ then
13:       $\overline{CQ_{s_{shared}}} \leftarrow CQ$
14:   $\overline{CQ_{t_{shared}}} \leftarrow \emptyset$
```

```
15: for $CQ \in \overline{CQ_t}$ do
16:   if $wrap(CQ) \cap wrappersInBothSides \neq \varnothing$ then
17:     $\overline{CQ_{t_{shared}}} \leftarrow CQ$
18: for $\langle CQ_s, CQ_t \rangle \in \overline{CQ_{s_{shared}}} \times \overline{CQ_{t_{shared}}}$ do
19:   if MINIMAL($wrap(CQ_s \oplus CQ_t), c_s \times e \times c_t$) then
20:     $\overline{CQ} \cup= CQ_s \oplus CQ_t$
21: $\overline{CQ_{s_{notShared}}} \leftarrow \overline{CQ_s} \setminus \overline{CQ_{s_{shared}}}$
22: $\overline{CQ_{t_{notShared}}} \leftarrow \overline{CQ_t} \setminus \overline{CQ_{t_{shared}}}$
23: $\underline{ID_s} \leftarrow \langle ?ID_s, \texttt{subClass}, \texttt{ID} \rangle \wedge \langle c_s, \texttt{hasFeature}, ?ID_s \rangle(\mathcal{G})$
24: $\underline{ID_t} \leftarrow \langle ?ID_t, \texttt{subClass}, \texttt{ID} \rangle \wedge \langle c_t, \texttt{hasFeature}, ?ID_t \rangle(\mathcal{G})$
25: $\overline{CQ_{s-ID_s}} \leftarrow \varnothing, \overline{CQ_{s-ID_t}} \leftarrow \varnothing$
26: for $CQ \in \overline{CQ_{s_{notShared}}}$ do
27:   for $w \in wrap(CQ)$ do
28:     if $\underline{ID_s} \in patt(\mathcal{M}(w))$ then
29:       $\overline{CQ_{s-ID_s}} \cup= CQ$
30:     if $\underline{ID_t} \in patt(\mathcal{M}(w))$ then
31:       $\overline{CQ_{s-ID_t}} \cup= CQ$
32: $\overline{CQ_{t-ID_s}} \leftarrow \varnothing, \overline{CQ_{t-ID_t}} \leftarrow \varnothing$
33: for $CQ \in \overline{CQ_{t_{notShared}}}$ do
34:   for $w \in wrap(CQ)$ do
35:     if $\underline{ID_s} \in patt(\mathcal{M}(w))$ then
36:       $\overline{CQ_{t-ID_s}} \cup= CQ$
37:     if $\underline{ID_t} \in patt(\mathcal{M}(w))$ then
38:       $\overline{CQ_{t-ID_t}} \cup= CQ$
39: for $\langle CQ_s, CQ_t \rangle \in \overline{CQ_{s-ID_s}} \times \overline{CQ_{t-ID_s}}$ do
40:   $\overline{CQ} \cup=$ FINDJOINS($CQ_s, CQ_t$)
41: for $\langle CQ_s, CQ_t \rangle \in \overline{CQ_{s-ID_t}} \times \overline{CQ_{t-ID_t}}$ do
42:   $\overline{CQ} \cup=$ FINDJOINS($CQ_s, CQ_t$)
43: return $\overline{CQ}$
```

Example 3.2
Taking pairs of connected concepts from the previous phase, for instance the pair *InfoMonitor-Monitor* will detect that all wrappers from *InfoMonitor* have to be joined with w_{mon} from *Monitor*. Other combinations (e.g., two involving w_1 would be directly combined). Using the previous pair of concepts, we would generate three new CQs joining w_{mon} with w_1, w_2 and w_3. Assume another iteration taking such resulting set of queries and trying to join them with the CQs in *SoftwareApp*. All except the query involving w_{mon} would be discarded as there is no way to join them with w_{apps}.

Discovering joins for two CQs

Algorithm 19 Find joins

Input Q_s and Q_t are CQs, ID is an identifier feature
Output CQ is a combination of Q_s and Q_t with equi join conditions

```
1: function FINDJOINS($Q_s$, $Q_t$, $ID$)
```

```
2:  $\overline{W}_s \leftarrow \varnothing$
3:  for $w \in wrap(Q_s)$ do
4:    if $ID \in patt(\mathcal{M}(w))$ then
5:      $\overline{W}_s \cup= w$
6:  $\overline{W}_t \leftarrow \varnothing$
7:  for $w \in wrap(Q_t)$ do
8:    if $ID \in patt(\mathcal{M}(w))$ then
9:      $\overline{W}_t \cup= w$
10: $CQ \leftarrow Q_s \oplus Q_t$
11: for $\langle w_s, w_t \rangle \in \overline{W}_s \times \overline{W}_t$ do
12:   $a_s \leftarrow \langle w_s, \texttt{hasAttribute}, ?a_s \rangle \wedge \langle ?a_s, \texttt{sameAs}, ID \rangle(\mathcal{G})$
13:   $a_t \leftarrow \langle w_t, \texttt{hasAttribute}, ?a_t \rangle \wedge \langle ?a_t, \texttt{sameAs}, ID \rangle(\mathcal{G})$
14:   $CQ \leftarrow CQ \oplus \texttt{Pred}(a_s = a_t)$
15: return $CQ$
```

Appendix B

Extended Experiments for Rewriting Conjunctive Queries

In this appendix we exhaustively report on the experimental results of REWRITECQ. We assume the same experimental setting as described in Section 6.1.

1 Evolution of response time based on wrappers

We first analyse how the response time evolves based on the number of wrappers. To this end we plot the evolution of R for different values of $|W|$, as depicted in Figures B.1, B.2 and B.3 (which respectively correspond to $|F| = 5$, $|F| = 10$ and $|F| = 20$). In all figures, the horizontal axis contain combinations of $|E_W|$, $Frac_W$ and $|W|$, while the vertical axis contain $Frac_Q$, and the colored legend corresponds to different values of $|E_Q|$. Note that there are no charts where $|E_W| > |E_Q|$ (i.e., wrappers cover only the graph associated with the query).

From the previous results we observe that in all cases there is an exponential trend for R as the number of available sources grows. Nonetheless, in many of the situations, we are capable of handling efficiently 128 wrappers. The limitation on number of wrappers occurs as the number of edges covered by the query (i.e., $|E_Q|$) grows . We also observe that the worst case occurs when $Frac_W \approx 50\%$. As expected, this case might generate many combinations of wrappers intra-concept to cover all the requested features. Contrarily, when $Frac_W \ll 50\%$ it is harder to find combinations covering the requested features (as wrappers have a smaller coverage). Similarly, when $Frac_W \gg 50\%$ it is easier that a single wrapper (or a combination of few) covers the requested

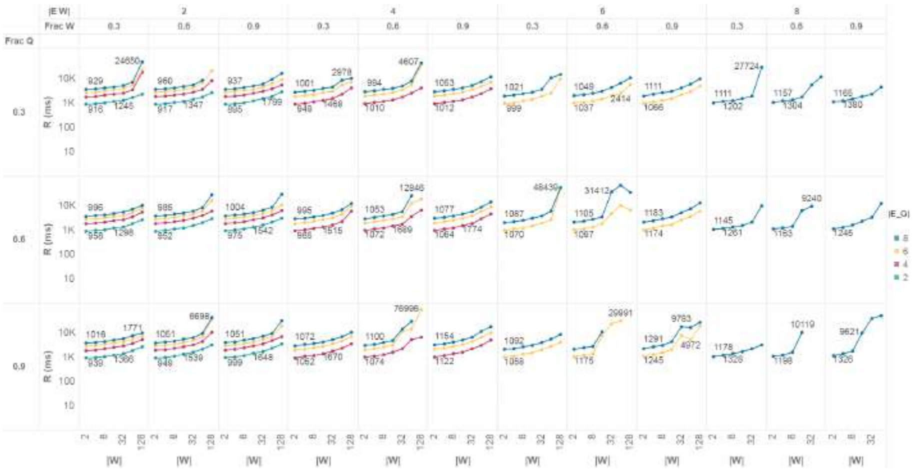


Fig. B.1: Evolution of R w.r.t. $|W|$ for $|F| = 5$

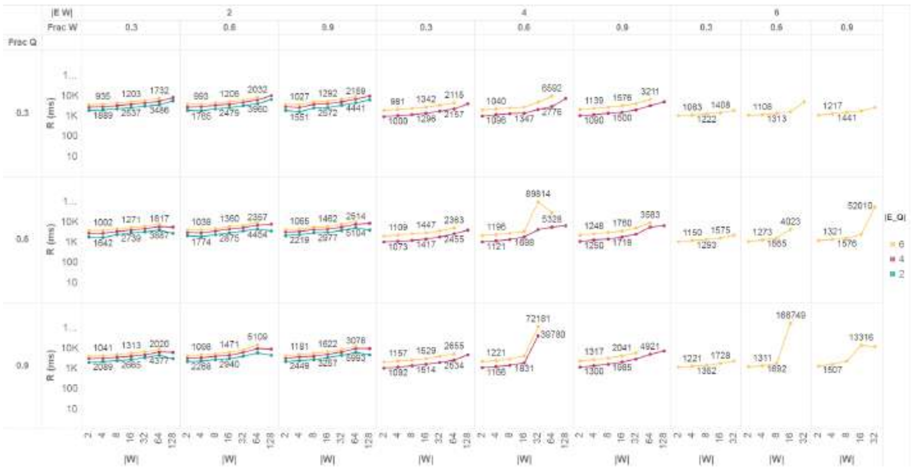


Fig. B.2: Evolution of R w.r.t. $|W|$ for $|F| = 10$

features.

This worst case fact also reflects on the variability among executions, which leads to some inconsistencies in the trends. Note, that in some cases for a higher number of wrappers their response time seems to be reduced. However, with a closer look, we observe high standard deviations between the three executions (if they did not crash due to lack of memory as the number of intermediate results grow). Thus, we conjuncture that with more resources and a large enough sample of executions the exponential trend would clearly be reflected.

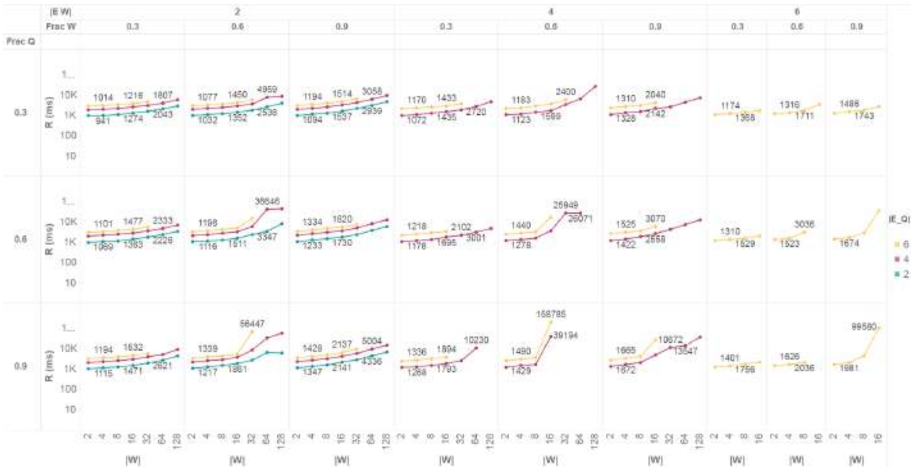


Fig. B.3: Evolution of R w.r.t. $|W|$ for $|F| = 20$

2 Evolution of response time based on edges in the query.

In the second experimental analysis, we are concerned with studying the impact of the size of the query on the time to perform a rewriting. To this end, we plot the evolution of R for different values of $|E_Q|$, as depicted in Figures B.4, B.5 and B.6 (respectively corresponding to $|F| = 5$, $|F| = 10$ and $|F| = 20$). Similarly as before, all figures contain a horizontal axis with the different combinations of $|E_W|$, $|Frac_W|$ and $|E_Q|$, a vertical axis containing different values of $Frac_Q$, and a legend with $|W|$. Note we have filtered out $|W| = 128$ due to the high variability yield in the results, which hindered the visual analysis.

The analytical results show how the cost of rewriting is almost linear regardless of $|E_Q|$ for low values of $|E_W|$. This is not a surprising result, as we can expect a large pruning of candidate solutions in the intra-concept generation phase. As the number of covered edges by wrappers grows, we start seeing variability and a more exponential trend. As a matter of fact, we observe the same trend as in the previous case (i.e., the worst case scenario is when $Frac_W \approx 50\%$).

We also observe that increasing the number of features $|F|$ leads to an exponential trend faster. Nonetheless, still we are capable of efficiently handling large number of sources in a matter of few seconds.

2. Evolution of response time based on edges in the query.

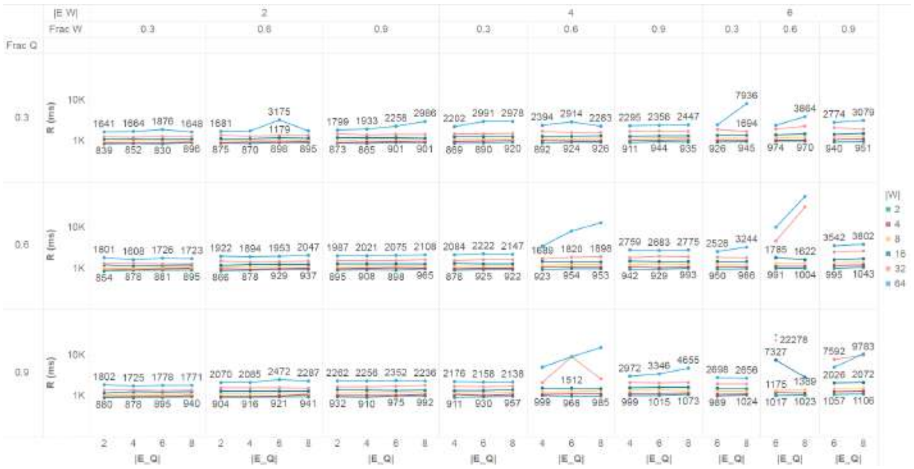


Fig. B.4: Evolution of R w.r.t. $|E_Q|$ for $|F| = 5$

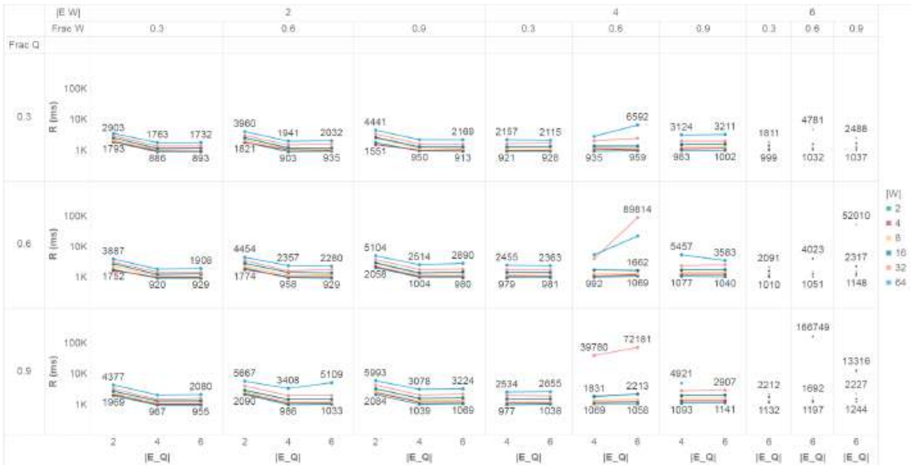


Fig. B.5: Evolution of R w.r.t. $|E_Q|$ for $|F| = 10$

2. Evolution of response time based on edges in the query.

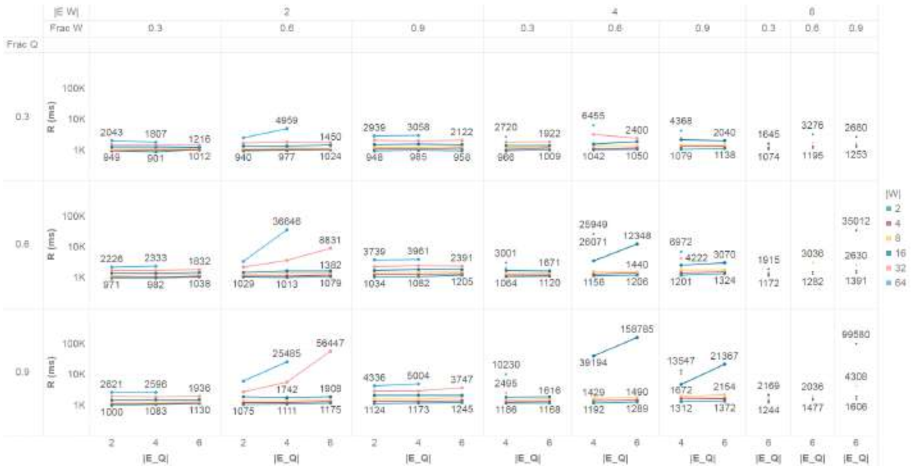


Fig. B.6: Evolution of R w.r.t. $|E_Q|$ for $|F| = 20$

Appendix C

MDM: Governing Evolution in Big Data Ecosystems

This appendix has been published as a paper in the Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018.
The layout of the papers has been revised
DOI: https://doi.org/10.5441/002/edbt.2018.84

Abstract

On-demand integration of multiple data sources is a critical requirement in many Big Data settings. This has been coined as the data variety challenge, which refers to the complexity of dealing with an heterogeneous set of data sources to enable their integrated analysis. In Big Data settings, data sources are commonly represented by external REST APIs, which provide data in their original format and continously apply changes in their structure (i.e., schema). Thus, data analysts face the challenge to integrate such multiple sources, and then continuosly adapt their analytical processes to changes in the schema. To address this challenges, in this chapter, we present the Metadata Management System, shortly MDM*, a tool that supports data stewards and analysts to manage the integration and analysis of multiple heterogeneous sources under schema evolution.* MDM *adopts a vocabulary-based integration-oriented ontology to conceptualize the domain of interest and relies on local-as-view mappings to link it with the sources.* MDM *provides user-friendly mechanisms to manage the ontology and mappings. Finally, a query rewriting algorithm ensures that queries posed to the ontology are correctly resolved to the sources in the presence of multiple schema versions, a transparent process to data analysts. On-site, we will showcase using real-world examples how* MDM *facilitates the management of multiple evolving data sources and enables its integrated analysis.*

1 Introduction

In recent years, a vast number of organizations have adopted data-driven approaches that align their business strategy with advanced data analysis. Such organizations leverage Big Data architectures that support the definition of complex data pipelines in order to process heterogeneous data, from multiple sources, in their original format. External data (i.e., neither generated nor under control of the organization) are commonly ingested from third party data providers (e.g., social networks) via REST APIs with a fixed schema. This requires data analysts to tailor their processes to the imposed schema for each source. A second challenge that data analysts face is the adaptation of such processes upon schema changes (i.e., a release of a new version of the API), a cumbersome task that needs to be manually dealt with. For instance, in the last year Facebook's Graph API[1] released four major versions affecting more than twenty endpoints each, many of them breaking changes. The maintenance of such data analysis processes is critical in scenarios integrating tenths of sources and exploiting them in hundreds of analytical processes, thus its automation is badly needed.

The definition of an integrated view over an heterogeneous set of sources is a challenging task that Semantic Web technologies are well-suited for to overcome the *data variety* challenge [75]. Given the simplicity and flexibility of ontologies, they constitute an ideal tool to define a unified interface (i.e., global vocabulary or schema) for such heterogeneous environments. This family of systems, that perform data integration using ontologies, propose to define a global conceptual schema (i.e., by means of an ontology) over the sources (i.e., by means of mappings) in order to rewrite ontology-mediated queries (OMQs) to the sources. The state of the art approaches for such integration-oriented ontologies are based on generic reasoning algorithms, that rely on certain families of description logics (DLs). Such approaches rewrite an OMQ, first to an expression in first-order logic and then to SQL. This approach, commonly referred as *ontology-based data access* (OBDA) [130], does not consider the management of changes in the sources, and thus such variability in their schema would cause OMQs either crash or return partial results. This issue, which is magnified in Big Data settings, is caused because OBDA approaches represent schema mappings following the *global-as-view* (GAV) approach, where elements of the ontology are characterized in terms of a query over the source schemata. GAV ensures that the process of query rewriting is tractable and yields a first-order logic expression, by just unfolding the queries to the sources, but faulty upon source schema changes [33]. To overcome this issues a desiderata is to adopt the *local-as-view* (LAV) approach. Oppositely to GAV, LAV characterizes elements of the source schemata in

[1] `https://developers.facebook.com/docs/graph-api/changelog`

terms of a query over the ontology, making it inherently more suitable for dynamic environments [82]. LAV flexibility, however, comes at the expense of computational complexity in the query answering process.

To address these challenges, we adopt a vocabulary-based approach for data integration. These approaches are not necessarily restricted to the expressiveness of a DL and its generic reasoning algorithms. Such settings rely on rich metamodels for specific integration tasks, here focused on schema evolution. Under certain constraints when instantiating the metamodel, it is possible to define specific efficient algorithms that resolve LAV mappings without ambiguity. To this end, we created the Metadata Management System, or shortly MDM[2], an end-to-end solution to assist data stewards and data analysts during the Big Data integration lifecycle. Data stewards are provided with mechanisms to semi-automatically integrate new sources and accomodate schema evolution into a global schema. In turn, data analysts have means to pose OMQs to such global schema by making transparent the underlying mechanisms to query the sources with LAV mappings.

MDM implements a vocabulary-based integration-oriented ontology, represented by means of two RDF graphs, specifically the global graph and the source graph [98]. The former representing the domain of interest (also known as domain ontology) and the latter the schema of the sources. The key concepts are *releases*, which represent a new source or changes in existing sources. A relevant element of releases are *wrappers* (from the well-known mediator/wrapper architecture in data integration), the mechanism enabling access to the sources (e.g., an API request or a database query). Upon new releases the schemata of wrappers are extracted and their RDF-based representation stored in the source graph. Afterwards, the data steward is aided on the process of linking such new schemata to the global graph (i.e., define the LAV mapping). Orthogonally, data analysts pose OMQs to the global graph. The current de-facto standard to query ontologies is the SPARQL query language, however to enable non-expert analysts to query the sources MDM offers an interface where OMQs are graphically posed as subgraph patterns of the global graph, which are automatically translated to SPARQL. A specific query rewriting algorithm takes care of how to properly resolve LAV mappings, a process that consists on the discovery of joins amongst wrappers and their attributes, regardless of the number of wrappers per source.

1.1 Motivational use case

As motivational use case, and for the sake of understandability, we will analyse information related to european football teams. This represents the simple use case that will be demoed on-site amongst others with higher complexity

[2] `http://www.essi.upc.edu/~snadal/mdm.html`

(i.e., the SUPERSEDE project). Precisely, we aim to ingest data from four data sources, in the form of REST APIs, respectively providing information about players, teams, leagues and countries. The integrated schema of this scenario is conceptualized in the UML depicted in Figure C.1, which we use as a starting point to provide a high-level representation of the domain of interest, used to generate the ontological knowledge captured in the global graph.

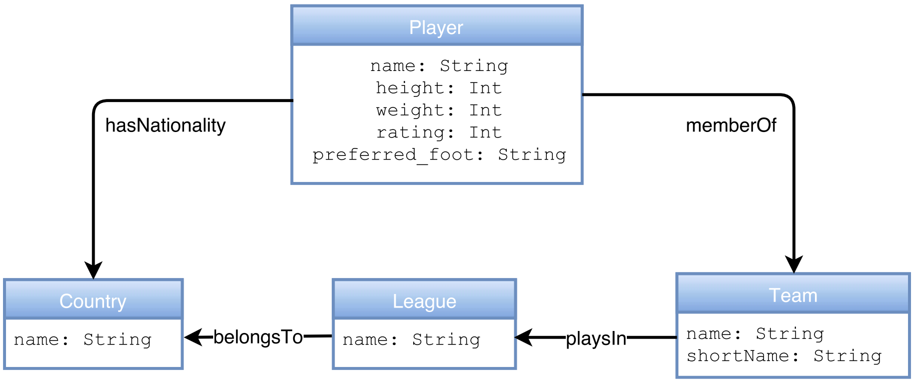


Fig. C.1: UML of the motivational use case

Each of the APIs is independent from each other, and thus they differ in terms of schema and format. Thus, for instance, the *Players API* provides data in JSON format while the *Teams API* in XML. An excerpt of the content provided by such two APIs is depicted in Figure C.2. Next, the goal is to enable data analysts to pose OMQ to the ontology-based representation of the UML diagram (i.e., global graph) by navigating over the classes. Specifically, we aim the sources to be automatically accessed under multiple schema versions. An exemplary query would be, *"who are the players that play in a league of their nationality?"*.

```json
{
  "id": 6176,
  "name": "Lionel Messi",
  "height": 170.18,
  "weight": 159,
  "rating": 94,
  "preferred_foot": "left",
  "team_id": 25
}
```

```xml
<team>
  <id>25</id>
  <name>FC Barcelona</name>
  <shortName>FCB</shortName>
</team>
```

Fig. C.2: Sample data for *Players API* and *Teams API*

Outline In the rest of the chapter, we will introduce the demonstrable features to resolve the motivational and other exploratory queries. We first provide an overview of MDM and then, we present its core features to be demonstrated. Lastly, we outline our on-site presentation, involving the motivational use case and a complex real-world use case.

2 Demonstrable Features

MDM presents an end-to-end solution to integrate and query a set of continuously evolving data sources. Figure C.4 depicts a high-level overview of the approach. Its pillar is the Big Data integration (BDI) ontology [120], the metadata model (i.e., set of design guidelines) that allow data stewards to semantically annotate the integration constructs that enable automating the evolution process and unambiguously resolve query answering.

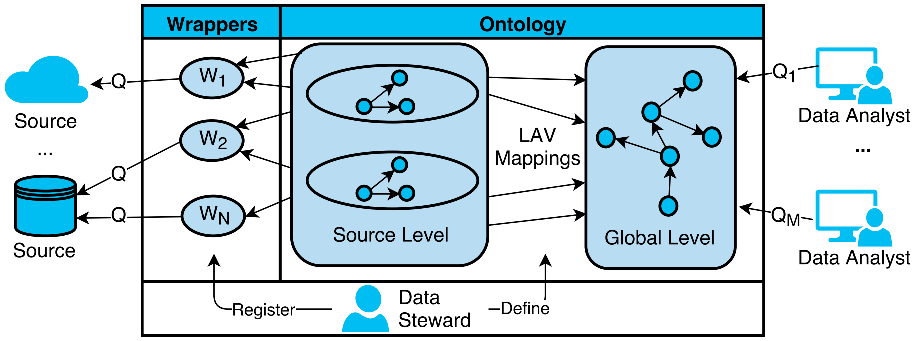


Fig. C.4: High-level overview of our approach

We devise four kinds of interaction with the system, which are in turn the offered functionalities: *(a) definition of the global graph*, where data stewards define the domain of interest for analysts to query; *(b) registration of wrappers*, either in the presence of a new source or the evolution of an existing one; *(c) definition of LAV mappings*, where LAV mappings between the source and the global graphs are defined; and *(d) querying the global graph*, where data analysts pose OMQs to the global graph which are automatically rewritten over the wrappers. In the following subsections, we describe how MDM assists on each of them.

2.1 Definition of the global graph

The global graph, whose elements are prefixed with G, reflects the main domain concepts, relationships among them and features of analysis. To this end, we distinguish between two main constructs *concepts* and *features*.

Concepts (i.e., instances of `G:Concept`) are elements that group features (i.e., `G:Feature`) and do not take concrete values from the sources. Only concepts can be related to each other using any user-defined property, we also allow to define taxonomies for them (i.e., `rdfs:subClassOf`). It is possible to reuse existing vocabularies to semantically annotate the data at the global graph, and thus follow the principles of Linked Data. This, enables data to be self-descriptive as well as it opens the door to publish it on the Web [25]. Furthermore, we restrict features to belong to only one concept.

MDM supports the definition of the global graph avoiding the need to use external ontology modeling tools (e.g., *Protégé*). Figure C.5 depicts an excerpt of the global graph for the demo use case, focusing on the concepts *Player* and *Team*. Like we said, we reuse vocabularies as much as possible, hence the concept *Team* is reused from `http://schema.org/SportsTeam`. When no reuse is possible, we define the example's custom prefix `ex`. As data stewards interact with MDM to define the global graph, the corresponding RDF triples are being generated automatically.

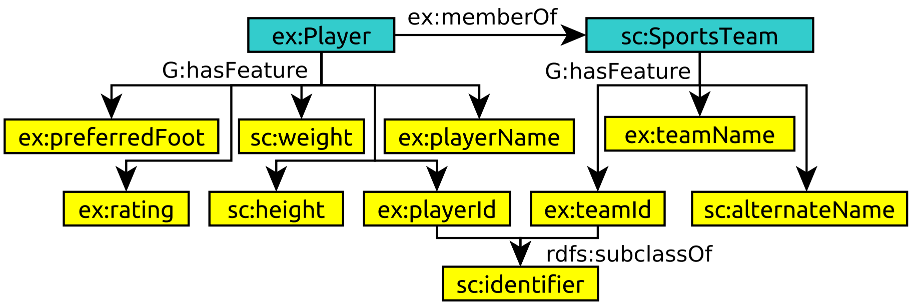


Fig. C.5: Global graph for the motivational use case. Blue and yellow nodes denote concepts and features

2.2 Registration of new data sources

New wrappers are introduced either because we want to consider data from a new data source, or because the schema of an existing source has evolved. Nevertheless, in both cases the procedure to incorporate them to the source level, whose elements are prefixed with `S`, is the same. To this end, we define the data source (i.e., `S:DataSource`) and wrapper (i.e., `S:Wrapper`) metaconcepts. Data stewards must provide the definition of the wrapper, as well as its signature. We work under the assumption that wrappers provide a flat structure in first normal form, thus the signature is an expression of the form $w(a_1, \ldots, a_n)$ where w is the wrapper name and $a_1, \ldots, a_n$ the set of attributes. With such information, MDM extracts the RDF-based representation of the wrapper's schema (i.e., creates elements of type `S:Attribute`) which are incorporated to the existing source level. In the case of a wrapper for an existing data source, MDM will try to reuse as many attributes as possible from the previous wrappers for that data source. However, this is not possible among different data sources as the semantics of attributes might differ. In the case of attributes in the source graph, as they are not meant to be shared, oppositely to features in the global graph, there is no need to reuse external vocabularies.

Figure C.6 depicts an excerpt of the source graph for the sources related to players and teams, the former with a wrapper's signature $w_1(id, pName, height, weight, score, foot, teamId)$ and the latter $w_2(id, name, , shortName)$. Note that, for w_1, some attribute names differ from the data stored in the source (see Figure C.2), this is due to the fact that the query contained in the wrapper might rename (e.g., $foot$) or add new attributes (e.g., $teamId$). The definition of a wrapper (e.g., a MongoDB query, a Spark job, etc.) is out of the scope of MDM and should be carried out by the data steward.

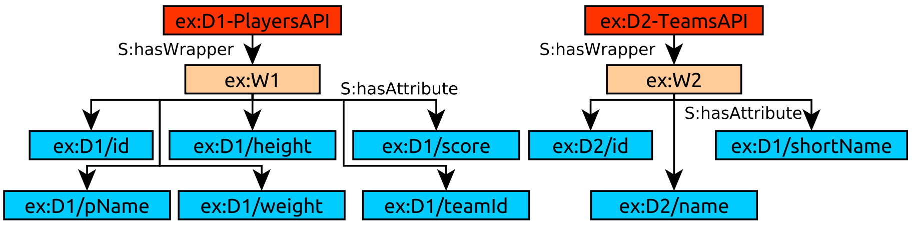


Fig. C.6: Source graph for the motivational use case. Red, orange and blue denote data sources, wrappers and attributes

2.3 Definition of LAV mappings

LAV mappings are encoded as part of the ontology. We represent them as two components, (a) a subgraph of the global graph, one per wrapper, and (b) a function linking attributes from the source graph to features in the global. The former are achieved thanks to RDF named graphs, which allow to identify subsets of other RDF graphs identified by an URI. In this case, the URI will be the one for the wrapper. The latter are achieved via the `owl:sameAs` property. Note that, traditionally, the definition of LAV mappings was a difficult task even for IT people. However, in MDM LAV mappings can be easily asserted through the interface: each wrapper must map to a named graph (i.e., a subset of the global graph), and a set of `owl:sameAs` from attributes to features. The task consists of first selecting a wrapper, and then, with the mouse, drawing a contour around the set of elements in the global graph that this wrapper is populating (including concept relations).

Figure C.7 depicts the LAV mappings for wrappers w_1 and w_2, respectively in red and green. Note the intersection in the concept `sc:SportsTeam` and its identifier, this will be later used when querying in order to enable joining such concepts. However, this joins are only restricted to elements that inherit from `sc:identifier`.

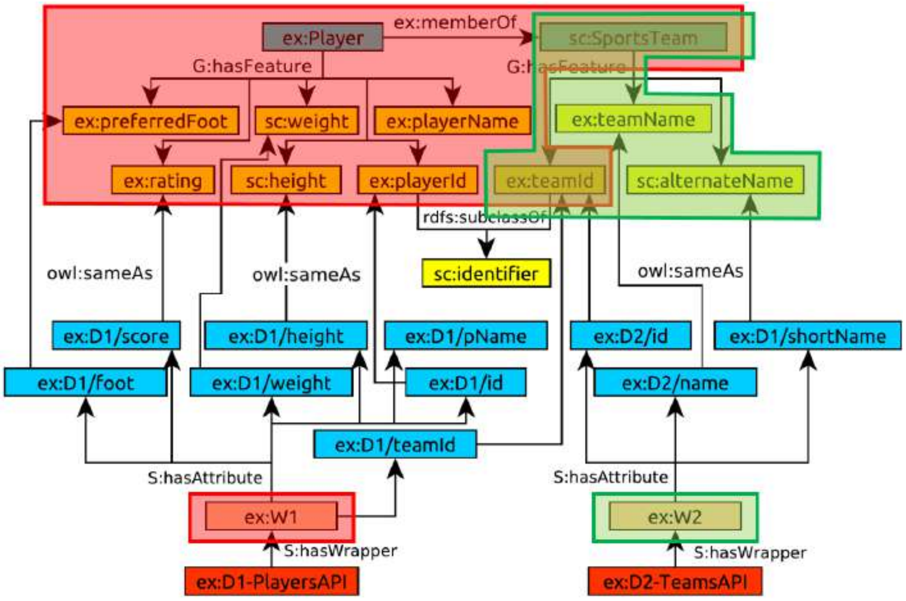
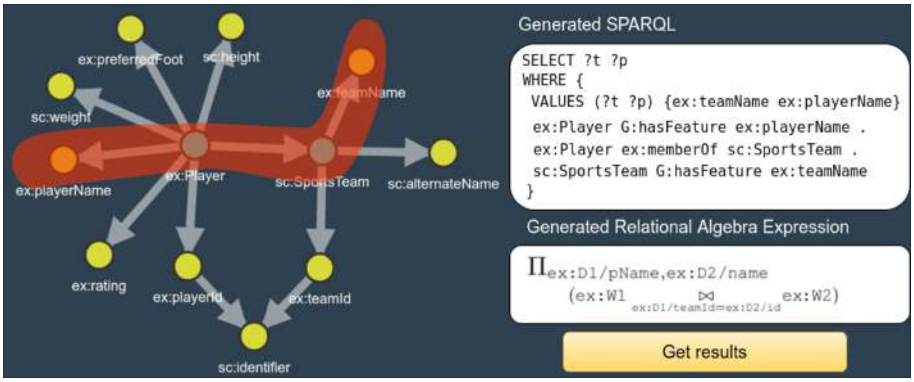


Fig. C.7: LAV mappings for the motivational use case

2.4 Querying the global graph

To overcome the complexity of writing SPARQL queries over the global graph, MDM adopts a graph pattern matching approach to enable non-technical data analysts perform their OMQs. Recall that the *WHERE* clause of a SPARQL query consists of a graph pattern. To this end, the analyst can graphically select a set of nodes of the global graph representing such pattern, we refer to it as a *walk*. Then, a specific query rewriting algorithm takes as input a walk and generates as a result an equivalent union of conjunctive queries over the wrappers resolving the LAV mappings [120]. Such process consists of three phases: *(a) query expansion*, where the walk is automatically expanded to include concept identifiers that have not been explicitly stated; *(b) intra-concept generation*, that generates partial walks per concept indicating how to query the wrappers in order to obtain the requested features for the concept at hand; and *(c) inter-concept generation*, where all partial walks are joined to obtain a union of conjunctive queries.

Using the excerpt of the ontology depicted in Figure C.7, we could graphically pose an OMQ fetching the name of the players and their teams. Figure C.8 shows how such query can be defined in MDM by drawing a contour (in red) around the concepts and features of interest in the global graph. On the right hand side, it is depicted the equivalent SPARQL query, as well as the generated relational algebra expression over the wrappers. Table C.1 depicts a sample of the output resulting of the execution of the query.



```
SELECT ?t ?p
WHERE {
 VALUES (?t ?p) {ex:teamName ex:playerName}
  ex:Player G:hasFeature ex:playerName .
  ex:Player ex:memberOf sc:SportsTeam .
  sc:SportsTeam G:hasFeature ex:teamName
}
```

$$\Pi_{ex:D1/pName,ex:D2/name}(ex:W1 \bowtie_{ex:D1/teamId=ex:D2/id} ex:W2)$$

Fig. C.8: Posing an OMQ in MDM

ex:teamName	ex:playerName
FC Barcelona	Lionel Messi
Bayern Munich	Robert Lewandowski
Manchester United	Zlatan Ibrahimovic

Table C.1: Sample output for the exemplary query.

2.5 Implementation details

MDM has been developed at UPC BarcelonaTech in the context of the SUPERSEDE[3] project using a service-oriented architecture. It is the cornerstorne of the Big Data architecture supporting the project, and a central component of its Semantic Layer [117]. On the frontend, MDM provides the web-based component to assist the management of the Big Data evolution lifecycle. This component is implemented in *JavaScript* and resides in a *Node.JS* web server. The interface makes heavy use of the *D3.js* library to render graphs and enables the user to interact with them. Web interfaces are defined using the *Pug* template engine, and a number of external libraries are additionally used. The backend is implemented as a set of REST APIs defined with *Jersey* for *Java*, thus the frontend interacts with the backend by means of HTTP REST

[3]https://www.supersede.eu

calls. This enables extensibility of the system and a separation of concerns in such big system. The backend makes heavy use of *Jena* to deal with RDF graphs, as well as its persistence engine *Jena TDB*. Additionally, a *MongoDB* document store is responsible of storing the system's metadata. Concerning the execution of queries, the fragment of data provided by wrappers is loaded into temporal SQLite tables in order to execute the federated query.

3 Demonstration overview

In the on-site demonstration, we will present the functionality of MDM relying based on two use cases. First, we will focus on the chapter's motivational scenario, in order to comprehensively show the functionalities offered by MDM. Next we will focus on the SUPERSEDE use case, a real-world scenario of Big Data integration under schema evolution in order to show the full potential and benefits of MDM. We will cover the four possible kinds of interactions with MDM, taking the role of both data steward (definition of the global graph, registration of new wrappers, definition of LAV mappings) and data analyst (querying the global graph). We will showcase how MDM aids on each of the processes, considering as well the input from participants. Precisely, the following scenarios will be covered:

System setup. In the first scenario we will take the role of a data steward that has been given a UML diagram (likewise Figure C.1), and assigned the task of setting up a global schema to enable integrated querying of a disparate set of sources. Thus, we will show how MDM supports the definition of its equivalent global graph (likewise Figure C.5) within the interface. Once finished, we will introduce the four sources (i.e., the players API, teams API, etc.) and a wrapper for each. We will show how MDM automatically extracts the schemata of wrappers to automatically generate the source graph (likewise Figure C.6). Finally, we will show how MDM supports the graphical definition of named graphs, which are the basis for LAV mappings, and thus properly maps the source and global graphs (likewise Figure C.7).

Ontology-mediated queries. With the global graph set up and a set of data sources and wrappers in place, now we can act as data analysts in order to pose OMQs to the system. We will encourage participants to propose their queries of interest, this is possible because MDM presents the global graph and allows to graphically draw a walk around its nodes. This is later automatically translated to its SPARQL form (likewise Figure C.8), and to a relational algebra expression derived from the query rewriting process. MDM presents the execution of the query in tabular form.

Governance of evolution. In Big Data ecosystems, changes in the structure of the data sources will frequently occur. In this scenario, we will release a new version of one of the APIs including breaking changes that would cause the previously defined queries to crash. First, we will showcase how MDM easily supports the inclusion of this new source into the existing global graph and the definition of its LAV mappings. Next, we will execute again the queries that were supposed to crash showing how MDM has adapted the generated relational algebra expressions, where the two schema versions are now fetched and yield correct results.

Acknowledgments This work was partly supported by the H2020 SUPERSEDE project, funded by the EU Information and Communication Technologies Programme under grant agreement no 644018, and the GENESIS project, funded by the Spanish Ministerio de Ciencia e Innovación under project TIN2016-79269-R.

References

[1] D. Abadi, R. Agrawal, A. Ailamaki, M. Balazinska, P. A. Bernstein, M. J. Carey, S. Chaudhuri, J. Dean, A. Doan, M. J. Franklin, J. Gehrke, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, D. Kossmann, S. Madden, S. Mehrotra, T. Milo, J. F. Naughton, R. Ramakrishnan, V. Markl, C. Olston, B. C. Ooi, C. Ré, D. Suciu, M. Stonebraker, T. Walter, and J. Widom. The beckman report on database research. *Commun. ACM*, 59(2):92–99, 2016.

[2] A. Abelló, J. Darmont, L. Etcheverry, M. Golfarelli, J. Mazón, F. Naumann, T. B. Pedersen, S. Rizzi, J. Trujillo, P. Vassiliadis, and G. Vossen. Fusion Cubes: Towards Self-Service Business Intelligence. *IJDWM*, 9(2):66–88, 2013.

[3] A. Abelló, O. Romero, T. B. Pedersen, R. B. Llavori, V. Nebot, M. J. A. Cabo, and A. Simitsis. Using semantic web technologies for exploratory OLAP: A survey. *IEEE Trans. Knowl. Data Eng.*, 27(2):571–588, 2015.

[4] F. N. Afrati and P. G. Kolaitis. Answering aggregate queries in data exchange. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada*, pages 129–138, 2008.

[5] F. N. Afrati and J. D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Trans. Knowl. Data Eng.*, 23(9):1282–1298, 2011.

[6] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[7] D. Agrawal, S. Das, and A. El Abbadi. Big data and cloud computing: current state and future opportunities. In *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 530–533, 2011.

[8] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database tuning advisor for microsoft SQL server 2005. In *VLDB*, pages 1110–1121, 2004.

[9] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. Asterixdb: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

[10] S. Angelov, P. W. P. J. Grefen, and D. Greefhorst. A framework for analysis and design of software reference architectures. *Information & Software Technology*, 54(4):417–431, 2012.

[11] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017.

[12] M. Arenas, P. Barceló, L. Libkin, and F. Murlak. *Foundations of Data Exchange*. Cambridge University Press, 2014.

[13] M. Arenas, L. E. Bertossi, J. Chomicki, X. He, V. Raghavan, and J. P. Spinrad. Scalar aggregation in inconsistent databases. *Theor. Comput. Sci.*, 296(3):405–434, 2003.

[14] A. Artale, D. Calvanese, R. Kontchakov, V. Ryzhikov, and M. Zakharyaschev. Reasoning over extended ER models. In *Conceptual Modeling - ER 2007, 26th International Conference on Conceptual Modeling, Auckland, New Zealand, November 5-9, 2007, Proceedings*, pages 277–292, 2007.

[15] A. Artale, R. Kontchakov, A. Kovtunova, V. Ryzhikov, F. Wolter, and M. Zakharyaschev. First-order rewritability of temporal ontology-mediated queries. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 2706–2712, 2015.

[16] A. Artale, R. Kontchakov, F. Wolter, and M. Zakharyaschev. Temporal description logic for ontology-based data access. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 711–717, 2013.

[17] M. Aufaure. What's up in business intelligence? A contextual and knowledge-based perspective. In *Conceptual Modeling - 32th International Conference, ER 2013, Hong-Kong, China, November 11-13, 2013. Proceedings*, pages 9–18, 2013.

[18] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 1–16, 2002.

[19] P. B. Baeza. Querying graph databases. In *PODS*, pages 175–188. ACM, 2013.

[20] C. Batini, A. Rula, M. Scannapieco, and G. Viscusi. From data quality to big data quality. *J. Database Manag.*, 26(1):60–82, 2015.

[21] R. Bean. Variety, not volume, is driving big data initiatives, March 2016.

[22] B. Behkamal, M. Kahani, and M. K. Akbari. Customizing ISO 9126 Quality Model for Evaluation of B2B Applications. *Information & Software Technology*, 51(3):599–609, 2009.

[23] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, and E. Tsamoura. Benchmarking the chase. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 37–52, 2017.

[24] B. Bilalli, A. Abelló, T. Aluja-Banet, and R. Wrembel. Towards intelligent data analysis: The metadata challenge. In *Proceedings of the International Conference on Internet of Things and Big Data, IoTBD 2016, Rome, Italy, April 23-25, 2016*, pages 331–338, 2016.

[25] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *International Journal on Semantic Web Information Systems*, 5(3):1–22, 2009.

[26] T. Bleifuß, L. Bornemann, T. Johnson, D. V. Kalashnikov, F. Naumann, and D. Srivastava. Exploring change - a new dimension of data analytics. *PVLDB*, 12(2):85–98, 2018.

[27] E. Botoeva, D. Calvanese, B. Cogrel, J. Corman, and G. Xiao. A generalized framework for ontology-based data access. In *AI*IA*, volume 11298 of *Lecture Notes in Computer Science*, pages 166–180. Springer, 2018.

[28] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA.*, page 7, 2000.

[29] A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini. On the expressive power of data integration systems. In *Conceptual Modeling - ER 2002, 21st International Conference on Conceptual Modeling, Tampere, Finland, October 7-11, 2002, Proceedings*, pages 338–350, 2002.

[30] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *J. Autom. Reasoning*, 39(3):385–429, 2007.

[31] D. Calvanese, E. Kharlamov, W. Nutt, and C. Thorne. Aggregate queries over ontologies. In *Proceedings of the 2nd International Workshop on Ontologies and Information Systems for the Semantic Web, ONISW 2008, Napa Valley, California, USA, October 30, 2008*, pages 97–104, 2008.

[32] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[33] L. Caruccio, G. Polese, and G. Tortora. Synchronization of queries and views upon schema evolutions: A survey. *ACM Trans. Database Syst.*, 41(2):9:1–9:41, 2016.

[34] C. L. P. Chen and C. Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Inf. Sci.*, 275:314–347, 2014.

[35] Y. Chen, S. Alspaugh, and R. H. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. *PVLDB*, 5(12):1802–1813, 2012.

[36] S. Das, S. Sundara, and R. Cyganiak. R2RML: RDB to RDF mapping language. W3C recommendation, W3C, Sept. 2012. https://www.w3.org/TR/r2rml/.

[37] N. Daswani, H. Garcia-Molina, and B. Yang. Open problems in data-sharing peer-to-peer systems. In *ICDT*, volume 2572 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2003.

[38] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[39] A. Doan, A. Y. Halevy, and Z. G. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.

[40] X. L. Dong and D. Srivastava. *Big Data Integration*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2015.

[41] P. Downey. XML Schema Patterns for Common Data Structures. *W3.org*, 2005.

[42] O. M. Duschka, M. R. Genesereth, and A. Y. Levy. Recursive query plans for data integration. *J. Log. Program.*, 43(1):49–73, 2000.

[43] J. O. e Sá, C. Martins, and P. Simões. Big data in cloud: A data architecture. In *New Contributions in Information Systems and Technologies - Volume 1 [WorldCIST'15, Azores, Portugal, April 1-3, 2015].*, pages 723–732, 2015.

[44] I. Elghandour and A. Aboulnaga. ReStore: Reusing Results of MapReduce Jobs. *PVLDB*, 5(6):586–597, 2012.

[45] D. Esteban. Interoperability and standards in the european data economy - report on EC workshop. *European Commission*, 2016.

[46] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[47] R. C. Fernandez, P. Pietzuch, J. Kreps, N. Narkhede, J. Rao, J. Koshy, D. Lin, C. Riccomini, and G. Wang. Liquid: Unifying nearline and offline big data integration. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[48] G. Fox and W. Chang. NIST Big Data Interoperability Framework: Volume 3, Use Case and General Requirements. *NIST Special Publication*, (1500-3), 2015.

[49] M. Friedman, A. Y. Levy, and T. D. Millstein. Navigational plans for data integration. In *Proceedings of the IJCAI-99 Workshop on Intelligent Information Integration, Held on July 31, 1999 in conjunction with the Sixteenth International Joint Conference on Artificial Intelligence City Conference Center, Stockholm, Sweden*, 1999.

[50] M. Galster and P. Avgeriou. Empirically-grounded reference architectures: a proposal. In *7th International Conference on the Quality of Software Architectures, QoSA 2011 and 2nd International Symposium on Architecting Critical Systems, ISARCS 2011. Boulder, CO, USA, June 20-24, 2011, Proceedings*, pages 153–158, 2011.

[51] A. Gani, A. Siddiqa, S. Shamshirband, and F. Hanum. A survey on indexing techniques for big data: taxonomy and performance evaluation. *Knowl. Inf. Syst.*, 46(2):241–284, 2016.

[52] S. García, O. Romero, and R. Raventós. DSS from an RE perspective: A systematic mapping. *Journal of Systems and Software*, 117:488–507, 2016.

[53] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[54] B. Geerdink. A reference architecture for big data solutions - introducing a model to perform predictive analytics using big data technology. *IJBDI*, 2(4):236–249, 2015.

[55] A. Giacometti, P. Marcel, and E. Negre. A framework for recommending OLAP queries. In *DOLAP 2008, ACM 11th International Workshop on Data Warehousing and OLAP, Napa Valley, California, USA, October 30, 2008, Proceedings*, pages 73–80, 2008.

[56] B. Golshan, A. Y. Halevy, G. A. Mihaila, and W. Tan. Data integration: After the teenage years. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 101–106, 2017.

[57] I. Gorton and J. Klein. Distribution, data, deployment: Software architecture convergence in big data systems. *IEEE Software*, 32(3):78–85, 2015.

[58] N. W. Grady, M. Underwood, A. Roy, and W. L. Chang. Big data: Challenges, practices and technologies: NIST big data public working group workshop at IEEE big data 2014. In *2014 IEEE International Conference on Big Data, Big Data 2014, Washington, DC, USA, October 27-30, 2014*, pages 11–15, 2014.

[59] B. C. Grau, A. Fokoue, B. Motik, Z. Wu, and I. Horrocks. OWL 2 web ontology language profiles (second edition). W3C recommendation, W3C, Dec. 2012. http://www.w3.org/TR/2012/REC-owl2-profiles-20121211.

[60] J. Gray, D. T. Liu, M. A. Nieto-Santisteban, A. S. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34(4):34–41, 2005.

[61] O. Grodzevich and O. Romanko. Normalization and other topics in multi-objective optimization. In *FMIPW*, pages 42–56, 2006.

[62] A. Grosskurth and M. W. Godfrey. A reference architecture for web browsers. In *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*, pages 661–664, 2005.

[63] M. Grover, T. Malaska, J. Seidman, and G. Shapira. *Hadoop Application Architectures*. O'Reilly Media, Inc., 2015.

[64] T. Gruber. Ontology. In *Encyclopedia of Database Systems*, pages 1963–1965. 2009.

[65] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Comp. Syst.*, 29(7):1645–1660, 2013.

[66] K. Guo, W. Pan, M. Lu, X. Zhou, and J. Ma. An effective and economical architecture for semantic-based heterogeneous multimedia big data retrieval. *Journal of Systems and Software*, 102:207–216, 2015.

[67] H. Gupta and I. S. Mumick. Selection of Views to Materialize in a Data Warehouse. *IEEE Trans. Knowl. Data Eng.*, 17(1):24–43, 2005.

[68] R. Halasipuram, P. M. Deshpande, and S. Padmanabhan. Determining essential statistics for cost based optimization of an ETL workflow. In *EDBT*, pages 307–318. OpenProceedings.org, 2014.

[69] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[70] A. Y. Halevy, N. Ashish, D. Bitton, M. J. Carey, D. Draper, J. Pollock, A. Rosenthal, and V. Sikka. Enterprise information integration: successes, challenges and controversies. In *SIGMOD Conference*, pages 778–787. ACM, 2005.

[71] A. Y. Halevy, A. Rajaraman, and J. J. Ordille. Data integration: The teenage years. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 9–16, 2006.

[72] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing Data Cubes Efficiently. In *SIGMOD*, pages 205–216, 1996.

[73] M. J. Harry and R. R. Schroeder. *Six Sigma: The breakthrough management strategy revolutionizing the world's top corporations*. Broadway Business, 2005.

[74] V. Herrero, A. Abelló, and O. Romero. NOSQL design for analytical workloads: Variability matters. In *Conceptual Modeling - 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings*, pages 50–64, 2016.

[75] I. Horrocks, M. Giese, E. Kharlamov, and A. Waaler. Using semantic technology to tame the data variety challenge. *IEEE Internet Computing*, 20(6):62–66, 2016.

[76] F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in data flow optimization. *PVLDB*, 5(11):1256–1267, 2012.

[77] M. Interlandi, K. Shah, S. D. Tetali, M. Gulzar, S. Yoo, M. Kim, T. D. Millstein, and T. Condie. Titian: Data provenance support in spark. *PVLDB*, 9(3):216–227, 2015.

[78] B. Ionescu, D. Ionescu, C. Gadea, B. Solomon, and M. Trifan. An architecture and methods for big data analysis. In *Soft Computing Applications - Proceedings of the 6th International Workshop Soft Computing Applications, SOFA 2014, Volume 1, Timisoara, Romania, 24-26 July 2014*, pages 491–514, 2014.

[79] ISO. IEC25010: 2011 Systems and software engineering–Systems and software Quality Requirements and Evaluation (SQuaRE)–System and software quality models. 2011.

[80] ISO. ISO/IEC 9075-11:2016: Information technology – Database languages – SQL – Part 11: Information and definition schemas (SQL/Schemata). 2016.

[81] H. V. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, and C. Shahabi. Big data and its technical challenges. *Commun. ACM*, 57(7):86–94, 2014.

[82] P. Jovanovic, O. Romero, and A. Abelló. A Unified View of Data-Intensive Flows in Business Intelligence Systems: A Survey. *T. Large-Scale Data- and Knowledge-Centered Systems*, 29:66–107, 2016.

[83] P. Jovanovic, O. Romero, A. Simitsis, and A. Abelló. Incremental Consolidation of Data-Intensive Multi-Flows. *IEEE Trans. Knowl. Data Eng.*, 28(5):1203–1216, 2016.

[84] P. Jovanovic, O. Romero, A. Simitsis, A. Abelló, H. Candón, and S. Nadal. Quarry: Digging up the gems of your data treasury. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, pages 549–552, 2015.

[85] P. Jovanovic, A. Simitsis, and K. Wilkinson. Engine independence for logical analytic flows. In *ICDE*, pages 1060–1071. IEEE Computer Society, 2014.

[86] V. Kalavri, H. Shang, and V. Vlassov. m2r2: A framework for results materialization and reuse in high-level dataflow systems for big data. In *CSE*, pages 894–901, 2013.

[87] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. *IEEE Trans. Vis. Comput. Graph.*, 18(12):2917–2926, 2012.

[88] R. M. Karp. On the computational complexity of combinatorial problems. *Networks*, 5(1):45–68, 1975.

[89] C. M. Keet and E. A. N. Ongoma. Temporal attributes: Status and subsumption. In *11th Asia-Pacific Conference on Conceptual Modelling, APCCM 2015, Sydney, Australia, January 2015*, pages 61–70, 2015.

[90] J. Kephart, D. Chess, C. Boutilier, R. Das, J. O. Kephart, and W. E. Walsh. An architectural blueprint for autonomic computing. 2007.

[91] V. Khatri and C. V. Brown. Designing data governance. *Commun. ACM*, 53(1):148–152, 2010.

[92] R. Kimball and M. Ross. *The data warehouse toolkit: the complete guide to dimensional modeling, 2nd Edition*. Wiley, 2002.

[93] T. Kirk, A. Y. Levy, Y. Sagiv, D. Srivastava, et al. The information manifold. In *Proceedings of the AAAI 1995 Spring Symp. on Information Gathering from Heterogeneous, Distributed Enviroments*, volume 7, pages 85–91, 1995.

[94] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering, 2007.

[95] J. Kontio. A case study in applying a systematic method for COTS selection. In *ICSE 1996*, pages 201–209, 1996.

[96] E. V. Kostylev and J. L. Reutter. Complexity of answering counting aggregate queries over dl-lite. *J. Web Sem.*, 33:94–111, 2015.

[97] J. Kroß, A. Brunnert, C. Prehofer, T. A. Runkler, and H. Krcmar. Stream processing on demand for lambda architectures. In *Computer Performance Engineering - 12th European Workshop, EPEW 2015, Madrid, Spain, August 31 - September 1, 2015, Proceedings*, pages 243–257, 2015.

[98] M. Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 233–246, 2002.

[99] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California, USA*, pages 95–104, 1995.

[100] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *22th International Conference on Very Large Data Bases (VLDB)*, pages 251–262, 1996.

[101] J. Li, Y. Xiong, X. Liu, and L. Zhang. How Does Web Service API Evolution Affect Clients? In *2013 IEEE 20th International Conference on Web Services, Santa Clara, CA, USA, June 28 - July 3, 2013*, pages 300–307, 2013.

[102] W. Litwin. From database systems to multidatabase systems: Why and how. In *BNCOD*, pages 161–188. Cambridge University Press, 1988.

[103] F. Liu, J. Tong, J. Mao, R. Bohn, J. Messina, L. Badger, and D. Leaf. *NIST Cloud Computing Reference Architecture: Recommendations of the National Institute of Standards and Technology*. 2012.

[104] A. Löser, F. Hueske, and V. Markl. Situational business intelligence. In *Business Intelligence for the Real-Time Enterprise - Second International Workshop, BIRTE 2008, Auckland, New Zealand, August 24, 2008, Revised Selected Papers*, pages 1–11, 2008.

[105] C. Lutz, F. Wolter, and M. Zakharyaschev. Temporal description logics: A survey. In *15th International Symposium on Temporal Representation and Reasoning, TIME 2008, Université du Québec à Montréal, Canada, 16-18 June 2008*, pages 3–14, 2008.

[106] N. H. Madhavji, A. V. Miranskyy, and K. Kontogiannis. Big picture of big data software engineering: With example research challenges. In *1st IEEE/ACM International Workshop on Big Data Software Engineering, BIGDSE 2015, Florence, Italy, May 23, 2015*, pages 11–14, 2015.

[107] P. Manousis, P. Vassiliadis, and G. Papastefanatos. Impact analysis and policy-conforming rewriting of evolving data-intensive ecosystems. *J. Data Semantics*, 4(4):231–267, 2015.

[108] R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004.

[109] S. Martínez-Fernández, C. P. Ayala, X. Franch, and E. Y. Nakagawa. A survey on the benefits and drawbacks of autosar. In *Proceedings of the First International Workshop on Automotive Software Architecture*, WASA '15, pages 19–26, New York, NY, USA, 2015. ACM.

[110] M. A. Martínez-Prieto, C. E. Cuesta, M. Arias, and J. D. Fernández. The solid architecture for real-time management of big semantic data. *Future Generation Comp. Syst.*, 47:62–79, 2015.

[111] N. Marz and J. Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2015.

[112] P. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 227–238, 2003.

[113] E. Meijer and G. M. Bierman. A co-relational model of data for large shared data banks. *Commun. ACM*, 54(4):49–58, 2011.

[114] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. T. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. G. Stephan, and J. V. den Bussche. The open provenance model core specification (v1.1). *Future Generation Comp. Syst.*, 27(6):743–756, 2011.

[115] R. F. Munir, O. Romero, A. Abelló, B. Bilalli, M. Thiele, and W. Lehner. Resilientstore: A heuristic-based data format selector for intermediate results. In *Model and Data Engineering - 6th International Conference, MEDI 2016, Almería, Spain, September 21-23, 2016, Proceedings*, pages 42–56, 2016.

[116] S. Nadal, V. Herrero, O. Romero, A. Abelló, X. Franch, and S. Vansummeren. Details on Bolster - State of the Art (`www.essi.upc.edu/~snadal/Bolster_SLR.html`), 2016.

[117] S. Nadal, V. Herrero, O. Romero, A. Abelló, X. Franch, S. Vansummeren, and D. Valerio. A software reference architecture for semantic-aware big data systems. *Information & Software Technology*, 90:75–92, 2017.

[118] S. Nadal, O. Romero, A. Abelló, P. Vassiliadis, and S. Vansummeren. Wordpress Evolution Analysis `www.essi.upc.edu/~snadal/wordpress_evol.txt`, 2016.

[119] S. Nadal, O. Romero, A. Abelló, P. Vassiliadis, and S. Vansummeren. An integration-oriented ontology to govern evolution in big data ecosystems. In *Proceedings of the Workshops of the EDBT/ICDT 2017 Joint Conference (EDBT/ICDT 2017), Venice, Italy, March 21-24, 2017.*, 2017.

[120] S. Nadal, O. Romero, A. Abelló, P. Vassiliadis, and S. Vansummeren. An integration-oriented ontology to govern evolution in big data ecosystems. *Inf. Syst.*, 79:3–19, 2019.

[121] T. Nguyen, S. Bimonte, L. d'Orazio, and J. Darmont. Cost models for view materialization in the cloud. In *EDBT/ICDT Workshops*, pages 47–54, 2012.

[122] N. F. Noy and M. C. A. Klein. Ontology evolution: Not the same as schema evolution. *Knowl. Inf. Syst.*, 6(4):428–440, 2004.

[123] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: Sharing Across Multiple Queries in MapReduce. *PVLDB*, 3(1):494–505, 2010.

[124] C. Ordonez. Statistical model computation with udfs. *IEEE Trans. Knowl. Data Eng.*, 22(12):1752–1765, 2010.

[125] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[126] P. Pääkkönen and D. Pakkala. Reference architecture and classification of technologies, products and services for big data systems. *Big Data Research*, 2(4):166–186, 2015.

[127] P. Panov, S. Dzeroski, and L. N. Soldatova. Ontodm: An ontology of data mining. In *Workshops Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008), December 15-19, 2008, Pisa, Italy*, pages 752–760, 2008.

[128] C. Pautasso, O. Zimmermann, and F. Leymann. Restful Web Services vs. "Big" Web Services: Making The Right Architectural Decision. In *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, pages 805–814, 2008.

[129] D. L. Phuoc, H. Q. Nguyen-Mau, J. X. Parreira, and M. Hauswirth. A middleware framework for scalable management of linked streams. *J. Web Sem.*, 16:42–51, 2012.

[130] A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *Journal on Data Semantics*, 10:133–173, 2008.

[131] R. Pottinger and A. Y. Halevy. Minicon: A scalable algorithm for answering queries using views. *VLDB Journal*, 10(2-3):182–198, 2001.

[132] S. Qanbari, S. M. Zadeh, S. Vedaei, and S. Dustdar. Cloudman: A platform for portable cloud manufacturing services. In *2014 IEEE International Conference on Big Data, Big Data 2014, Washington, DC, USA, October 27-30, 2014*, pages 1006–1014, 2014.

[133] W. Qu and S. Dessloch. A Real-time Materialized View Approach for Analytic Flows in Hybrid Cloud Environments. *Datenbank-Spektrum*, 14(2):97–106, 2014.

[134] M. T. Roth and P. M. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 266–275, 1997.

[135] A. Roukh, L. Bellatreche, A. Boukorca, and S. Bouarar. Eco-DMW: Eco-Design Methodology for Data warehouses. In *DOLAP*, pages 1–10, 2015.

[136] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD Conference*, pages 249–260. ACM, 2000.

[137] P. Russom. Big data analytics. *TDWI Best Practices Report, Fourth Quarter*, page 6, 2011.

[138] K. Sattler. Data quality dimensions. In *Encyclopedia of Database Systems*, pages 612–615. Springer US, 2009.

[139] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[140] R. Sharda, D. A. Asamoah, and N. Ponna. Business analytics: Research and teaching perspectives. In *Proceedings of the ITI 2013 35th International Conference on Information Technology Interfaces, Cavtat / Dubrovnik, Croatia, June 24-27, 2013*, pages 19–27, 2013.

[141] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, 1990.

[142] A. Simitsis and K. Wilkinson. Revisiting ETL Benchmarking: The Case for Hybrid Flows. In *TPCTC*, pages 75–91, 2012.

[143] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Qox-driven ETL design: reducing the cost of ETL consulting engagements. In *SIGMOD*, pages 953–960, 2009.

[144] I. Skoulis, P. Vassiliadis, and A. V. Zarras. Growing Up with Stability: How Open-source Relational Databases Evolve. *Inf. Syst.*, 53:363–385, 2015.

[145] J. Song, C. Guo, Z. Wang, Y. Zhang, G. Yu, and J. Pierson. Haolap: A hadoop based OLAP system for big data. *Journal of Systems and Software*, 102:167–181, 2015.

[146] J. Stefanowski, K. Krawiec, and R. Wrembel. Exploring complex and big data. *Applied Mathematics and Computer Science*, 27(4):669–679, 2017.

[147] M. Stonebraker. What does 'big data' mean? *Communications of the ACM, BLOG@ ACM*, 2012.

[148] M. Stonebraker. Why the 'data lake' is really a 'data swamp'. *Communications of the ACM, BLOG@ ACM*, 2014.

[149] F. M. Suchanek, S. Abiteboul, and P. Senellart. PARIS: Probabilistic Alignment of Relations, Instances, and Schema. *PVLDB*, 5(3):157–168, 2011.

[150] R. Tan, R. Chirkova, V. Gadepally, and T. G. Mattson. Enabling query processing across heterogeneous data models: A survey. In *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*, pages 3211–3220, 2017.

[151] H. J. ter Horst. Extending the RDFS entailment lemma. In *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima, Japan, November 7-11, 2004. Proceedings*, pages 77–91, 2004.

[152] I. Terrizzano, P. M. Schwarz, M. Roth, and J. E. Colino. Data wrangling: The challenging yourney from the wild to the lake. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[153] D. Theodoratos and M. Bouzeghoub. A general framework for the view selection problem for data warehouse design and evolution. In *Third ACM International Workshop on Data Warehousing and OLAP (DOLAP 2000), Washington, DC, USA, November 10, 2000*, pages 1–8, 2000.

[154] D. Theodoratos and T. K. Sellis. Data warehouse configuration. In *VLDB*, pages 126–135. Morgan Kaufmann, 1997.

[155] D. Theodoratos and T. K. Sellis. Dynamic data warehouse design. In *DaWaK*, volume 1676 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 1999.

[156] C.-W. Tsai, C.-F. Lai, H.-C. Chao, and A. V. Vasilakos. Big data analytics: a survey. *Journal of Big Data*, 2(1):1–32, 2015.

[157] B. Twardowski and D. Ryzko. Multi-agent architecture for real-time big data processing. In *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), Warsaw, Poland, August 11-14, 2014 - Volume III*, pages 333–337, 2014.

[158] T. Vanhove, G. van Seghbroeck, T. Wauters, F. D. Turck, B. Vermeulen, and P. Demeester. Tengu: An experimentation platform for big data applications. In *IEEE 35th International Conference on Distributed Computing Systems Workshops, ICDCS Workshops 2015, Columbus, OH, USA, June 29 - July 2, 2015*, pages 42–47, 2015.

[159] J. Varga, O. Romero, T. B. Pedersen, and C. Thomsen. Towards next generation BI systems: The analytical metadata challenge. In *Data Warehousing and Knowledge Discovery - 16th International Conference, DaWaK 2014, Munich, Germany, September 2-4, 2014. Proceedings*, pages 89–101, 2014.

[160] M. Villari, A. Celesti, M. Fazio, and A. Puliafito. Alljoyn lambda: An architecture for the management of smart environments in iot. In *International Conference on Smart Computing, SMARTCOMP Workshops 2014, Hong Kong, November 5, 2014*, pages 9–14, 2014.

[161] G. Wang and C. Chan. Multi-Query Optimization in MapReduce Framework. *PVLDB*, 7(3):145–156, 2013.

[162] S. Wang, I. Keivanloo, and Y. Zou. How Do Developers React to RESTful API Evolution? In *Service-Oriented Computing - 12th International Conference, ICSOC 2014, Paris, France, November 3-6, 2014. Proceedings*, pages 245–259, 2014.

[163] Y. Wang, L. Kung, C. Ting, and T. A. Byrd. Beyond a technical perspective: Understanding big data capabilities in health care. In *48th Hawaii International Conference on System Sciences, HICSS 2015, Kauai, Hawaii, USA, January 5-8, 2015*, pages 3044–3053, 2015.

[164] M. Weyrich and C. Ebert. Reference architectures for the internet of things. *IEEE Software*, 33(1):112–116, 2016.

[165] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.

[166] D. Wood, R. Cyganiak, and M. Lanthaler. RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C, Feb. 2014. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225.

[167] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.

[168] Z. Xie, Y. Chen, J. Speer, T. Walters, P. A. Tarazaga, and M. Kasarda. Towards use and reuse driven big data management. In *Proceedings of the 15th ACM/IEEE-CE Joint Conference on Digital Libraries, Knoxville, TN, USA, June 21-25, 2015*, pages 65–74, 2015.

[169] F. Yang, G. Merlino, and X. Léauté. The radstack: Open source lambda architecture for interactive analytics, 2015.

[170] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.

[171] A. V. Zarras, P. Vassiliadis, and I. Dinos. Keep Calm and Wait for the Spike! Insights on the Evolution of Amazon Services. In *Advanced Information Systems Engineering - 28th International Conference, CAiSE*

2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings, pages 444–458, 2016.

[172] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *PVLDB*, 6(4):265–276, 2013.

[173] R. Zhang, I. Manotas, M. Li, and D. Hildebrand. Towards a big data benchmarking and demonstration suite for the online social network era with realistic workloads and live data. In *Big Data Benchmarks, Performance Optimization, and Emerging Hardware - 6th Workshop, BPOE 2015, Kohala, HI, USA, August 31 - September 4, 2015. Revised Selected Papers*, pages 25–36, 2015.

[174] P. Zhao, X. Li, D. Xin, and J. Han. Graph cube: on warehousing and OLAP multidimensional networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 853–864, 2011.

[175] Y. Zhuang, Y. Wang, J. Shao, L. Chen, W. Lu, J. Sun, B. Wei, and J. Wu. D-ocean: an unstructured data management system for data ocean environment. *Frontiers of Computer Science*, 10(2):353–369, 2016.