QOMPLIANCE: DECLARATIVE DATA-CENTRIC POLICY COMPLIANCE ON SQL-BASED DATA MOVEMENTS

DAAN OUDEJANS

FEBRUARY 22, 2022

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science

to the

Distributed Systems Group, Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, Delft, The Netherlands

AUTHOR
D.A.J. Oudejans

TITLE
Qompliance: Declarative Data-Centric Policy Compliance on SQL-based Data Movements

SUPERVISOR
Prof. dr. J.S. Rellermeyer

GRADUATION DATE
February 22, 2022

GRADUATION COMMITTEE
Dr. ir. J.A. Pouwelse, *Delft University of Technology*
Prof. dr. J.S. Rellermeyer, *Leibniz University Hannover & Delft University of Technology*
Dr. A. Katsifodimos, *Delft University of Technology*
A. Zorin, *IBM Research Zurich*

ABSTRACT

Data compliance is essential for ensuring that organizations do not run afoul of data protection and privacy legislation. Geographically distributed data is an especially relevant topic because of recent developments in cross-border data protection agreements between the United States and the European Union. We introduce Qompliance, a novel system for automated data-centric compliance evaluation in cloud environments. This approach fills a gap in the research for higher-level data-centric compliance systems with a particular focus on geographically distributed data. Its declarative and extensible policy model allows for defining policies that can govern data movements across borders and is intended to be understandable without explicit knowledge of the governed data by employing a tag-based abstraction layer. The particular challenge is to automate data-centric policy compliance on data movements in a maintainable manner. Qompliance analyzes SQL-defined data movements to extract what data is being addressed and combines this information with additional attributes to match policies in a static manner. Policies can decide whether data movements are allowed and specify requirements on the query and the execution that should be enforced. We provide a qualitative comparison between our approach and related work, and we performed a performance analysis that shows that compliance evaluation can be done in seconds for large sets of policies.

ACKNOWLEDGMENTS

Throughout the process of researching and writing this thesis, I have received a large amount of support.

Foremost, I would like to express my sincere gratitude to my supervisor Prof. dr. Jan Rellermeyer for the continuous support, insightful discussions, and valuable feedback.

I would also like to thank the team from the IBM Research Zurich Lab for allowing me to join as a research student, and for their accommodations to make this possible during the pandemic. In particular, I would like to thank Anton Zorin for his extensive knowledge and insights and for always being available for a discussion. Furthermore, I would like to extend my sincere thanks to Douglas Dykeman, Christopher Giblin and Niels Pardon for the valuable guidance and interesting ideas that have helped shape this work.

Finally, I could not have completed this thesis during a pandemic without the great deal of support from my family and my friends, in particular Eliza, Henk-Jan, Martijn, Nick, Olav, and Philippe.

CONTENTS

1 INTRODUCTION

Given the complexity and growing scope of regulations, compliance is a significant challenge for organizations. Data compliance is essential for ensuring that organizations do not run afoul of the national privacy and data protection legislation in countries where they operate. In addition, avoidance of compliance violations gives customers and employees confidence in the organization's ability to properly handle potentially sensitive data [34]. A relatively new high-profile example of data-protection legislation that demonstrates the need for solid data compliance practices is provided by the General Data Protection Regulation (GDPR) introduced by the European Union in 2016 [15]. Privacy is an especially interesting theme that has been getting attention in the compliance space because of these significant regulatory developments. Failure to comply with these regulations can result in fines and other undesirable consequences for organizations such as reputational damage. A wide range of research has gone into modeling, encoding, and automating the implementation of systems and models to enforce regulations like GDPR [42, 46] and HIPAA [14, 28]. However, legislation tends to be very general and unspecific about the measures or controls that must be implemented for an organization to comply. Furthermore, lawmakers tend to underestimate the difficulties of automating the compliance to rules with such a broad scope. As a result, organizations need to fill in the gaps themselves and often try to compromise by implementing what is feasible without assurance that their measures are sufficient [48].

Data compliance comprises more than just regulatory compliance. For example, organizations may have privacy policies or internal policies that govern the organization's storage and use of data. Thus, the policy enforcement systems will include rules to comply with regulations but likely also consist of additional measures that the organization wants to take to protect their data. For example, the organization may want to restrict access to financial records, or they might want to anonymize customer personally identifiable information (PII) when used for analytics in order to comply with privacy regulations. The list of controls and measures can be long, with many variations of conditions and data transformations required. Therefore, building a robust system that can take these various controls and measures and verify or even enforce them is not trivial. This also shows why a compliance system should support many different checks and operations on the data and ideally should be extensible for future changes in regulations and policies.

A fascinating trend is the rising importance of regulating cross-border data movements (also in some literature referred to as *data flows*) [6]. Although some argue that the value of data is negatively impacted when it cannot flow freely between countries, the reality is that many countries are increasingly trying to regulate cross-border data movements [12]. Apart from the need to safeguard the privacy of individuals, there are various other reasons why countries wish to regulate data movements.

Governments may require organizations to keep data in their country for audit purposes, for national security purposes, or for stimulating the development of the domestic digital sector [6]. Note that we can distinguish between data movement restrictions and local storage restrictions. The former restricts in some way whether data can move across borders, whereas the latter requires the data to be stored in the country but does not necessarily restrict cross-border data movement. Further distinctions can be made based on the level of restriction, ranging from unrestrained data flow to the need to conform to some safeguards to the requirement of case-by-case *ad-hoc* approval. Especially in the global public cloud environments that we know today, where the lines between countries are blurred, compliance is even harder to enforce. Due to the inherent flexibility in how clouds operate, data may be processed in many different places and may be quickly moved to optimize for various aspects like cost or processing time. An interesting challenge here is attempting to automate the compliance to these safeguards.

The data may be subject to multiple jurisdictions depending on factors like the location of the data, the jurisdictions that apply to the organization, and in particular the jurisdictions of individuals whose data the organization processes. Data movements that are conditionally regulated based on safeguards are especially relevant because this requirement is quite widespread in regulations like GDPR. For instance, GDPR allows for cross-border data movement if the receiving country has been determined to have an adequate data protection system in place by the standards of the European Commission [6]. On the other hand, the Personal Information Protection Act in South Korea requires companies to get consent from the data subjects before moving the data in the first place [12]. Attempts have been made to build legal frameworks for cross-border data movements such as the (now invalidated) Safe Harbor and Privacy Shield agreements [47]. Privacy Shield was the more recent effort of the two (originally intended to be GDPR compliant) but it was ultimately deemed incompatible with GDPR by the Court of Justice of the European Union [13]. This is troubling to companies like Meta (formerly known as Facebook), which is saying at the time of writing that Meta may not be able to continue offering many of their services in Europe if a new transatlantic data transfer framework will not be adopted.[1]

Another challenge presents itself when we consider who in the organization is using the data and for what purpose, which is another dimension that we will consider in this research. Stationary data can be validated for compliance (e.g., whether sensitive information is encrypted), but there are additional aspects that need to be considered. Regulations and enterprise policies might also govern the use of this data. For example, only specific teams should be given access to certain data. Moreover, regulations like GDPR also have a notion of purpose, ensuring that individuals know for what purposes their data is being used. This introduces the need for access control measures, for example through Role-based or Purpose-based Access Control (RBAC and PBAC, respectively). There already has been much research towards policy compliance

1 https://www.sec.gov/ix?doc=/Archives/edgar/data/1326801/000132680122000018/fb-20211231.htm

for data at rest through the use of access control. However, research towards compliance for data on the move is lacking. Moreover, these conventional access control methods are not suitable for modern privacy requirements with notions like purpose and obligations [31]. According to Colombo and Ferrari [9], many Big Data systems lack proper data protection tools such as fine-grained access control or support for enforcing privacy policies.

It is also essential to consider how to translate and specify these controls and measures into a machine-readable format when attempting to automate their validation and enforcement. Since regulations and policies are likely to be written in natural language and mostly by people without a technical background, converting these policies into a format that can be processed and enforced by computers is not a trivial task [31]. Moreover, the set of policies can quickly grow to become very large. Therefore, a standard method for writing and managing these policies should be considered when designing a compliance validation and enforcement system.

1.1 OUR WORK

In this report, we present a novel system called Qompliance that addresses the challenges and developments that we have introduced above. We are specifically targeting data movements as opposed to data at rest since, as far as known to the authors, only limited research has been done towards data movement compliance. This scope is especially relevant for batch (ETL-like) workloads and analytics workloads where data needs to move between (geographically distributed) datacenters, as employed by companies like Microsoft [45]. SQL is a common denominator for these types of workloads because it is widely adopted and understood for querying and transforming data. SQL is widely used in distributed database systems like CockroachDB[2] and various ETL and big/distributed data systems. SQL is also increasingly being used as a basis in efforts towards building a unified method for querying different (semi-)structured data sources, including non-relational ones [33, 39]. For example, Presto[3] and its fork Trino[4] allow data scientists to run analytical SQL queries across different structured and semi-structured data sources [39]. Therefore, SQL is suitable for a unified query experience when addressing a heterogeneous set of data stores. Furthermore, the SQL query itself is the earliest point in a SQL-based data processing pipeline where one can test for compliance. This enables giving early compliance feedback to the query author. Thus, by targeting SQL-defined transformations, our system will work across many different storage solutions and can integrate into existing ETL and analytical workflows, all while providing valuable feedback to the user. Existing research exists on enforcing fine-grained access control policies through SQL and on enforcing privacy policies and regulations. However, as far as known to the authors, little research has been done towards data-centric compli-

2 https://cockroachlabs.com/product/sql
3 https://prestodb.io
4 https://trino.io

ance that takes other factors such as the geography of the data flows into account. Recent work by Beedkar, Quiané-Ruiz, and Markl [3] attempts to address this topic by integrating geographical attributes into the query optimization process for geo-distributed data.

Our work combines these ideas to build a higher-level declarative policy model and compliance enforcement system that can enable organizations to implement compliance policies governing data movements. This approach combines ideas from various fields such as access control and query processing to enforce compliance in an SQL-based data movement and transformation context. Note that if we can analyze these data movements (in the form of SQL queries) before they are executed, we can enforce access control, validate queries for specific requirements and even enforce certain required data modifications by statically analyzing the SQL. In other words, we can make sure that the resulting dataset is compliant (at least at submission time). Early detection is beneficial in an ETL context or when executing large analytical queries since these jobs might take a lot of resources and time to complete. This also removes some of the burden of achieving compliant data transformations for the data managers since many aspects of assuring compliance are automated, in particular ensuring that the result of their queries will be compliant.

This research aims to design and prototype a concept for a system that defines, evaluates, and enforces data-centric compliance policies on data movements defined by SQL queries while taking the challenges introduced here into account. By automating enforcement and simplifying the policy specification, this system aims to remove some of the difficulty associated with compliance validation and enforcement when moving and transforming numbers of datasets in hybrid cloud environments.

1.2 PROBLEM STATEMENT

We summarize the challenges addressed in this research as the following research question:

How can we define, evaluate and enforce declarative data-centric policy compliance on SQL-based data movements across distributed data stores?

To answer this question, we aim to meet the following objectives:

1. Compare systems and approaches from the related work on how they approach policy definition, evaluation and enforcement.

2. Study and define how we can process SQL to evaluate and enforce policies on a query.

3. Define an extensible policy definition model.

4. Define a way to write data-centric policies without directly referring to data and without requiring data access.

5. Determine a set of policy attributes and operations that enable authoring declarative policies, based on relevant properties from regulations and related work.

6. Design a system that can evaluate and enforce the policies from this policy model on data movements defined by a SQL query.
7. Define a method for evaluating policies and for resolving conflicts between them in a reasonable amount of time.

1.3 CONTRIBUTIONS

Our work makes the following contributions:

1. An evaluation of related systems and an identification of gaps in the research.
2. A description of a system designed to fill these gaps.
3. An extensible universal policy model, along with a suggested set of policy attributes that fit this model, that can be used to write declarative policies.
4. A compliance evaluation process based on this policy model.
5. An open source reference implementation of this system.
6. A discussion and future work suggestions that can bring further advancements in this field.

1.4 REPORT OUTLINE

The next chapter discusses related work concerning relevant access control and compliance automation systems, policy languages, (privacy) regulations and SQL processing. We will compare different systems based on their benefits and drawbacks to guide our system design. In the chapters thereafter, we will discuss the three main elements of this system: the overall proposed compliance system design in Chapter 3, the data model in Chapter 4 and the compliance evaluation and enforcement process in Chapter 5. Chapter 6 will then describe the reference implementation and Chapter 7 will discuss how we have evaluated the system. In Chapter 8 we discuss our approach and how we got there, and suggest topics for future work. Finally, Chapter 9 concludes this report.

2 RELATED WORK

This research touches upon many different topics. In this section, we will discuss relevant related work from the literature to meet a number of objectives. First of all, we want to get a better understanding of the context and positioning in which this concept and system will operate. For example, this includes regulations and control frameworks. Then we will discuss work that has made similar efforts towards access control, compliance enforcement and policy languages. Finally, we will discuss literature about some more specific topics that have influenced the direction of this thesis.

2.1 REGULATIONS & POLICIES

To understand the types of policies that our system intends to support, we draw inspiration from existing regulations and control frameworks. Because this is a large research (and legal) space, we give a few representative examples in this section.

2.1.1 *Regulations*

Privacy regulation is increasingly prevalent around the world. Important representative examples that we will discuss here are the EU General Data Protection Regulation (GDPR) introduced in 2016 [15] and the California Consumer Privacy Act (CCPA) introduced in 2018 [5]. These regulations both dictate the protection and use of Personally Identifiable Information (PII) from consumers. This includes both explicit/direct and quasi/indirect identifiers and sensitive information [17, 40].

The databases of many enterprises contain data from users from different jurisdictions. Because both regulations apply to specific groups of people (i.e. GDPR on EU citizens and CCPA on California residents), enterprises have to deal with many different laws and regulations. As a result, they opt to apply a general set of privacy measures that covers these different laws and regulations for all users. Thus, these laws might allow for a broader protection of consumers than they initially appear to do [40].

From these regulations we can derive the main requirements in order to motivate the need for supporting certain conditions and operations that can be used to automate privacy regulation compliance.

PURPOSE CHECKING Both GDPR and CCPA have a notion of purpose. These regulations mandate that consumers have a right to know about the personal information that is being collected, for what purpose and whether this information is shared with other parties. When transforming this data, it could be of great help to know whether the (result of the) transformation complies with the stated purposes. It should be noted that purpose restriction checking and enforcement is not a trivial problem because it essentially requires knowledge about the intent and planning of the actor using the data [44]. However, a data transformation system could aid well-intentioned actors to comply with purpose policies when submitting a data transformation, for example by using concept lattices as done by [22, 38]. By encoding the purposes from a policy or regulation in a lattice, one can label data attributes with the allowed/restricted purposes and compare these with the purpose for transforming the data.

CONDITIONALS CHECKING Both GDPR and CCPA require companies to allow consumers to opt in or out of certain uses for their data [17, 43]. For example, California residents have a right to opt out of the selling of their personal information. These kinds of conditionals can be easily checked and even enforced on a data transformation level (if you know the purpose of the resulting transformed data).

GEOLOCATION The physical location of data has certain compliance consequences. Different laws might apply depending on where the data resides and regulations might restrict the transfer of data to different regions [34]. For example, GDPR limits the transfer of data to countries outside of the EEA.[1] Transfer of data to countries outside of the EEA is only allowed when the third country has similar regulations compared to GDPR or when special measures are taken. Due to the increasing amount of data privacy regulation, this is becoming less of a problem. Still, it might be quite useful for the actor wanting to transform and/or store personal data in another country to be able to check whether this transformation complies with relevant geolocation policies. Another interesting aspect that could be considered is that some data center locations (or cloud providers) might have certain certifications that are required for the data transformation or storage, which could be used to optimize the transformation plan.

ANONYMIZATION AND PSEUDONYMIZATION Data without explicit or quasi-identifiers are not considered to be personal information under GDPR and CCPA [17, 40]. In other words, it should not be possible to reidentify individuals from the personal data. Therefore, anonymization or at least pseudonymization are useful preprocessing steps for many different applications such as data mining [26]. Important techniques for doing this include [17]:

- Suppression (completely removing attribute values)
- Generalization (replacing values with more general values)
- Permutation (partitioning and shuffling data within groups)
- Perturbation (replacing values while keeping statistical properties similar)

1 https://ec.europa.eu/info/law/law-topic/data-protection/reform/rules-business-and-organisations/obligations/what-rules-apply-if-my-organisation-transfers-data-outside-eu_en

It would be useful if (some of) these kinds of operations are built in to the data transformation system in order to be able to ensure anonymized processing of data. Because there is a lot of research around these different techniques, we will just limit our research to some basic strategies while taking the support for more advanced procedures into account.

access control GDPR requires "appropriate technical and organizational measures to ensure a level of security appropriate to the risk" and access control is a common measure for this [17]. Therefore, the data transformation system should prevent unauthorized users from submitting transformations on data that they do not have access to. Additionally, an access control system could also be used to give access only to select parts of a dataset. However, it should be noted that access control is a large topic and to limit the scope of this system and thesis, only higher-level access control measures can be taken into account.

encryption In the same spirit as access control, encryption is another measure to protect the data at rest and during transfer [17]. A simple way for the actor submitting a data transformation job to encrypt the data or certain attributes of the data would be a useful feature, for example to define what data should be encrypted when the transformed data is stored.

processing records GDPR Article 30 mandates that data processors maintain a record of the data processing that occurs. These records should for example contain information on the purposes of the processing, the categories of personal data and data subjects being used and whether personal data has been transferred to third countries or international organizations. A system for compliance in data movement can ensure that these kinds of records are being generated.

2.1.2 *Control Frameworks*

Companies might also have internal policies to implement certain controls. Such control instruments can be derived from existing frameworks like NIST SP 800-53 [21] and ISO/IEC 27001 which list controls for security and privacy for information systems (the latter is not freely accessible). Frameworks like SP 800-53 implement controls with requirements similar to regulations, and even mention that the appropriate regulations should be adhered to. For example, control PT-2 "Authority to process personally identifiable information" requires that the organization determines what authorities permit certain data processing and that the processing is restricted to PII for which it is authorized [21]. Additionally, PT-2 lists two relevant control enhancements. The first one is called "Data tagging" and suggests attaching data tags containing the types of processing that are allowed on the data to the PII. The second one is called "Automation" and suggests managing the enforcement of processing authorization for PII using automated mechanisms. SP 800-53 also has other relevant controls like PT-3 "Personally identifiable information processing purposes" and

PT-4 "consent" which introduce notions of purpose and consent similar to regulations like GDPR [21].

2.2 ACCESS CONTROL

Many compliance requirements can be enforced through the use of or some variation on access control. Therefore, a lot of research in the area of compliance enforcement builds upon access control methodology. In this section, we will first introduce useful access control research that relates to our work towards compliance enforcement. In the next section, we will discuss other approaches towards compliance automation beyond traditional single-point access control (e.g. in a database), which may also include access control ideas.

The most basic way to enforce policies that govern data is at the moment of data access through an access control layer. Most relational database management systems but also many non-relational systems support access control to some extent. Common access control techniques include Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC) [19]. RBAC is the most common access control paradigm and can be found in many relational and non-relational DBMSs. In the RBAC paradigm, roles describe access privileges which can then be assigned to users or groups of users.

RBAC can be seen as a subset of ABAC.

ABAC on the other hand is a paradigm where access is granted through combining attributes of the user and the context using boolean logic in order to make access decisions. These attributes can include attributes about the user making the request and the attempted actions, but also attributes from the data being accessed or even attributes from the environment of the request [19]. This is a useful property because it decouples policies from the data subject and object, i.e., no prior knowledge about the subject or object is needed. This makes access control very customizable, and the power of the system is essentially defined by the available attributes and the expressiveness of the policy language. However, it also requires some extra work to define these attributes and their allowable values, and to associate them with subjects, objects and the environment. NIST SP 800-162 gives a definition of ABAC and a framework for how it can be implemented in an organization [19]. The framework also provides an ABAC-compatible set of "functional points" which together function to make the access decisions. The following main functional points are considered:

POLICY ADMINISTRATION POINT (PAP) Responsible for the interface for managing and creating policies. Also manages storing policies.

POLICY INFORMATION POINT (PIP) Responsible for fetching attribute metadata to evaluate policies, such as environment conditions.

POLICY DECISION POINT (PDP) Responsible for making the access decisions by evaluating the applicable policies and resolving conflicts between them.

POLICY ENFORCEMENT POINT (PEP) Responsible for enforcing policy decisions made by the PDP once a request comes in.

A notable difficulty with ABAC systems is that it is generally less straightforward to check beforehand what access each individual has (before the fact audit) or to check who has access to a particular resource or set of resources compared to simpler access control paradigms like RBAC. Other challenges include planning the ABAC system (e.g. the required attributes, authorities, etc.) and maintaining traceability between high-level natural language policies and the implemented (low-level) ABAC policies.

Other access control variations have been proposed which more closely relate to compliance and privacy. Ni et al. [31] introduce what they call Privacy-aware Role-Based Access Control (P-RBAC). It is an extension of RBAC with privacy-related features like purposes and obligations. They give algorithms for evaluation and conflict checking for their model, as well as a policy authoring tool intended to make policy authoring easier to understand. Colombo and Ferrari [9] refer to variations like P-RBAC using a broader term: Privacy Aware Access Control (PAAC).

2.3 systems & languages

In this section we compare relevant compliance-related systems and policy languages to get a better understanding of the field and to determine the benefits and drawbacks of different approaches, as described in Objective 1. We establish a framework of generally applicable challenges that can be used to compare different approaches. This framework of challenges can then be used to identify a gap in the research where Qompliance should fit in, which ultimately fulfills Contribution 1. In Chapter 3 we will revisit this framework and discuss where our approach should lie within these aspects to motivate our design choices. We consider the following aspects:

aspect 1, system scope: An important axis to differentiate systems on is their intended scope. Some systems are designed to work with very high-level policies like regulations or privacy policies, while other systems are only designed to deal with low-level rules. The latter is often the case for purely access control or database approaches. For example, whereas regulations and privacy policies tend to make general statements about how to treat PII, internal policies may reference specific columns or data types.

aspect 2, policy abstraction level: We can also distinguish data-centric systems on how abstracted away they are from the data. Some systems and languages allow for directly referencing data and are thus not abstracted away, whereas other systems and languages add layers of abstraction.

aspect 3, policy complexity: The complexity of a policy depends on how understandable its model is and how many variables there are. More complex policies are likely to be more difficult to maintain.

aspect 4, policy expressiveness: The expressiveness of a policy describes how much a policy can express. Policies with more attributes, functions, operators, etc. tend to be more expressive. However, policy models that are highly expressive are likely to be more complex because of the many options.

ASPECT 5, ENFORCEMENT VS. AUDIT: Some compliance systems are built for enforcement: they output an access decision, change the data or query in some way, etc. Other approaches take a less restrictive approach solely meant for audit purposes by for example retrospectively analyzing what events or data flows occurred [38].

ASPECT 6, STATIONARY VS. MOVING: Many traditional systems focus on stationary data, e.g. by putting access control in front of a database. However, other systems specifically take moving data into account.

2.3.1 *Policy & Compliance Systems*

We start by discussing systems that address the challenge of automating compliance in some way. A common method is automating compliance using access control, but many other methods are also proposed such as analyzing data flows. Note that this is not a properly confined space. Rather, we opt to include all systems that have an interesting or relevant approach to similar (sub)problems related to data-centric compliance.

An interesting approach to policy compliance for moving data is by analyzing the information flows between systems [38]. The authors introduce GROK which is a data inventory that tracks information flows in MapReduce-like systems, and LEGALEASE which is a language for formally encoding privacy policies in a way that can be mapped to these information flows. This system specifically focuses on public-facing privacy policies close to natural language and is thus high-level according to Aspect 1. It works in an auditing manner rather than proactively enforcing access control techniques, since the information flows are analyzed retrospectively. The LEGALEASE policy model has a low complexity, low expressiveness and a high abstraction level because the authors argue that four attributes are enough to describe all privacy policies that they have analyzed, see Section 2.3.2.

Full end-to-end enforcement of compliance policies, from data ingestion to data storage to data use, is a complex challenge and presents challenges over the whole data lifecycle. A paradigm that allows for end-to-end policy enforcement is the Data Capsule paradigm [46]. It specifically targets regulations like GDPR and its consent and data processing concepts, which makes the scope high-level following Aspect 1. This paradigm has the data subjects encapsulate their personal data along with the policy that will govern the use of this data, which then moves along with the data from ingestion to its use. However, this approach as described by the authors is limited to personal data and also requires complete control over the data lifecycle. Nevertheless, this paradigm is an interesting approach towards compliance checking that can potentially be generalized. Their policy language is heavily inspired by LEGALEASE albeit a bit more expressive and complex. On the contrary, data capsules are built for enforcement as opposed to LEGALEASE. Although the authors do not mention it, the Data Capsule paradigm can be considered an improvement over the previously proposed Sticky Policy paradigm which is a common paradigm in many policy enforcement systems [16, 23, 37]. It should also be noted that many of these aforementioned approaches mostly concern compliance at the point of data access and do not take data transformations into account.

A step in the direction of data transformations is made by Khaitzin et al. [24]. They propose a system called Deep Enforcement with a strong focus on enforcement that cannot only make permit/deny decisions for data access requests, but can also transform the data to be compliant as required by the policies. These transformations are performed at the data storage level by so-called Deep Enforcers which are customized for different storage systems, in order to prevent the system from being circumvented by simply accessing the data store directly (as would be possible with some of the other systems mentioned in this section). However, this system has no control over the data once it has left the data store. Moreover, the explanations for their enforcement methods (the authors propose query rewriting and dynamic views) are lacking in this paper. Their policies are metadata based but have a limited set of attributes meaning that the policies have low expressiveness. Because enforcing happens at the database level, the deep enforcement system has a focus on stationary data.

Similar but less sophisticated database-level approaches are proposed by Agrawal et al. [1] and LeFevre et al. [25]. Agrawal et al. [1] suggest a compliance enforcement strategy using a simple query rewriting approach that only seems to work for non-nested queries. LeFevre et al. [25] suggest two enforcement approaches: one by using table views and the other by modifying the query that is limited to cell-level enforcement (i.e. they do not consider removing columns or tables). Both works suggest a meta-model to which higher-level policy language like EPAL and P3P are supposed to map to. However, these meta-models are restricted to low-level data references and a few simple other attributes like purposes and recipients. Overall, although insightful, their approach is quite limited.

Another database-level approach but specifically aimed at purpose-based access control is proposed by [4]. Their model allows for attaching multiple purpose labels on data at various levels of granularity. Furthermore, they propose to represent purposes as trees and give a method for storing these trees. Lastly, they also give a query rewriting algorithm for enforcing fine-grained access control based on these purposes, but their rewriting approach is also limited to basic queries.

There is also quite some research at the database level for NoSQL data stores, mostly document stores [9, 11]. An example of an approach close to the data but specifically aimed at document stores has been proposed by Colombo and Ferrari [10]. It relies on SQL++, a SQL extension with support for JSON documents, and needs to operate on the data directly because schema data cannot be known *a priori*. These kinds of approaches are interesting but generally out of scope for our goals because they require significant deviations from standard SQL that are not commonly used yet, and because the policies in this work cannot be evaluated in advance.

Limited research has been done towards compliance automation for geographically distributed data. Recent work by Beedkar, Quiané-Ruiz, and Markl [3] on Compliant Geo-distributed Query Processing (CGQP) introduces a method for taking geographical data flow restrictions into account when building query plans. This allows the query optimizer to not only take cost into account but also compliance restrictions imposed on the geolocation by policies. Thus, their work puts a focus on enforcement and moving data. Furthermore, they introduce SQL-like statements for expressing these policies that directly operate on the data and thus have a low abstraction level. The policies have low complexity but are limited by their expressiveness. However, their approach is limited to distributed DBMSs, but it does allow for very granular control and optimized execution plans.

The systems discussed until now all have a fairly specific application. There are also more generally applicable systems for policy evaluation and management. A notable example in this space is Open Policy Agent[2], a relatively novel general purpose policy system with its own policy language called Rego. The system itself does not make any assumptions about the kind of policies that it supports or what the policies manage, and is fully able to implement RBAC and ABAC. Even though OPA can technically be implemented to work with a higher-level scope, we consider their system to be low-level according to Aspect 1 because the policy language is very low level. The policy language essentially requires all of the logic to be in the rules, and is thus very expressive but they can also be quite complex. Generic systems like OPA are not specifically intended to be data-centric and thus they can have both a high and low abstraction level, depending on the implementation.

In the space of generic policy systems, one can also look at public cloud offerings. Most cloud vendors have some (first generation) compliance services. For example, Azure provides a compliance system for the cloud with Azure Policy, AWS has AWS Audit Manager and Google Cloud has Google Cloud Security Command Center. However, these systems mainly operate on the infrastructure and system compliance level for the various cloud products and thus cannot properly operate in a data-centric way like the other systems that we have discussed.

2.3.2 *Policy Languages*

Many different policy languages can be found in the literature, with various approaches and intended uses. For the policies for our system, an ABAC-like approach seems suitable: making access decisions based on contextual attributes from the user, the data and the transformation. An example of an ABAC policy language is the eXtensible Access Control Markup Language or XACML, originally introduced in 2001 [32]. It defines how access requests should be handled based on policy rules that rely on attributes to describe the user, action or resource. XACML is a standard published by OASIS in order to provide a specification of a common language which can be used across different systems. This in turn prevents the need to write multiple policies for different systems. XACML is XML-based and intended to be used by development teams.

2 https://openpolicyagent.org

Therefore, it is not very human-friendly and cannot easily be used by other teams such as a compliance team.

IBM introduced a similar approach to XACML in 2003, called the Enterprise Privacy Authorization Language (EPAL) [2]. It is approximately a functional subset of XACML, with a special focus on privacy. This resulted in some privacy specific design decisions like the required purpose attribute. Furthermore, EPAL has support for not-applicable policies: policies that do not make a decision but still can list obligations.

Yet another approach is Rego, a declarative policy language used in the Open Policy Agent policy engine. It has an even broader scope than XACML. Rego is very generic and does not make any assumptions about the context in which it is used. In essence, OPA can be seen as a solver for the logic programming language Rego that is used to write context-aware policies. Rego is inspired by Datalog, another logic programming language.

A significantly different approach is LEGALEASE [38]. LEGALEASE is specifically designed for encoding privacy policies. The main goal when designing this language was to create a highly usable language that can also be used by people with no knowledge about first-order or temporal logic. The language closely resembles regular policies written in the English language. The authors of LEGALEASE argue that in practice the language only needs four attributes for encoding public-facing privacy policies:

- `InStore`: Used to restrict how data is stored (e.g., certain data being stored together).
- `UseForPurpose`: Used to restrict what purposes data can be used for.
- `AccessByRole`: Used to restrict access to data to certain roles.
- `DataType`: Used to restrict policies to a certain data type.

2.4 FORMALIZATION & SEMANTICS

For getting a better understanding of SQL and policy systems, one can also look at the research done towards formalizing the semantics of these fields.

The semantics of XACML have been formalized by Masi, Pugliese, and Tiezzi [27]. The XACML standard is written in prose and does not have a formal specification of its semantics. According to Masi, Pugliese, and Tiezzi [27], it has a number of loose points which can result in interpretation issues. The authors also give a BNF-like grammar alternative because XML is confusing and insufficient for formally defining the semantics of XACML. Furthermore, the authors give a formalization of the denotational semantics of XACML.

Various attempts have been made towards formalizing the semantics of SQL [7, 8, 18, 30]. These attempts often rely on translating SQL to relational algebra [7] or some other representation compatible with the SQL semantics like (extended) predicate calculus [30]. However, many of these approaches make some simplifying assumptions which do not represent SQL in its entirety.

The approach by Guagliardo and Libkin [18] instead attempts to formalize the semantics of SQL directly. The authors also experimentally test their formalization by executing many different queries and comparing the results with existing RDBMSs. Their approach assumes what they refer to as *basic SQL*, which excludes aggregation operations. Furthermore, they assume that queries have been type-checked and compiled and thus that all attribute names have been annotated with their corresponding table names. This means that all base tables and subqueries have an explicit name and that the names of all output attributes are explicitly listed in the `SELECT` clause. Moreover, they discuss how full names in queries can be resolved. However, the rest of this paper is more oriented towards the further evaluation of SQL queries, which is less relevant for this work.

Another approach has been made by Chu et al. [8], which does take major SQL features into account like bags, aggregation and indexes, but instead leaves out three-valued logic (NULLs). The semantics they propose are based on K-Relations and homotopy type theory, which allows for implementing the semantics in the Coq theorem prover. Their main goal is to show query equivalence for use in query optimizers and thus the query semantics have to be preserved. However, preservation of semantics is not a goal for this work.

2.5 QUERY REWRITING

Query rewriting for privacy policy enforcement and access control has been researched before. For example, LeFevre et al. [25] demonstrate a method for rewriting queries to include access conditions based on policies. These access conditions are specified on a cell level and are also stored in a table, which allows for joining these conditions with the actual data by modifying the query to limit the disclosure of values. Similarly, Agrawal et al. [1] and Byun and Li [4] provides similar techniques but are only shown to work for basic or non-nested queries.

Other relevant work towards query rewriting for the purposes of this research is the work towards SQL++ query rewriting for ABAC in NoSQL data stores from Colombo and Ferrari [10]. They demonstrate a method for modifying the SQL++ queries to create in-memory *authorized views* which comply with fine-grained access policies, meaning that this method can also enforce policies at the cell level. The main benefit of this proposed method is that the system does not need any prior knowledge about the schema of the data. However, their approach also suffers from disadvantages: because of the reliance on SQL++ it is not compatible with RDBMSs and for performance reasons they have to modify the source data to include policy information that apply on a document.

3 SYSTEM DESIGN

In this chapter, we will combine the challenges set out in Chapter 1 with the ideas from various fields of related work discussed in Chapter 2 to introduce our proposed system. First, we will discuss the positioning of our intended approach compared to related work. We will then discuss important aspects of the system design and how they relate to the challenges and related work. The details about the policy model and evaluation are then elaborated upon over the next chapters.

3.1 POSITIONING

As we have seen in the introduction and related work, compliance is a complex topic with requirements and challenges at many different levels. To be able to discuss the design decisions made for this system, we first explain the design spectrum in which the system will be positioned. We do this by making a relative comparison between different relevant approaches from the related work using the aspects defined in Section 2.3. We then discuss where our own approach is intended to be positioned in this design spectrum. Note that the common denominator between the systems that we compare here is that they are all data-centric approaches towards compliance in some way. An overview of this analysis is depicted in Figure 3.1. Note that for this comparison and in this figure, we have used a representative set of the most interesting and relevant systems from the related work with widely differing approaches. Some systems are not included on all axes if the axis is not directly applicable.

Data-centric here means that the compliance system uses the governed data as part of the context (and possibly also performs actions on it).

ASPECT 1, SYSTEM SCOPE: Systems clearly have varying expectations and goals when it comes to the targeted scope of the system. It becomes clear from Figure 3.1 that no systems that we have evaluated seem to target the middle ground between higher-level policies like privacy regulations and lower-level policies like business rules on tables. Systems like LEGALEASE [38] and Data Capsule [46] both are purpose-built for regulations and therefore have limited support for more versatile lower-level policies. On the other hand, the systems that live closer to the database level like Compliant Geo-distributed Query Processing [3] generally have a policy model that is far away from high-level policies. OPA does not have a clearly defined system scope as it is designed to work with any kind of input data. Our intention is to target the gap in the middle: the system should be able to handle lower-level internal rules about the data, but also high level privacy policies and regulations. This means that we sacrifice something on both sides: the policy model will be too low-level to be able to directly encode regulations without some effort, but will be too high-level to be able to enforce highly specific cell-level rules.

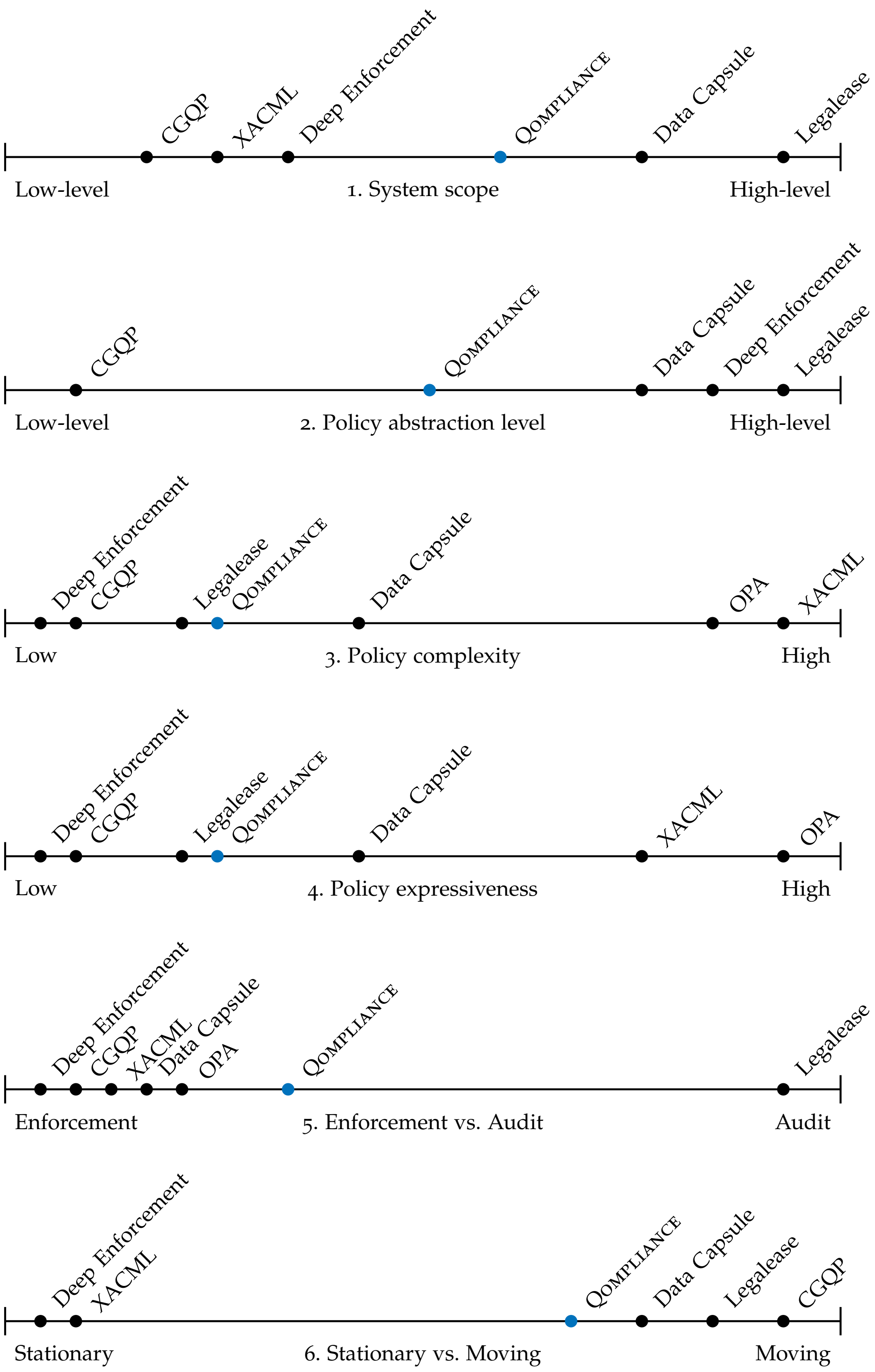


Figure 3.1: Comparison of selected related work and our intended approach for various aspects. The absolute position of the points do not represent any concrete value, but rather serve to compare the related work in a relative manner. Mentioned are: Compliant Geo-distributed Query Processing [3], XACML [32], Deep Enforcement [24], Data Capsule [46], Legalease/Grok [38] and Open Policy Agent.

ASPECT 2, POLICY ABSTRACTION LEVEL: This is somewhat related to the previous aspect, as is apparent from Figure 3.1. For systems that attempt to take on a broad scope, the policies will tend to be more abstracted away from the underlying data, Deep Enforcement [24] being the exception. Both approaches have their advantages and disadvantages. A notable disadvantage of low-abstraction policies is that they tend to make direct references to the data, such as in many ABAC or database approaches. However, this does allow for a lot of flexibility and custom low-level rules. For policies that are abstracted away from the data, the inverse holds true: policies become more maintainable and understandable, but are generally less powerful. Moreover, higher-level policies tend to be closer to the natural language used in privacy policies or regulations by using some kind of abstraction layer such as references to data types or tags. Ideally, our approach would live somewhere in the middle to correspond with our intentions for Aspect 1.

ASPECT 3, POLICY COMPLEXITY: Policy complexity seems to go hand in hand with expressiveness. For example, the XACML policy model is quite complex since it supports many different operators, matching functions, conditions and obligations which makes the policies quite hard to understand and write. OPA is similarly complex, but has an easier to read declarative language. However, these policies are very versatile and powerful. On the other hand, LEGALEASE is on the other end of the spectrum with only 4 attributes, but is therefore also very limited [38]. Our goal is to be in the middle here, leaning towards the lower complexity side. We can remove a lot of complexity by moving the attribute definition and evaluation to the application layer (as opposed to models like XACML or OPA/Rego where the matching logic is determined in the policies themselves). That way, the model is still somewhat extensible by someone with some knowledge about the system, but the policies become easier to write and understand by laymen.

ASPECT 4, POLICY EXPRESSIVENESS: As we have seen with Aspect 3, expressiveness seems to be correlated with complexity. One notable system is OPA, which is the most expressive because it is the most general approach. Again, the middle ground would be ideal, with a balance between expressiveness and complexity. Furthermore, it would be useful if our system takes extensibility into account so that someone who implements this system can steer this tradeoff based on their own goals.

ASPECT 5, ENFORCEMENT VS. AUDIT: Another distinctive aspect is whether the systems are targeted towards enforcement or audit. Basically all evaluated systems put a clear emphasis on enforcement. LEGALEASE [38] is the only exception, since it analyzes data flows retrospectively and thus does not enforce anything. Our system also targets early enforcement. However, we recognize the importance of auditability and thus our design should take this into account.

The importance of auditability is demonstrated by GDPR Article 30, which mandates that data processors maintain processing records. See Section 2.1.

ASPECT 6, STATIONARY VS. MOVING: With an increasing focus on cloud and distributed data, it makes sense to track data flows and take the geography into account. Our intention is to put the main focus on moving data, since the main challenges of this research are related to geographically distributed data. However, the approach we propose in this chapter is still suitable for stationary data in some ways.

As Figure 3.1 clearly shows, our approach is meant to cover the middle ground for many of these aspects. Over the next subsections, we will explain what QOMPLIANCE looks like and how it manages to balance the tradeoffs represented by these aspects.

3.2 ACCESS CONTROL PARADIGM

We will first approach the system design from the perspective of access control, a well-established field with related work that can serve as a basis for our design. The focus of this project is on designing a service that governs data movement and transformations for another system that handles the SQL-based data processing (e.g. an ETL platform), and will thus process the outcomes of our system. The policy language and the evaluation system are used as a way to authorize certain uses of data. Clearly, parallels can be drawn to the field of access control, which has been discussed in Section 2.3.2. Note that access control security policies and privacy policies often govern the same set of data and because access control is even a common measure to comply with GDPR requirements, using a single integrated model for doing both will simplify its management [31].

A suitable model for our policy language is the Attribute-Based Access Control (ABAC) model. It is especially suitable because we can attach numerous attributes to the users, data and environment [19, 20]. The power of policies in ABAC is in practice defined by the supported attributes and the policy model [19]. Therefore, attributes have the potential to bridge the gap between higher level policies for privacy policies and regulations and lower-level policies such as business rules about data, as identified in Aspects 1 and 2. These attributes can either be manually assigned or automatically inferred. Policies can then use these attributes to allow or deny data movements without directly addressing individual permissions, which makes ABAC very suitable for distributed and rapidly changing systems [20]. Because of this abstraction, if new users are added, the managed data changes or the environment changes, policies do not have to be updated which is a useful property that we wish to extend.

A policy language that implements ABAC like XACML would be somewhat suitable for our purposes. However, XACML is a fairly complicated policy model. XACML is XML-based and supports many different attributes, relationship, functions and even conflict resolution strategies within the policies. Therefore, it is difficult to get a quick understanding of a policy.

For the purposes of this research we can make a few simplifications and sacrifices compared to a sophisticated policy language like XACML, in favor of a simpler and higher level policy language as proposed for Aspect 3. This research attempts to demonstrate how data-centric compliance validation and enforcement could work when moving data in a cloud environment. For the purposes of this research we opt to design a novel policy model, inspired by ideas from the related work discussed in Section 2.3. This novel policy model is intended to be take take our goals for this system into account, notably Objectives 3 and 4. Especially the latter objective is the main reason why we need a new policy model, because by extension we aim to make the policies simpler to understand for people unfamiliar with technical details such as the schemas of the governed data and complex policy languages.

This novel policy model will be a similar but simpler ABAC-like implementation. We will use ideas from ABAC, but the intention is not to strictly adhere to the traditional definition of ABAC as defined by NIST [19]. For example, we will move most of the logic, which ABAC implementations like XACML allow for, to the application layer to reduce policy complexity (see Aspect 3). Moreover, our goal is not to perform strict access control to a dataset per se. This is because of our focus on moving data, as explained in Aspect 6. We want to govern how this data moves, which can mean denying a movement entirely, but also just requiring that the data will move to a particular location without making an access decision.

More on how we deviate from ABAC in the traditional sense will be discussed in the next sections.

3.3 POLICY MODEL

Now that we have a basis for our policy model and we have an idea of the challenges that we are trying to solve (Chapter 1) and an impression of the design spectrum (Section 3.1), we can start shaping our policy model more in-depth. Our system takes a unique approach towards the policy model. The policy model is intended to be declarative and relatively high-level, meaning that the policy author does not need to know of or engage with the exact schema of the governed data or with the supported checks and operations in the compliance system. This means that the policy author can write relatively simple policies that can be closely mapped to how someone would discuss them in natural language, like:

- *Store all medical personally identifiable information in HIPAA compliant storage*
- *Make sure that all data that is currently in the Netherlands stays within the Netherlands*
- *Deny data scientists access to transform sensitive data*

Note that the first two examples here do not actually allow or deny a data movement, they simply tell what a data movement should adhere to. This is a notable feature of our policy model which departs from access control in the traditional sense. The policy model supports what we call NON-DECIDING policies: policies that do not make a decision. They are meant to put restrictions on what data movements should adhere to, without deciding whether this data movement is allowed or not in the first place.

The examples above show how someone would ideally want to write policies for compliance. As discussed in Aspects 1 and 2, our goal is to find the middle ground between high-level and lower-level policies.

This requires a flexible abstraction layer to map between policies and data. This layer of abstraction between the declarative policies and the low-level implementation is facilitated by a few key assumptions and design decisions:

METADATA The system is metadata-heavy, meaning that a lot of attributes assume the presence of detailed metadata. This metadata gives meaning to the entities addressable by the policies, which enables better human comprehension. For example, data stores are assumed to have metadata on their location and additional attributes like compliance level (e.g. HIPAA compliant, suitable for financial data, suitable for government data).

TAGS This is an extension of the design decision to make the system metadata-heavy. Tags are used as a human-friendly way to give context to governed resources at almost any level of detail without knowledge about the exact schema. Moreover, tags can work on other data models than just the relational model, meaning that our design can support data stores with alternative data models as long as it supports SQL and mapping its data model to schemas and tables. In our design, tags can be attached to data stores, datasets/tables and columns. This way, policies are decoupled from the underlying data definition, which in turn results in easier policy writing and management. They can be used to make broad references to the data, like 'all PII' or 'all email addresses', that can provide more meaning to policy authors than data references would and without having knowledge of the data definition. In case the underlying schema changes, policies do not have to be changed. Compared to low-level related work that supports policies that directly reference data in some way, a set of policies will be easier to comprehend, author and maintain. Compared to high-level related work that only supports very general policies, a tag-based system allows for more flexibility because tags can also be highly specific (e.g. only apply on a single column or table) and therefore cater to the needs of policies closer to the data. A disadvantage of this approach is that you may need a lot of metadata on the governed data for this to work, depending on the complexity of the policies that need to be supported. However, a lot of research has been done towards automatically generating, inferring and adding metadata like this [38, 41, 48]. Moreover, tags can be used in other systems as well and may even already be present. Another challenge is that the lifecycle of the tags in the system has to be at least as long as the lifecycle of the data in the system, meaning that the set of tags can grow overtime to become more complicated. To keep the set of tags structured and maintainable, and to prevent tags with highly similar meanings from being added, we propose to store the tags in the system in a hierarchy.

An example of an alternative data model could be data files on HDFS managed using Apache Hive, queried with SQL through Presto.

ATTRIBUTE TYPES By having predefined attribute types, some flexibility is taken away in favor of reduced policy complexity and clear semantics (compared to highly expressive but complex languages like XACML and Rego). This decision is what makes the system and its policies the most opinionated because it makes some rigid assumptions about the attributes and how they are processed. A suitable example that demonstrates the usefulness of these assumptions is the hierarchical attribute type. Because most context attributes are stored in a hierarchy, how these attributes get evaluated becomes quite easy to understand. Furthermore, a hierarchical organization can reduce the number of attribute values needed in policies and by extension improve the clarity and maintainability of policies. The set of possible attribute values is limited by the tree, and conflict resolution is as easy as taking the more specific policy applicable on the input to take precedence. Note that in many applications of attributes there is already an implicit hierarchy (e.g. purposes and roles), which makes them trivial to understand. Tags are a special kind of hierarchical attribute, which not only improves maintainability but also allows tags to benefit from this conflict resolution property. Although we suggest attribute types like hierarchies in this research, the attribute system is designed to be extensible at the application layer.

From here, we can introduce the actual structure of a policy. Because of the context in which Qompliance is intended to operate, the structure is slightly different from traditional ABAC systems or other policy systems that we have reviewed in Chapter 2. A policy has four main elements, namely:

1. *Basic metadata* like a unique policy name which acts as a natural identifier, the owner/author of the policy, etcetera.

2. The *context* in which the policy applies, specified by attributes in a manner comparable to ABAC, i.e. attribute-value pairs. Between attributes we assume a conjunction relation (i.e. AND-relation) because this more closely follows natural language in legislation. Between attribute values for the same attribute, we assume a disjunction relation (i.e. OR-relation).

For more discussion on the assumed relations between attributes, see Section 8.1.1.

3. The *decision* about whether the movement is allowed under the policy's context. Note that this differs slightly from an access decision in the context of access control because this does not make a direct decision about access to the data itself but rather whether a data transformation (defined by the SQL and additional attributes) is allowed or not. Still, this can theoretically serve as an access control mechanism if all data access is required to go through this system first. As discussed at the beginning of this section, policies do not have to make a decision as they can also be *non-deciding*. This allows the policy author to specify requirements that should be enforced in a particular context, without making a final decision.

4. The *requirements* that should be fulfilled if the data movement is allowable. Allowable means that in principle, the set of applicable policies allows the data movement. However, it will only be allowed *iff* all requirements can be satisfied. Note that the 'requirements' naming differs from the conventional naming in some ABAC and privacy systems where a name like 'obligations' is more common.

Whereas obligations in these systems mostly refer to actions that should be taken once a policy is applied, requirements in our system work a little bit differently. The requirements name is carefully chosen because it implies that the requirements should also be fulfilled for the input to be valid and allowable. Because of this requirements element, an *allow* policy essentially reads as: "if the policy context applies to the input, allow the movement *iff* all requirements can be satisfied". Similarly, a *non-deciding* policy reads as: "if the policy context applies to the input, make sure the requirements are satisfied without making a decision". Note that this implies that *deny* policies cannot specify requirements, because if a movement is not allowed, requirements on this movement cannot be enforced anyway. This allows for obligation-like actions, but also for imposing additional restrictions on the data movement. In turn, this enables our system to provide recommendations to the user on how to make their input compliant or output requirements that can be automatically fulfilled by another system (e.g. the required data geolocation). Additionally, this naming also implies that these requirements can be evaluated at any time. For instance, if someone were to build a continuous/repeatable ETL pipeline where the compliance state needs to be reevaluated from time to time, the system would be able to check whether the data is still compliant with a "data should stay within the EU" requirement.

3.4 ATTRIBUTES

To get an understanding of the types of policies that we can write with this policy model, we propose a set of attributes that QOMPLIANCE supports. For determining this set of attributes, we have mainly looked at the related work and regulations that have been discussed in Chapter 2 as well as discussions with IBM Research employees with compliance-related experience about what would be useful attributes. We categorize attributes in the two categories that can be used in the policy model: context and requirements. Some attributes may be applicable to both categories, others may only be applicable to a single category. Note that these attributes also have types. The type of attribute determines the values that it can take and how they are processed. These types will be introduced in Chapter 4 once we have an understanding of the data model.

3.4.1 *Context Attributes*

TAG An important attribute which enables to the policy author to write data-centric yet declarative policies. Because of this layer of abstraction, data can be referred to by including tags in a policy. These tags are then resolved to the data schemas that have been labeled with these tags, which can then be matched to the input.

ROLE The role attribute is similar to Role-Based Access Control systems.

PURPOSE The purpose attribute is added because it is a common notion in data and privacy regulations. It can be used to select purposes for which a policy should apply. This attribute value is assumed to be provided by the submitter of the transformation.

DATA LOCATION The data location attribute enables the geolocation features of this system. It allows policy authors to write policies that govern how data should be managed across borders and in different jurisdictions. Together with the data location requirement, it can also be used to 'steer' data to the right location.

STORAGE CLASSIFICATION The storage classification context attribute can be used to write policies for data that lives on data stores with a particular classification.

3.4.2 *Requirement Attributes*

DATA LOCATION The requirement counterpart of the context attribute. This attribute can be used to require the data to be processed and stored in a certain location, for example to prevent data from leaving a particular jurisdiction.

STORAGE CLASSIFICATION The storage classification requirement dictates where data can be stored based on classification metadata attached to data stores. These can be used to ensure that data resides in data stores which conform to a particular classification, for example: compliance certifications, disk encryption, or classified as suitable to store sensitive data.

WITHOUT The 'without' requirement can be used to mandate that data is not included in the final query result. This is useful for writing policies that restrict what data can be included in the final output, without preventing the data access entirely. This attribute can be used to make rewrite suggestions for the SQL to make it compliant if it is not already.

AGGREGATE The 'aggregate' requirement can be used as an alternative to the 'without' requirement when the resulting dataset is still allowed to contain some statistical properties about certain data. It can for example be used as an alternative anonymization strategy.

3.5 DATA FLOW & PROCESSING

Given the policy model that QOMPLIANCE should support, we can now discuss how the data should flow to, within and from the compliance system. This data flow gives a general overview of how the system is supposed to work. Detailed descriptions of the data model used in the system and the compliance evaluation process are given in Chapters 4 and 5 respectively.

The data flows similar to how it would work in an ABAC system with its functional points as defined by NIST [19]. Figure 3.2 gives a visual representation of our system mapped to the ABAC functional points.

Refer to Section 2.2 for more information on the ABAC functional points.

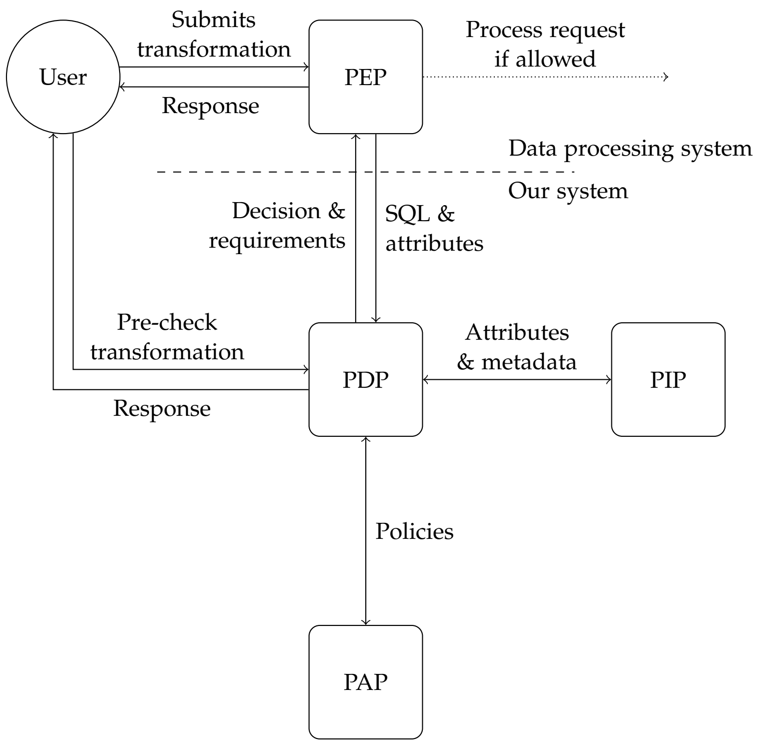


Figure 3.2: Diagram showing the different functional points in our system, based on the main functional points for the access control mechanism for ABAC described by Hu et al. [19].

Note that the scope of our system is a service which can interact with a data management system like an ETL platform to evaluate and enforce compliance on data transformations. Therefore, our service delegates the responsibility of enforcing the decisions and requirements to another system. The system is not designed to touch the actual data or to generate execution plans, which is possible because data references in policies are entirely based on and evaluated with the schemas of the data. In other words, our system outputs a decision along with a set of requirements that are supposed to be enforced by another system that processes the data. Thus, in terms of the ABAC functional points, our system is not responsible for the Policy Enforcement Point, but only for the Policy Information, Administration and Decision Points. This is represented by the dashed line in Figure 3.2.

The diagram also shows a few other small differences from the traditional ABAC model. First, for our system the PEP also responds to the user in case the user needs to modify their submitted transformation in some way or if the request gets denied entirely. Furthermore, the PDP does not only output a decision but also a set of requirements whenever applicable. Lastly, an important difference is that our system allows for pre-checking whether a transformation is allowed or not. Our approach with policy model from Section 3.3 and the set of attributes from Section 3.4 allows for completely static evaluation of the policies. Therefore, we can check whether a transformation will be allowed or not before it needs to be handled by a data processor and thus it can be evaluated in advance without touching the data. A major benefit here is that we can let the user more directly interact with the Policy Decision Point, by allowing the user to submit their intended transformation for a pre-check. The system can then just perform the evaluation like it normally would and provide the user with direct feedback, which is especially useful if the user needs to change something before the transformation can be submitted. A use case of this would be an editor-like user interface for writing SQL queries, which can then periodically submit the SQL for compliance evaluation. Thus, our system can provide early feedback to the user about whether their intended SQL-based transformation will be allowed, where the data can be processed and stored and whether the user needs to change anything to ensure compliance.

The PIP is responsible for retrieving and managing metadata and attributes. Similarly, the PAP is responsible for retrieving and managing policies. Because our system is designed to not require changing the data model of the governed data, or even data access in the first place, the PIP is also responsible for managing the schema information. This schema information is assumed to be fed by some other system. Not all attribute information has to come from the PIP though, as attributes can also be provided along with the submission to the PDP (as shown in Figure 3.2). For example, the user should provide the purpose for the transformation, as explained in Section 3.4.

The SQL query defining a data transformation is the earliest point where one can intercept what a transformation will look like (as opposed to later downstream in a data processing pipeline). Therefore, analyzing the SQL query is a suitable method providing early compliance feedback. Because our system is intended to operate directly on the SQL specification of a data transformation, the system has to include SQL parsing and validation capabilities. That way, the system is fully able to ingest a user-specified job containing a SQL query and possibly additional attributes. The need for parsing the SQL is evident. The SQL needs to be validated to make sure that it contains valid references to schemas that our system is aware of (through the PIP), or provide feedback to the user if this is not the case. Moreover, parsing and validating the SQL in our system is especially useful in the scenario where the user directly interacts with the compliance system. The user not only gets compliance feedback but also feedback about the validity of their SQL query, all without reaching the actual data stores or a data processing service. This feedback can for example be added to the editor-like interface we proposed earlier in this section.

3.6 other considerations

There are a few other smaller design considerations that are worth mentioning. First, because our system is supposed to operate as an external service to a data processing system, care should be taken to ensure that the communication between these systems is secure. Part of the solution can be networking measures that are out of scope of this research. However, a good measure for our system would be to attach cryptographic signatures to the output of our system. This way, anyone can validate whether a particular decision was indeed made by our system at a particular point in time.

Furthermore, as discussed in Aspect 5, we want to take audit into account as is for example required by GDPR. For this, our system should write all of the decisions made to a log, along with the signature.

Lastly, one of the challenges of this research was to target distributed settings and data movements. However, it is worth noting that the techniques presented here are also suitable for stationary data in a single database setting. By employing our system as a layer in front of the database, many of the same ideas can be utilized (apart from distributed data-related attributes). That way, our work can be compared to and used in a similar manner as some of the other related work that we have discussed, such as the work by Agrawal et al. [1], Khaitzin et al. [24], and LeFevre et al. [25].

4 UNIVERSAL DATA MODEL

In this chapter we introduce a universal data model that is used throughout this work to map all related data back to a common model. The basis for this model is the policy definition model that is used for writing the declarative data-centric compliance policies introduced in Chapter 3. Furthermore, we present a metadata model for storing the metadata that can be used for writing policies. We give a formal description of this universal model and explain its various components. The benefit of this universal model is that we can map this model to other representations, such as a policy language and a database implementation. These representations are explained in Chapter 6 which discusses the reference implementation of the system and how it relates to this model. The formalization provides a common language for further use throughout this work.

4.1 SCHEMA METADATA

First, we present a model for the metadata that the system needs to keep track of. The most important type of metadata are the schemas of the data that the system is governing. Other metadata for example includes information on the data stores and tags.

We will start by defining the three levels of schema data that the system keeps track of.

DEFINITION 1, DATA STORE: A data store d contains metadata about the data stores/databases that the system keeps track of. It is defined as $d = \langle id, name, location, S \rangle$. D with $d \in D$ denotes a set of data stores.

DEFINITION 2, DATA SET: A data set s contains metadata about the individual data sets/tables in the tracked data stores. It is defined as $s = \langle id, name, CL \rangle$. S with $s \in S$ denotes a set of data sets.

DEFINITION 3, COLUMN: A column cl is the lowest level of data that we consider. It contains metadata about the columns/fields within data sets. A column is defined as $cl = \langle id, name, type \rangle$, and CL with $cl \in CL$ denotes a set of columns. The type of the column is the data type in the database.

4.2 ATTRIBUTE METADATA

Attribute metadata is additional metadata that enables the functionality of a policy attribute. This can be the values that the attribute can take, and possibly additional metadata for connecting the attribute values to other data such as schema data. Thus, the specific metadata that is needed will differ per attribute. We will provide some generic definitions that can be extended to fit the requirements of a particular attribute.

DEFINITION 4, ATTRIBUTE: An attribute is a policy element that is used to add functionality to a policy, specifically context information or requirements. An attribute *at* is defined as $at = \langle id, type, category, values\rangle$, with AT being a set of attributes where $at \in AT$. Here, $type$ is the attribute type and *category* specifies whether this attribute is a context attribute, a requirement attribute, or both. Our proposed attributes have three main attribute types: *enum*, *hierarchy* and *tag-reference*. $values$ are the set of values that the attribute can take, which may differ by type.

Thus, Definition 4 introduces the notion of attribute types for the attributes introduced in Section 3.4. These attribute types determine how attribute values are validated, matched and defined.

DEFINITION 5, ATTRIBUTE VALUES: An attribute value v can have varying properties depending on the attribute type. In the case of an *enum* type, we assume that $v = \langle enum\_value\rangle$ where $enum\_value$ is just a unique name for the value. In the case of a *hierarchy* type, a value will instead look something like $v = \langle id, node, children\rangle$. Valid attribute values for *tag-reference* types are simply names of the tags that the system is aware of, i.e. $v = \langle t.name\rangle$ for some t where $t \in T$ (see Definition 8). This is a somewhat rough recursive definition of a tree, where *node* is the unique name of the node in the tree and *children* refers to the *id* of other tree values.

Of course, the actual implementation of a tree can differ from this definition.

It is important to recognize that Definition 5 is just a generic representation of an attribute value, and that the actual semantics of these values will differ between attributes. For example, for determining a policy match or for processing requirements, additional metadata may be needed. An example of which are storage classifications:

DEFINITION 6, STORAGE CLASSIFICATION: A storage classification *sc* provides additional information about the properties of a data store. It extends the idea of an *enum* attribute type and is defined as $sc = \langle name, D\rangle$, with SC being a set of storage classifications where $sc \in SC$. Storage classifications thus have a many-to-many relationship with data stores D.

To be able to discuss attributes in a universal way and thus without knowing their exact underlying semantics, we define functions which all attributes should have:

1. $evaluate(inputVals, policyVals)$, which determines if the policy contains attribute values which are considered to match with the input value.

2. $checkConflicts(inputVals, policyVals1, policyVals2)$, which checks if there are conflicts between the two sets of attribute values on the input and returns what set of attribute values is more specific.

3. $evaluateRequirement(policyVals)$, which processes the attribute values from a policy for this particular requirement (i.e. checking if they are valid values and generating further output).

4. *generateFinalOutcomes*(*evaluatedRequirements*), which generates the final set of outcomes by combining the evaluated requirements for this attribute.

The first two functions should be available for context attributes, the latter two functions should be available for requirement attributes. The exact implementation of these functions may differ, but this functionality should be present. The system does not even need to be aware of additional attribute metadata for certain attribute implementations, as long as two properties are satisfied: the above functions are implemented and the system knows how to parse and process the attribute's values. An example of such an attribute could be a 'role' attribute that externally validates whether the input role (e.g. the role of the user) complies with a role listed in a policy. The system just needs to be aware of how to process these role values and implement the above methods using this external service.

Tags deserve some special attention as they serve an important role within the system: they enable the layer of abstraction between the governed schemas and the policies. A reference to a tag can be used as a special type of attribute value.

DEFINITION 7, DATA REFERENCE: A data reference dr points to a particular schema element, either a data store, dataset or column and can be referenced by tags. It is defined as $\langle id, ref \rangle$, where DR is a set of data references with $dr \in DR$. Here, ref is a SQL like data reference, e.g. `DB.Table.Column`.

DEFINITION 8, TAG: A tag t references a set of data references DR to map the name of the tag to the underlying data. It extends the idea of a *hierarchy* attribute, and is defined as $t = \langle id, name, children, DR \rangle$, where T denotes a set of tags with $t \in T$. Thus, tags have a many-to-many relationship with data references.

Thus, tags are a hierarchy attribute with the special property that the nodes also contain a set of data references which map the tags to actual data. Organizing tags in trees has the benefit that it gives additional meaning to the tags, which helps with writing clear declarative policies.

4.3 POLICIES

Now that we have established the metadata models, we can finally define what policies should look like. A policy consists of a combination of attributes from Section 4.2, for context and optionally for requirements.

DEFINITION 9, CONTEXT ATTRIBUTE VALUE: A set of context attribute values C together define in what context a policy should apply on the input. A context attribute value should be valid as defined by Definition 5 for the particular attribute (Definition 4) that it references. Formally: $c = \langle id, at.id, val \rangle$, where $c \in C$, $at \in AT$ and $val \in at.values$.

In other words, the context of a policy is defined by a set of attribute name-value pairs. The system can then use the name to resolve what attribute the value belongs to, and pass the value to its functions. The definition of requirement attribute values is very similar:

DEFINITION 10, REQUIREMENT ATTRIBUTE VALUE: A set of requirement attribute values R together define the requirements that should be enforced or output by the system if the policy applies. A requirement attribute value should be valid as defined by Definition 5 for the particular attribute (Definition 4) that it references. Formally: $r = \langle id, at.id, val \rangle$, where $r \in R$, $at \in AT$ and $val \in at.values$.

Finally, we can define a policy:

DEFINITION 11, POLICY: A policy p is defined by context and requirement attribute values, a decision, and additional metadata. We define it as: $p = \langle id, name, owner, decision, C, R \rangle$. The *decision* should be one of: ALLOW, DENY, NON-DECIDING.

5 COMPLIANCE EVALUATION

In Chapter 3 we have introduced the general design of the system and in Chapter 4 we set out the data model for the policies and metadata. In this chapter, we describe how the policies are actually matched and processed to reach a final compliance outcome.

The input for a compliance evaluation job consists of the raw SQL which represents the data transformation, internal metadata and external metadata. Internal metadata is metadata that is managed or can be accessed directly by the compliance engine, an example of which would be the location of a data store. External metadata is metadata that should be provided by the user alongside the SQL or that is retrieved by an attribute for evaluation, e.g., the purpose for a transformation.

The entire compliance evaluation process of a job is divided into the following general steps:

1. Parse and validate the SQL and extract what tags are applicable on the SQL
2. Match input attributes with policy attributes to find what policies are applicable on the input
3. Check for policies that make conflicting decisions and try to resolve these conflicts
4. Evaluate and process the policies' requirements and resolve conflicts between them
5. Generate, sign and return/forward the final output

This chapter dedicates a section to each of these steps with explanations of the process along with examples and pseudocode.

5.1 ANALYZING THE SQL

The first step of a compliance evaluation job is to validate the user-defined SQL to see whether it matches the schemas that the system is keeping track of and to see if the SQL itself is valid in the first place. This involves parsing the SQL (while taking to account possible dialects and extensions), checking the SQL semantics and checking whether the data references in the SQL conform to a known schema. The assumption here is that our system has to be aware of the schemas of the data that it is governing. Although parsing and validating SQL is an important step in an evaluation job, it is not in the scope of this work and existing solutions exist that can be used here (see Chapter 6).

Remember that for the proposed static compliance evaluation we only keep track of the data stores, tables and columns.

After the SQL has been validated, all data references should be extracted. Note that in this work we only consider `SELECT` queries because data definition and manipulation are not relevant for the purposes of this research. However, the ideas presented here can be extended to include other types of SQL statements for systems with broader applications.

Because the proposed system is intended to work across different data stores (possibly also with other models than the relational model, as long as they support SQL), we will make a simplifying assumption about how data is referenced in the SQL statements that our system supports: a schema name refers to a data store. This simplification was made because data stores are assumed to have their own attributes such as their location. Furthermore, it makes reasoning about the system somewhat easier and schemas can still be used if you map them as data store names with aliases. Thus, we consider a fully qualified data identifier to consist of the column name, the table name and the data store name. Tags can reference all three of these data identifier levels, thus we have to extract all three from the SQL.

A typical SQL `SELECT` query over a relational database under our model contains references to columns, tables and data stores. The entire set of referenced data identifiers is the set of columns, tables and data stores referenced in any place in the SQL. This set can then be joined with the set of all tags tracked by the system to discover what tags are applicable on the query. Important to note is that we need to know what tables and data stores a column belongs to, otherwise the column name may not be unique. Similarly, we need to know what data store a table belongs to. In some cases, these may not be immediately apparent, such as when using the asterisk or when columns are referenced without explicit table references, as shown in Listing 5.1.

```sql
SELECT column1, column2
FROM db.table1, db.table2
WHERE column1 IS NOT NULL;
```

(a) Column references without explicit table references

```sql
SELECT *
FROM db.table1;
```

(b) Use of asterisk to project all columns

Listing 5.1: Examples of column references that need to be resolved.

We assume that the validation process enriches the data references in the SQL by introducing aliases by using the database schemas to be able to resolve these cases. This means that we are now dealing with SQL that looks like the example in Listing 5.2. Building the set of data identifiers then becomes as simple as building two maps: a map from the column aliases to column-table pairs and a map from the table aliases to table-store pairs. To resolve what data stores columns are under, these two maps can be joined. By then adding all data store, table and column references to a set we have a set of all data identifiers in the query. For Listing 5.2, this set will look like the following:

{`db, db.table1, db.table2, db.table1.column1, db.table2.column2`}

It is trivial to then join this set of data identifiers with the set of tags tracked by the system to reveal what tags apply on the query, which is used as input for the policy matching alongside the other attributes (see Definitions 7 and 8).

```sql
SELECT t1.column1 AS c1,
       t2.column2 AS c2
FROM db.table1 AS t1,
     db.table2 AS t2
WHERE c1 IS NOT NULL;
```

Listing 5.2: SQL with resolved references by introducing aliases.

5.2 MATCH INPUT WITH POLICIES

At this point, we assume to have all of the input attributes, including the tags. Some attributes may require internal metadata before we can add them to the input attributes, so these should be retrieved at this point as well. Thus, from now on we assume that the input will contain all attribute values so that the policy matching can commence. We consider matching policies with the input at two levels: per policy and per attribute.

5.2.1 *Policy Match Evaluation*

The basic matching algorithm for matching policies on the input is shown in Algorithm 1. For the pseudocode used in this chapter we will use the notation from Chapter 4.

Algorithm 1: Pseudocode for policy matching

```
Data: Set of all policies $P$, set of input attributes $I$
Result: Set of applicable policies $AP$
1  for $p \in P$ do
2      if $I.atIds$ does not contain all $p.C.atIds$ then
3          continue
4      end
5      $policyMatch \leftarrow true$
6      $policyAttrs \leftarrow$ group $p.C$ by $atId$
7      for $(atId, atVals)$ in $policyAttrs$ do
8          $inputAtVals \leftarrow I[atId].atVals$
9          if $evaluate(inputAtVals, atVals) = false$ then
10             $policyMatch \leftarrow false$
11         end
12     end
13     if $policyMatch$ then
14         $AP \leftarrow AP \cup p$
15     end
16 end
17 return $AP$
```

Simply put, the algorithm loops over all policies and checks whether they apply on the set of input attributes (i.e. the attribute values related to or derived from the request). Because all of our proposed context attribute types have the property that the set of policy attribute values that will result in a match can be solely determined based on the input attribute values, we can make an important optimization. By only determining the set of allowable values once in advance, we reduce the evaluation of a single attribute value match to a simple $\mathcal{O}(1)$ lookup. This would not be possible in case the attribute evaluation requires other data than just the input attribute values, e.g. through an external service. However, in practice the implementation allows for a hybrid approach because this evaluation is implemented per attribute type, see Chapter 6. The optimized version based on this property is shown in Algorithm 2.

Algorithm 2: Optimized version of pseudocode for policy matching

```
Data: Set of all policies $P$, set of input attributes $I$
Result: Set of applicable policies $AP$
1  for $i \in I$ do
2      $allowableValues_{i.atId} \leftarrow getAllowableValues(i)$
3  end
4  for $p \in P$ do
5      if $I.atIds$ does not contain all $p.C.atIds$ then
6          continue
7      end
8      $policyMatch \leftarrow true$
9      $policyAttrs \leftarrow$ group $p.C$ by $atId$
10     for $(atId, atVals)$ in $policyAttrs$ do
11         $attrMatch \leftarrow false$
12         for $atVal$ in $atVals$ do
13             if $allowableValues_{atId}$ contains $atVal$ then
14                 $attrMatch \leftarrow true$
15             end
16         end
17         if $attrMatch = false$ then
18             $policyMatch \leftarrow false$
19         end
20     end
21     if $policyMatch$ then
22         $AP \leftarrow AP \cup p$
23     end
24 end
25 return $AP$
```

5.2.2 *Attribute Match Evaluation*

What constitutes a match at the attribute level can differ per attribute type. With our proposed set of context attributes, we have two main attribute types: *hierarchical* and *enum*. Evaluating a match for the enum type is trivial: if the set of registered enum values contains the value that we are checking, it is a match. This can obviously be implemented in $\mathcal{O}(1)$.

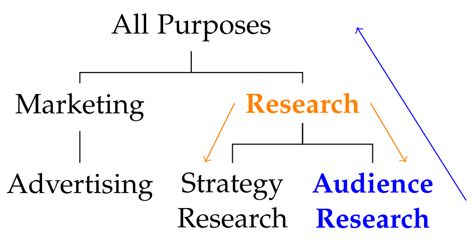


Figure 5.1: Example of a hierarchical attribute. Blue represents the input, orange represents the policy.

Checking for a match for a hierarchical attribute is also fairly intuitive. If a policy references a value from an hierarchical attribute, it matches that value and all values below it in the tree. Thus, checking for a match involves looking up the input value in the tree, and seeing if any of the policy's values for that same attribute is a parent. The set of all values that result in a match (assuming OR-semantics) can be built by looking up all of the input values and their parents and adding them to a set.

This is illustrated in Figure 5.1. Assume we have a policy that references the purpose 'Research'. The orange arrows show that this purpose and its two children are covered by the policy. Now say a data transformation is submitted for purpose 'Audience Research'. We then search for this purpose in the tree and find its parents, which is shown by the blue node and arrow in the tree (this only has to be done once!). Because there is an overlap between the orange and blue paths in the tree, we know that this policy attribute value matches the input attribute value for $\{Research, Audience\ Research\}$.

5.3 POLICY DECISIONS & CONFLICTS

In the set of matched policies, there can be policies with different decisions. In this section we introduce the different possible decisions, how they can conflict and how we resolve these conflicts.

5.3.1 *Decisions*

As we have seen, the system supports three different policy decisions: ALLOW, DENY, NON-DECIDING. However, the result of evaluating a policy can actually have four different states: ALLOW, DENY, NON-DECIDING and INDETERMINATE. Note that ALLOW in our system actually implies ALLOW *if requirements can be enforced*, which shows the need for this new INDETERMINATE state. The INDETERMINATE result will be returned if there is a problem with evaluating the policy, for example if not all requirements can be evaluated or enforced. The exact behavior can be defined per attribute, as various situations may warrant an INDETERMINATE. This result implies that a transformation will be denied (even if the original decision was ALLOW) and that the user should be notified about the problem. For example, if a particular tag used in the *require* section could not be resolved in the current query, the result will be INDETERMINATE. This indicates that the submitted transformation needs modification before it can be allowed. Note that yet another set of final decisions apply when we have evaluated all applicable policies: ALLOW, DENY, INDETERMINATE. The final outcome of evaluating all applicable policies can never be NON-DECIDING because if no applicable policies make a decision, the system-wide default decision will be returned.

5.3.2 *Context Conflicts*

Consequentially, if multiple policies apply on the input, there can be conflicts between policies with different decisions and overlapping contexts. We define a conflict as a policy that makes the opposite decision of another policy while having a (partially) overlapping context that applies on a particular input. Note that we only consider ALLOW and DENY decisions to be able able to conflict. NON-DECIDING are by definition not able to conflict. We suggest to have INDETERMINATE always take precedence over the other decisions and thus always result in an INDETERMINATE outcome. This is a strict approach for the sake of clarity and security. Furthermore, we only have to look at the context of the policy, not its requirements since requirements can only be enforced if the policy decision is ALLOW. Conflicts between requirements will be considered separately as they are semantically different from context conflicts.

Other more lenient approaches to deal with INDETERMINATE *are possible, depending on the requirements and strictness of the organization using the system. See Chapter 8.*

Since a policy context consists of a conjunction of different attributes, conflicts can be quite complex. A (partial) overlap is considered to occur if any of the individual context attributes of two policies overlap together with the input. Suppose we have two policies with purposes $P_1 = \{Advertising, Research\}$ and $P_2 = \{Advertising, Reporting\}$, and suppose P_1 makes decision ALLOW while P_2 makes decision DENY. If we then receive a transformation for purpose 'Advertising', both policies apply and make conflicting decisions. However, if this transformation has purpose 'Research', there would not be a conflict because P_2 does not apply in the first place.

Note that the tag attribute requires some special attention. The decision was made to also make this a hierarchical attribute. This has certain consequences for how policies and conflicts are interpreted by someone who runs into a conflict. It is important to recognize the distinction between two interpretations of overlap:

1. Overlap in terms of tag trees. This is the same notion of overlap as for the other hierarchical attributes: when two policies refer to overlapping branches in the value tree.

2. Overlap in terms of the selected data. One could also look at what data is overlapping (after resolving the tags). For example, if we have two policies that select columns $\{a, b, c\}$ and $\{a, b, d\}$ from the same table, these policies have an overlap on columns a and b.

The former interpretation of overlap was chosen because it is similar to how all other hierarchical attributes work. Moreover, this approach puts the emphasis on the *meaning* of the data and not the actual data itself, which is consistent with our declarative approach towards policy authoring.

Our proposed conflict detection and resolution strategy again rely on the property of our policy model that most attributes have a hierarchical basis. Because there is an inherent *scope* in these hierarchical attribute values, we can relatively easily resolve the vast majority of these conflicts by taking these scopes into account. Attributes like tags and geolocations are all structured in trees, meaning that we can take the most specific context to take precedence. In the rare case that two policies live in the exact same scope, the system will return the system-wide default decision for the sake of clarity and security. In the following subsections we will first discuss how conflict detection and resolution works for individual attributes of the two main context attribute types, and then introduce how this gets combined in a conflict resolution algorithm between policies.

Again, various improvements to this basic approach towards conflict resolution edge cases can be made, see Chapter 8.

5.3.2.1 *Hierarchical Attributes*

To detect and resolve conflicts between values of the same hierarchical attribute, we look between each value from both sets of attribute values at what attribute value is the most specifically applicable to the input. Algorithm 3 lists pseudocode for how this algorithm works. We assume that previously it has already been checked what branches are selected by the input values (see Section 5.2.2). For each branch, we then compare all pairs to see what values are more specific (if they can be found in the branch). By counting the amount of times either a value from policy A or B is more specific, we can decide which policy is overall the most specific. Note that in Line 11, we need to check whether a policy already has listed an attribute value before in the same branch. If policies were allowed to have multiple values within the same branch (e.g. 'Research' and 'Audience Research' in Figure 5.1), one can cheat the system by adding many values within the same branch. Furthermore, multiple values in the same branch do not make logical sense in the first place because the hierarchical definition makes this redundant.

As an example we can take the values from Figure 5.1 again. Take the (user-provided) purpose value 'Audience Research'. Then, $branch = [Audience\ Research, Research, All\ Purposes]$, as indicated by the blue arrow. Thus, if the index of a value from policy A is lower in this list than the index of a value from policy B, A is more specific.

5.3.2.2 *Enum Attributes*

For enum attributes, the conflict resolution strategy is simpler but also less powerful. We simply take the intersection of the attribute values from the input with the attribute values from policies A and B. If the number of values in the residual list from A is larger than the residual list from B, we take A to be more specific. This is quite a rudimentary approach and therefore hierarchical attributes are clearly a better choice if applicable.

5.3.3 *Conflict Resolution*

Now that we know how to detect and resolve conflicts for the two attribute types, we can introduce an algorithm for detecting and resolving

Algorithm 3: Conflict resolution pseudocode for two sets of hierarchical attribute values.

```
Data: Set of attribute values for policy A $A$, set of attribute values
      for policy B $B$, set of branches covered by input $Branches$
Result: 1 if $A$ is more specific, 2 if $B$ is more specific, 0 if one is not
        more specific than the other, $-1$ in other cases
$ACount, BCount \leftarrow 0$
for $branch \in Branches$ do
    $AInBranch, BInBranch \leftarrow false$
    for $a \in A$ do
        if $a \notin branch$ then
            continue
        end
        for $b \in B$ do
            if $b \notin branch$ then
                continue
            else if $AInBranch$ or $BInBranch = true$ then
                illegal policy
            else if $a$ is deeper in $branch$ than $b$ then
                $ACount \leftarrow ACount + 1$
                $AInBranch \leftarrow true$
            else if $b$ is deeper in $branch$ than $a$ then
                $BCount \leftarrow BCount + 1$
                $BInBranch \leftarrow true$
        end
    end
end
return 1 if $ACount > BCount$, 2 if $ACount < BCount$, 0 if
    $ACount = BCount$, otherwise $-1$
```

conflicts in a set of policies. Algorithm 4 lists the pseudocode for this algorithm.

We have to compare the contexts of all policy combinations by calling the `checkConflicts` method on the individual attributes, which are implemented at the attribute type level as described in Sections 5.3.2.1 and 5.3.2.2. Like with the individual attribute values, a pair of policies are scored based on how many of the attributes of one policy are more specific than the other policy. The less specific policy of the two gets removed from the set of policies. In case the two policies have the same scope, the policy that conforms to the system's default decision takes precedence. Thus, the policy with the opposite decision of the system's default decision gets removed from the set of policies.

Note that the complexity of this algorithm is exponential because of the nested loop to check between all policies for conflicts. However, this algorithm should be run after the initial policy matching already has been done, meaning that P should only contain policies that actually apply on the input, which is not expected to grow very large where this becomes a problem. Furthermore, the `checkConflicts` algorithm can have a high complexity depending on the implemented attribute types.

Still, a maintainable set of attribute values is also not expected to become large enough for this complexity to become a problem.

Algorithm 4: Conflict resolution pseudocode for a set of policies.

```
Data: Set of policies $P$, set of input attributes $I$, system default
      decision $d$
Result: Set of policies with resolved conflicts $R$ where $R \subseteq P$
$R \leftarrow P$
for $P_a$ at index $a$ from $0$ until $size(P)$ do
    if $P_a.decision \neq$ ALLOW or DENY then
        continue
    end
    for $P_b$ at index $b$ from $a+1$ until $size(P)$ do
        if $P_a$ does not have opposite decision of $P_b$ then
            continue
        end
        $AScore, BScore \leftarrow 0$
        for $(inputId, inputVals) \in I$ do
            $atValsA \leftarrow P_a.C[inputId]$
            $atValsB \leftarrow P_b.C[inputId]$
            if $atValsA$ or $atValsB = \emptyset$ then
                continue
            else
                $precedence \leftarrow$
                  $checkConflicts(inputVals, atValsA, atValsB)$
                if $precedence = 1$ then
                    $AScore \leftarrow AScore + 1$
                else if $precedence = 2$ then
                    $BScore \leftarrow BScore + 1$
        end
        if $AScore = BScore$ then
            if $P_a.decision = d$ then
                $R \leftarrow R - \{P_b\}$
            else if $P_b.decision = d$ then
                $R \leftarrow R - \{P_a\}$
        else if $AScore > BScore$ then
            $R \leftarrow R - \{P_b\}$
        else if $BScore > AScore$ then
            $R \leftarrow R - \{P_a\}$
    end
end
return $R$
```

5.4 evaluate requirements

Evaluating the requirement attributes involves checking whether the requirements are applicable, generating the outcomes and combining outcomes (and resolving conflicts if applicable). The basic algorithm is listed in Algorithm 5. Policies that do not have requirements can just be added *as is* to the resulting set of evaluated policies. Also, note that DENY and INDETERMINATE policies cannot list requirements. In case a policy does have requirements, we call the *evaluateRequirement* method for that policy. Similar to policy matching, this is implemented at the attribute level as explained in Section 4.2.

Algorithm 5: Pseudocode for requirement evaluation and outcomes generation for individual policies in a set of policies.

```
Data: Set of policies $P$
Result: Set of evaluated policies with evaluated requirements $E$
$E \leftarrow \emptyset$
for $p \in P$ do
    if $p.R = \emptyset$ then
        $E \leftarrow E \cup p$
    else if $p.R \neq \emptyset$ and $p.decision$ = DENY or INDETERMINATE then
        illegal policy
    else
        $outcomes \leftarrow \emptyset$
        $indeterminate \leftarrow false$
        for $(atId, atVals) \in p.R$ do
            $res \leftarrow evaluateRequirement(atVals)$
            if $res$ = exception then
                $indeterminate \leftarrow true$
            else
                $outcomes \leftarrow (atId, res)$
        end
        $p_{eval} \leftarrow p$
        $p_{eval}.outcomes \leftarrow outcomes$
        if $indeterminate = true$ then
            $p_{eval}.decision \leftarrow$ INDETERMINATE
        end
        $E \leftarrow E \cup p_{eval}$
end
return $E$
```

For requirements, we propose three main attribute types: *enum*, *hierarchical* and *tag reference*. The first two are shared with the context attributes, while with the latter type, attribute values refer to tag names which can then be resolved to see whether they apply on the query so that something can be enforced on this data. The implementation of *evaluateRequirement* depends on the attribute type (or even the individual attribute). For the *enum* type, evaluating a value simply involves checking whether the value is valid. For the *tag reference* type, we not only check if the name belongs to a valid tag, but also resolve all data references in the query that the tag belongs to. Lastly, the result of evaluating a *hierarchy* type is a flattened list of the evaluated value and its children in the tree. This is because if we require a particular value from the tree, its children are also expected to conform to this requirement. An example of this is that if data location 'Europe' is required by a policy, data can also live in data stores with locations that are children of 'Europe' in the tree, such as countries like 'The Netherlands'.

5.5 GENERATE FINAL OUTPUT

The final output that QOMPLIANCE returns to the user or the data processor should consist of the following elements:

- The policies that were applied on the input, along with their individual decisions and possible explanations (for example why a policy evaluated to INDETERMINATE).
- The set of outcomes that resulted from the evaluation of the requirements.
- A final decision based on the combination of all policy decisions.
- The timestamp for when the final result was generated.
- A signature as security measure.
- The validated SQL.

Note that if the SQL was not valid in the first place, the system should respond with a validation error instead of the above response. Most of these elements are trivial, but we discuss finding the final outcomes, making the final decision and signing the output in more detail.

5.5.1 *Final Outcomes*

Determining the final outcomes is again implemented at the attribute level with the *generateFinalOutcomes* function first described in Chapter 4. The outcomes are the set of values that have to be enforced. Important to note is that requirement values have AND-semantics (as opposed to context attribute values), which is according to our definition that all requirements have to be applied, see Section 5.3. Outcomes have two meanings: they can either require the user to change something (which results in an INDETERMINATE decision), or require the system to enforce something and conditionally allow the transformation based on that assumption.

The semantics for the different outcomes differ between attributes, and it is up to the user-facing interface or data processing system to correctly interpret and process these requirements. The general outcomes generation algorithm is trivial and therefore we omit its pseudocode, but it simply involves calling the *generateFinalOutcomes* implementations for all requirements with a set of all evaluated requirement values.

However, "enforcing all requirements" can have different semantics per attribute. For *enum* requirements, we take the union of all values and the resulting set of values should all be enforced. E.g., if one policy requires HIPAA-compliant storage and another policy requires drive encryption, both of these requirements should be satisfied. For *hierarchical* requirements, we take the intersection of the flattened sets of evaluated requirement values selected by a policy to find the set of allowable values (that do not all have to apply!). E.g., if one policy requires data to be stored in Europe (and thus the evaluated requirement also contains countries like The Netherlands), and another policy requires data to be stored in The Netherlands, the intersection and thus the final outcome will only be The Netherlands. Thus, note that this result can be empty, meaning that two policies give conflicting requirements that cannot be resolved. This results in an INDETERMINATE final decision. For *tag reference* requirements, the semantics for outcomes and how to combine them will differ per attribute. If we take the 'without' attribute as an example, combining values involves taking the union of all resolved data references as the final set of data references that need to be excluded from the query.

5.5.2 *Final Decision*

After the individual decisions from policies have been considered, potential conflicts have been checked and resolved and requirements have been evaluated, the decisions from policies can be combined to reach a final decision. Thus, at this point we assume that the set of applicable policies has been reduced to only contain the more specific policies (whenever applicable). The final decision is made by taking the logical conjunction between all policies in the set of applicable policies. We assume a conjunction relation because this most closely resembles how legislation would work, e.g., usually all articles in an act should be enforced [31]. Formally, with d_i denoting an individual policy decision of a policy in the set of applicable policies and n denoting the total number of applicable policies:

$$d_{res} = d_0 \wedge \cdots \wedge d_n \tag{5.1}$$

Furthermore, we assume the following rules:

$$allow \wedge allow = allow \tag{5.2}$$
$$allow \wedge deny = deny \tag{5.3}$$
$$deny \wedge deny = deny \tag{5.4}$$
$$allow \wedge nondeciding = allow \tag{5.5}$$
$$deny \wedge nondeciding = deny \tag{5.6}$$
$$(allow \vee deny \vee nondeciding) \wedge indeterminate = indeterminate \tag{5.7}$$

5.5.3 *Signing the Output*

At this point, a final decision has been made and the output has been generated. Thus, the system can guarantee that at this point in time the submitted transformation has been validated and has resulted in a decision, plus any additional outcomes if applicable. This result is ready to be ingested by other systems, such as a user-facing UI or an ETL system that will actually execute the transformation while taking the outcomes into account. Adding a cryptographic signature to the output can allow someone to check that the validation decision and outcome was really generated by our system. It can also prevent the output from being tampered with along the way. In case the signature is invalid or absent when parsing the output from our system, it should refuse to perform any further action. A timestamp should be included in the output, which enables someone to demonstrate that at that particular time the transformation was authorized, which can be important for audit purposes.

A signature can be generated by using a public key cryptosystem [35], like RSA. By first encrypting (the hash of) the output using the system's private key, anyone can then decrypt the signature using the system's public key to verify that the message indeed came from the owner of the corresponding private key.

6 IMPLEMENTATION

For evaluating the proposed system, we have built a reference implementation that demonstrates the proposed algorithms and features. The implementation is written in Kotlin[1], which runs on the JVM (version 17). Appendix A contains more information about the source code. In this section we discuss notable details and challenges that warrant more in-depth explanations.

6.1 GENERAL ARCHITECTURE

The implementation consists of two main components which are implemented separately because of their different sets of responsibilities: the *data manager* and the *compliance checker*. The data manager is responsible for the PEP and PIP (as depicted in Figure 3.2) while the compliance checker is comparable to the Policy Decision Point. Furthermore, the project contains a utilities library for shared logic.

These components are built using a microservice approach where these two components can run as standalone services that expose a REST API using Spring[2]. Apart from the separation of concerns that this approach provides, it also allows for scaling the individual components. The data manager is responsible for handling the storing, querying and modifying of policies, schema information for the governed data and additional attribute metadata. Its implementation is fairly trivial, with simple CRUD operations on the database being its main responsibility. Furthermore, the data manager contains the logic for parsing the YAML policy language implementation that is described later in this chapter. The data manager's API has three consumers: the compliance checker, user-facing frontends and the data processor such as an ETL platform.

The compliance checker is solely responsible for handling validation requests which will be submitted by a user-facing frontend once the user submits a query for a pre-check, or by the PEP for a final decision. The step-by-step architecture for both modules is depicted in Figure 6.1.

The reference implementation uses a PostgreSQL database for storing the policies, schema information and additional metadata for the proposed attributes. The database model is trivially based on the universal data model from Chapter 4. The entity-relationship diagram for the database can be found in Figure 6.2.

Submitting a validation request is as simple as submitting a POST request to the validation endpoint from the compliance checker. A request comprises of the SQL defining the transformation and any additional external attributes. By external attributes we mean attributes that have to be provided by an external system such as the purpose for a transformation, whereas with internal attributes the values can be derived from

1 https://kotlinlang.org
2 https://spring.io

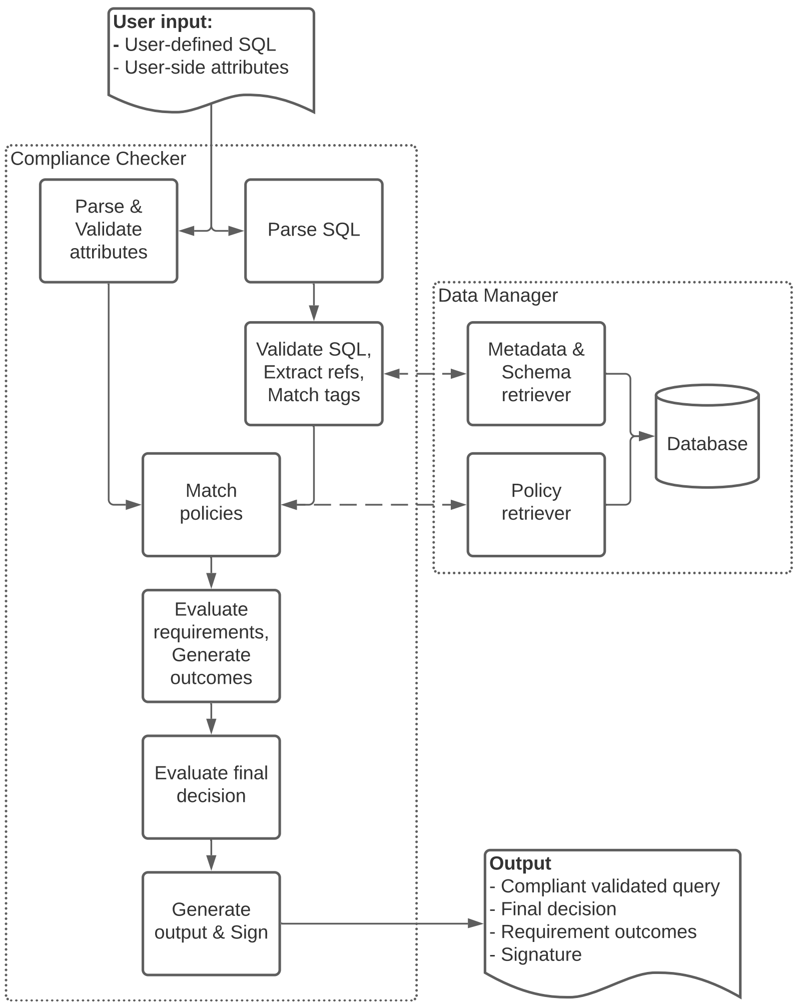


Figure 6.1: System design of the components and processing flow.

the metadata such as the geolocation of the data being addressed by the SQL. An example JSON request body is shown in Listing 6.1.

```
"sql": "SELECT * FROM \"DB0\".\"financial_data\"",
"attributes": {
    "purpose": ["Research"]
}
```

Listing 6.1: Example JSON body for a validation POST request.

6.2 SQL

For parsing and validating the SQL we use Apache Calcite[3], a framework for building databases while not managing the data or metadata itself. Its many features include a SQL parser and a relational algebra API which are especially useful for our purposes. Our system passes the schema

3 https://calcite.apache.org

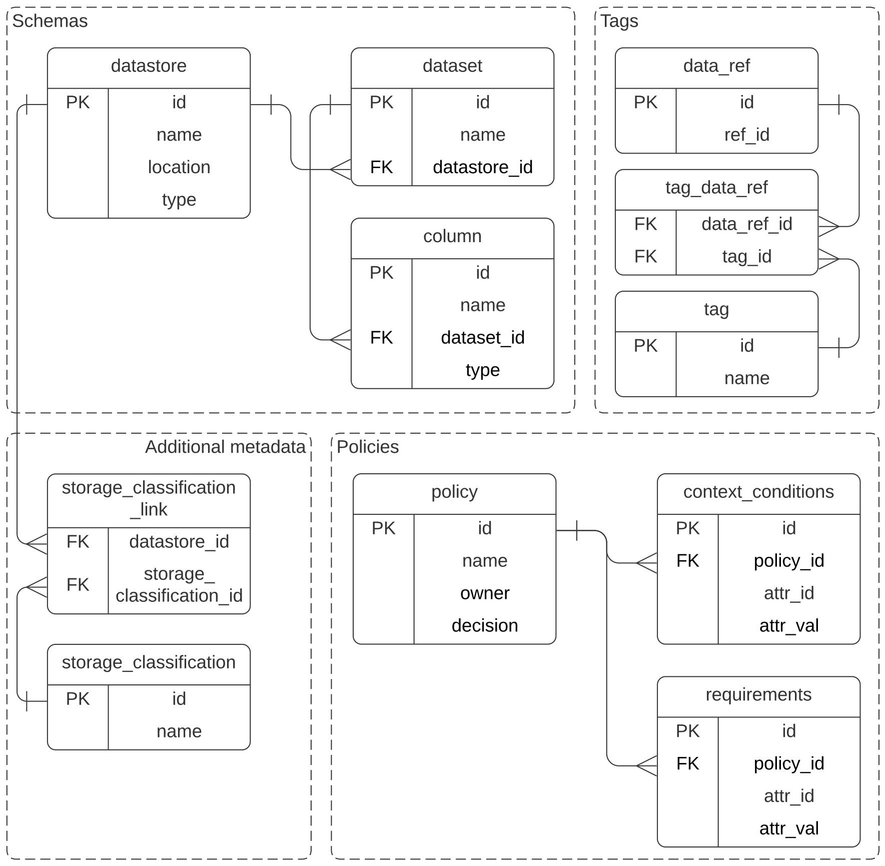


Figure 6.2: Entity-Relationship Diagram of the database for storing policies and (schema) metadata, based on the universal data model from Chapter 4.

information of the governed data to Calcite, which in turn can use these schemas to validate whether a SQL query conforms to this schema. By configuring Calcite to add aliases and expand data references as discussed in Section 5.1, our implementation can then resolve these aliases to extract all data identifiers from the query. An additional benefit of Calcite is that it has support for a large variety of SQL dialects, including ANSI and many RDBMSs but also for data processing systems like Hive, Spark and Presto. This is especially beneficial for our use case because it makes our service independent of the data processing system.

6.3 EXTENSIBILITY

Because this is only an implementation for demonstration purposes with carefully selected but still basic attributes, a design goal was to build the system in an extensible manner as mentioned in Aspects 3 and 4. By making use of object-oriented design patterns, the system defines interfaces that can easily be overridden to define additional attributes. Each attribute is its own class which inherits from two interfaces: what attribute it is (i.e., context or requirement) and its attribute type (e.g. hierarchical). The latter is actually implemented as an abstract class so that it can contain certain implementations for methods defined by the policy role. The interfaces for `ContextAttribute` and `RequirementAttribute` contain the methods from Chapter 4 that are required to be implemented for a functional attribute. The `AttributeType` abstract classes can then provide implementations for these functions, and can specify additional abstract functions and properties that need to be implemented by the actual attribute classes. Figure 6.3 gives a simplified illustration of this class hierarchy, the full version of which can be found in Figure B.1. Note that we only show two example types and attributes for clarity (indicated by the dots).

In this figure, the `DataLocationType` implements `ContextAttribute` and `HierarchyAttributeType` and thus only has to implement where to get the attribute value tree for it to become a fully functional attribute. In the case of the `WithoutRequirementAttribute`, the attribute itself has to implement more because it does some operations that are unique to that particular attribute (e.g. extracting the necessary information from the SQL). Note that, in the full version in Figure B.1, we also show a `SqlProcessingAttribute` interface that requires an extension of `evaluateRequirement` with an additional parameter for a `SqlProcessor`. This class contains the SQL parsing and processing logic that can be passed around to attributes and other classes that need to analyze the SQL in some way.

Attribute classes are then registered and matched using a factory pattern. A factory allows for easy matching between the attribute keys as used in the YAML and the corresponding attribute class implementations in an extensible way. Our current implementation just stores sets and hierarchies of attribute values at the application level. Thus, at the time of writing there is no interface to add values or tags. However, this can be trivially added at a later point in time.

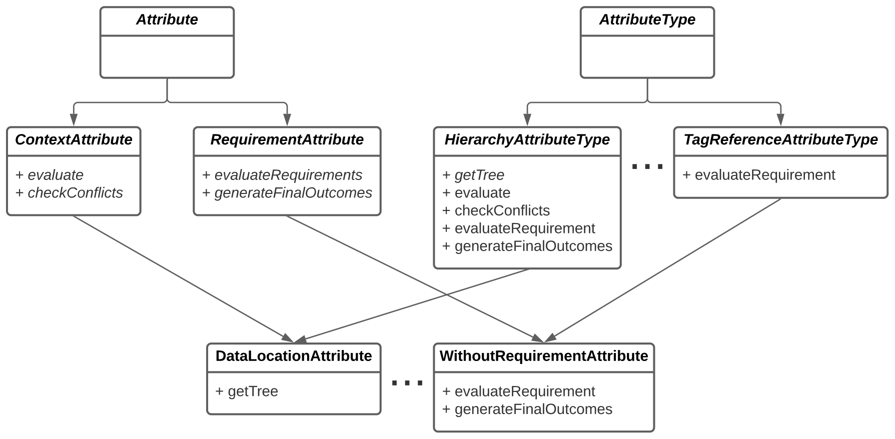


Figure 6.3: Simplified version of class diagram representing how attributes are implemented.

6.4 YAML POLICY LANGUAGE IMPLEMENTATION

To be able to easily write and process policies using the system's policy model (discussed in Chapter 4), we present a YAML implementation of the policy model. This implementation serves as a policy language which closely resembles the structure of the universal policy model. The language is based on YAML[4], a human-friendly data serialization language with minimal syntax. Examples of YAML-based policies can be found in Chapter 7. The entire specification of the YAML-based policy language can be found in Appendix C.

The decision for YAML was made because it strikes a balance between human-readability and machine-readability. A disadvantage of YAML is that its syntax may be somewhat hard to learn for someone unfamiliar with these kinds of languages. However, because of our relatively simple policy model, all attributes are as simple as a key for the attribute name with a list of attribute values. This decision also inherently requires us to adapt the policy language to fit the YAML model and syntax as well as possible. A policy language even closer to natural language would be helpful for the comprehension of policies to someone unfamiliar with the policy language, but this would require more extensive design which is out of scope for this research.

6.5 SIGNATURE

After the final output has been generated, we calculate a cryptographic signature over the JSON representation of the output. However, because many different representations can result in the same valid JSON, we need a way to consistently get to the same JSON representation. Otherwise, calculating a hash will be hard to repeat in other places than our specific implementation. Therefore, our implementation canonicalizes the JSON according to RFC8785 by Rundgren, Jordan, and Erdtman [36]. The canonicalized JSON is then hashed using SHA-256, which in turn gets encrypted using an RSA-2048 private key to create the signature. For demonstration purposes, our implementation generates the key pair in memory and logs the public key. However, a production implementation should use proper key management.

6.6 DATA GENERATOR & EXPERIMENT SCRIPT

The implementation includes a data generator that can generate schemas, tags, conditions, requirements and policies. The attribute values for conditions and requirements are retrieved from a default implementation of the different attribute value trees and enums. Schemas can either be generated using a predefined list of names for data stores, datasets and columns (that is randomly extended if necessary), or using the schema from the TPC-W database benchmark [29]. We use these predefined schemas so that we can use a fixed set of queries that are always valid. Furthermore, we use a fixed set of tags that only reference a fixed set of data stores. This means that if we query a different data store, an entirely different set of policies will apply. Policies are generated with a random set of context and requirement attributes chosen from pre-defined values so that the evaluation of the attributes itself is consistent across different tests. Different parameters can be set that influence the processing time

4 https://yaml.org

of a request, which will be explained in Chapter 7. For running the experiments using the data generator, we have built an accompanying Python script that runs the experiments with the appropriate variables.

7 EVALUATION

We evaluate our system design and compliance evaluation algorithms in three ways: by means of a use case, by means of a qualitative comparison based on the comparison framework from Section 2.3 and with a few empirical experiments.

7.1 USE CASE

In this section we discuss a qualitative evaluation of our design based on a use case. Qars is an imaginary car manufacturer from The Netherlands. They design, build and sell smart connected cars across Europe and they have just entered the United States market. Their cars can communicate all kinds of data points about their status, which Qars wants to use to detect problems and to improve their products.

When Qars was preparing to start selling their cars in the US, the legal team advised that it would be best to keep the data from cars that belong to US customers in a database in the US because of the new jurisdiction. However, these different jurisdictions introduce some challenges for Qars. Ideally, Qars's management wants to get insights over their entire fleet of cars, but their lawyers are really adamant about adhering to the regulations of the different jurisdictions. This is where Qompliance comes in.

First, we look at what data Qars is dealing with. Qars has two databases, one in The Netherlands called `qars-nl` and one in the United States called `qars-us`. Qars uses the schema shown in Figure 7.1 in both of these data stores. They have a `cars` dataset that contains unique identifiers for the cars called vehicle identification number (VIN) and other information such as its configuration. They have a `customers` dataset that contains customer information because car owners can create an account to connect to their car and which is considered to be PII by the legal department. This dataset contains a reference to the car that customers own. Furthermore, cars log their vitals (e.g. engine status, tire pressure, etc.) to a `car_vitals` table and events (e.g. seatbelt fastened, doors unlocked, etc.) to a `car_events` table. Entries in these tables also refer to the car that they belong to.

When the legal department was asked to come up with requirements for the data, they came up with the following list:

1. PII has to reside on an encrypted datastore.
2. PII has to stay in the country where it is generated at all times.
3. No one in the company can access car data, except for accident investigation by investigator employees.
4. Data scientists and engineers can also access car data for product improvement but without the VIN.

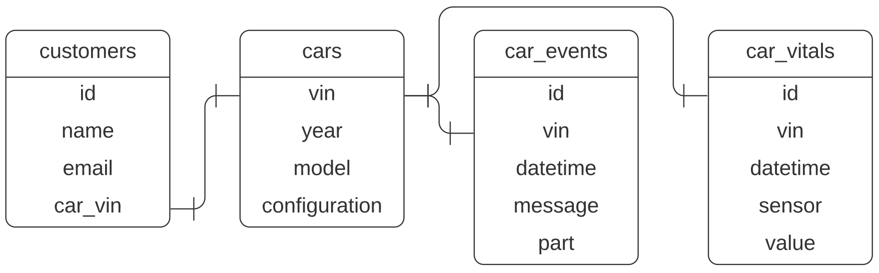


Figure 7.1: Database schema used by Qars in data stores `qars-nl` and `qars-us`.

7.1.1 *Defining Policies*

We now demonstrate how these requirements can be converted into policies in our policy model using the reference implementation. We will use the YAML policy representation introduced in Section 6.4 and appendix C for simplicity and because we can directly input them into the system. Note that Qars assumes allow-by-default.

```
name: PII has to reside on a disk-encrypted datastore
context:
  tags:
    - PII
require:
  storage-classification:
    - DiskEncrypted
```

Listing 7.1: PII has to reside on an encrypted datastore.

The first requirement is implemented in Listing 7.1, which reads as: in a context where PII data is referenced, it is required that data is stored on a data store with the `DiskEncrypted` classification.

```
name: EU PII stays in EU
context:
  tags:
    - PII
  data-location:
    - EU
require:
  data-location:
    - EU
```

(a) Policy that requires that PII that is currently in the EU stays in the EU.

```
name: US PII stays in US
context:
  tags:
    - PII
  data-location:
    - United States
require:
  data-location:
    - United States
```

(b) Policy that requires that PII that is currently in the US stays in the US.

Listing 7.2: PII has to stay in the country where it is generated at all times.

Listing 7.2 contains two policies to implement the second requirement. Our implementation currently does not support an explicit directive for ‘keeping data in the same place’. Thus, we have to write explicit policies for the two regions that we have, which are trivial to understand.

The YAML representation of the third requirement is shown in Listing 7.3. For this, we also have to implement two policies. Because Qars

```yaml
name: No one can access Car Data
context:
  tags:
    - Car Data
decision: deny
```

(a) Policy that denies all access to Car Data.

```yaml
name: Car data can be accessed for accident investigation
context:
  tags:
    - Car Data
  purposes:
    - Accident Investigation
  roles:
    - Investigator
decision: allow
```

(b) Policy that allows Car Data access for Accident Investigation.

Listing 7.3: No one in the company can access car data, except for accident investigation by investigator employees.

assumes allow-by-default, we have to define two policies. Therefore, we first write Listing 7.3a to make sure that we deny all access to Car Data. We can then make an exception by writing a more specific policy, which demonstrates the usefulness of the conflict resolution algorithm. Listing 7.3b is more specific because it puts more conditions on the context by adding more attributes. Only if the system can match all of these attributes on the input, the policy in Listing 7.3b will be applied instead and thus allow the Car Data access.

```yaml
name: Access car data for product improvement without VIN
context:
  tags:
    - Car Data
  roles:
    - Data Scientist
    - Engineer
  purposes:
    - Product Improvement
decision: allow
require:
  without:
    - VIN
```

Listing 7.4: Data scientists and engineers can also access car data for product improvement but without the VIN.

For the last requirement, we can write a policy as shown in Listing 7.4. This policy reads as: in a context where either a Data Scientist or an Engineer is accessing Car Data for the purpose of Product Improvement, allow the request given that the query does not in any way reference a VIN.

Understanding and writing these policies only requires very limited knowledge about the underlying data. The author or reader of the policies just has to have an understanding about the kind of data that is processed. Someone still has to do some manual work to make sure that the tags are right, or at least automate assigning these tags, but from that point the policies become very high-level and declarative. Therefore, this satisfies the goal that we have set out in Objective 4.

7.1.2 *Queries & Compliance Checking*

Now that Qars has implemented their first set of policies, they can start to submit their first queries for validation and compliance checking. In this section we will demonstrate how queries get analyzed and policies get matched to satisfy Objective 6. The first query that they submit looks like `SELECT * FROM qars-nl.customers`. This returns the output that can be seen in Listing 7.5. This response is entirely according to the system design as explained in Section 5.5. Note that two policies have been applied because the query addressed PII (the `customers` table) in an EU datastore. In fact, the metadata of the `qars-nl` indicates that it is located in the Netherlands, which is a good demonstration of hierarchical attributes. Because The Netherlands is part of the EU in the hierarchy, it is within the context of that particular policy. Furthermore, the outcomes of the policy in Listing 7.2a show how hierarchical requirements work: the `EU` policy value is resolved to the value itself and all children in the tree. These are both NON-DECIDING policies, meaning that the default access decision was used for the final decision of this check.

A data scientist at Qars now wonders if they can trick the system and query data from both the US and the EU. However, in this case Listing 7.2b will also apply and the system returns INDETERMINATE. This is because when combining the requirements from both location policies, there is no overlapping subtree meaning that there are no possible locations to execute or store this result.

Next, we consider the policies for the third requirement in Listing 7.3. Again, note that Listing 7.3a will apply any time any data tagged 'Car Data' is referenced in a query, also at lower levels in the query. For example, `SELECT model FROM qars-nl.cars` will also match 'Car Data' even though only a single column of this tagged dataset is referenced. Furthermore, this also applies when combined with other data. If we take a query: `SELECT vin, email FROM qars-nl.cars, qars-nl.customers`, the columns get resolved to the datasets that they belong to. Then, the system outputs three policies: 'EU PII stays in EU', 'PII has to reside on a disk-encrypted datastore' and 'No one can access Car Data'. This request is denied because of the last policy of the three.

However, now we set the role to 'Investigator' and the purpose for the transformation to 'Accident Investigation'. This demonstrates how the more specific policy takes precedence in our system, because now Listing 7.3b is returned instead and thus the query is allowed.

We have seen tags at the dataset level. Now we demonstrate tags at the column level. Remember that a tag can apply on any number of columns, datasets and data stores. This is why Qars has added a VIN tag to all

```
{
  "result": {
    "decision": "allow",
    "acceptablePolicies": [
      {
        "id": 1,
        "name": "EU PII stays in EU",
        "decision": "nondeciding",
        "contextConditions": [...],
        "requirements": [...],
        "outcomes": {
          "data-location": [
            "EU",
            "Netherlands",
            "Germany",
            "Belgium"
          ]
        },
        "evaluatedDecision": "nondeciding"
      },
      {
        "id": 3,
        "name": "PII has to reside on a disk-encrypted datastore",
        "decision": "nondeciding",
        "contextConditions": [...],
        "requirements": [...],
        "outcomes": {
          "storage-classification": [
            "DiskEncrypted"
          ]
        },
        "evaluatedDecision": "nondeciding"
      }
    ],
    "reason": "No deciding policies, default decision allow was used",
    "outcomes": {
      "data-location": [
        "EU",
        "Netherlands",
        "Germany",
        "Belgium"
      ],
      "storage-classification": [
        "DiskEncrypted"
      ]
    },
    "validatedAt": "2022-01-20T12:00:00.000000Z"
  },
  "validatedSql": "SELECT ...\nFROM \"qars-nl\".\"customers\"",
  "signature": "..."
}
```

Listing 7.5: Example JSON response for a compliance check. Some values have been collapsed for brevity as indicated by the dots.

columns containing a VIN, which is useful because the VIN is used as a (foreign) key in multiple tables. This allows Qompliance to detect queries that reference a VIN in any way, which is especially important because in the specific case of Listing 7.4 Car Data can only be used if the VIN is left out. Different examples of queries that reference a VIN are shown in Listing 7.6. In all cases except for Listing 7.6d the decision will be INDETERMINATE because the blocking 'without' requirement cannot be satisfied. Thus, the Data Scientist or Engineer submitting these queries should update their query until a VIN is not referenced in any way. Only then will the decision turn into an ALLOW.

```
SELECT *
FROM qars-nl.car_vitals
JOIN qars-nl.cars
  ON car_vitals.vin=cars.vin
```

(a) Illegal query because it references the VIN.

```
SELECT v.datetime, c.model
FROM qars-nl.car_vitals AS v
JOIN qars-nl.cars AS c
  ON car_vitals.vin=cars.vin
```

(b) Illegal query because even though a VIN is not in the output, it is still used in the join.

```
SELECT *
FROM qars-nl.car_vitals
```

(c) Illegal query because it references the VIN (through the asterisk).

```
SELECT id, datetime, sensor, value
FROM qars-nl.car_vitals
```

(d) Legal query because it does not reference VIN in any way.

Listing 7.6: Queries that demonstrate excluding a column.

7.1.3 *Extending the Policy Model*

At this point, Qars is very enthusiastic about the Qompliance reference implementation and they decide to properly implement it for their own purposes. However, they want an extra context attribute in their implementation, namely the clearance level of the user submitting the query. This can be implemented fairly quickly by introducing a new class `ClearanceAttribute` for this attribute that extends the `HierarchyAttributeType` and `ContextAttribute` interfaces. Furthermore, they have to register this class in the factory and in the policy parser in the data manager. Lastly, they have to define the value tree and override the `getTree` method to point to the tree. Although implementing this requires understanding of the system's code, Qars does not have a problem with this because they do not expect to extend this model often.

The attribute-value pairs just map like all other attributes to the model defined in Chapter 4. This demonstrates the extensibility of our policy model, which is an important part of Objective 3. Because of our decision to move more decision and processing logic to the application level, extending the policy model is a bit more involved than a language like XACML [32]. However, because of the simpler policy model and because extending the model is not regularly necessary, we think this is a fair compromise to make.

7.2 QUALITATIVE COMPARISON

In this section we reflect back on the design space framework that we introduced in Sections 2.3 and 3.1. We revisit the axes of comparison and give qualitative evidence to explain our positioning on these axes compared to other data-centric systems. Through this comparison, we confirm to what extent we have met the intended positioning. For comparison, we will use the same representative set of related approaches as used in Figure 3.1.

ASPECT 1, SYSTEM SCOPE: The systems we have surveyed for this research have widely differing scopes. This is inherent to the different intended applications of these systems. Whereas LEGALEASE [38] is specifically tailored towards privacy policies, the policies of Beedkar, Quiané-Ruiz, and Markl [3] are tailored towards low-level restrictions on cross-border data movements. And whereas Data Capsule's [46] policy model is intended to be able to encode high-level regulations like GDPR, the general purpose models of XACML and OPA are built for more complex decision logic.

QOMPLIANCE is somewhat in the middle of this spectrum. Its policy model is further away from natural language compared to LEGALEASE and by extension Data Capsules. However, it is higher-level than other systems like the work from Beedkar, Quiané-Ruiz, and Markl [3] that has very low-level policies (see the next aspect). Somewhat closer to the middle we find [24], who also mention that they attempt to bridge the gap between storage-level controls and regulations. Still, their approach relies on storage-level adaptations that we consider lower-level compared to QOMPLIANCE.

QOMPLIANCE is intended to be able to implement requirements close to regulations, but also lower level requirements on the data. This is enabled by the hierarchical approach and the direct link between tags and the governed data definition. Furthermore, NON-DECIDING policies is another important feature that allows for writing higher-level policies compared to systems that closely relate to storage-level controls. Lastly, its extensibility allows someone to steer the policy scope of the implementation.

ASPECT 2, POLICY ABSTRACTION LEVEL: The policy abstraction level is a major distinguishing factor for QOMPLIANCE. On the low-level end of the spectrum, we have approaches that include direct references to the schema of the governed data in the policies. These approaches are generally related to database access control or query processing. For example, in the work of Beedkar, Quiané-Ruiz, and Markl [3], a policy takes the following shape: '`SHIP` *attribute list* `FROM` *table* `TO` *location list* `WHERE` *condition list*'. On the other end of the spectrum we can find approaches like LEGALEASE/GROK by Sen et al. [38] and Wang et al. [46] that only rely on inferred abstract data types that are organized in lattices and without direct traceability to the governed data model.

QOMPLIANCE lives in between. Tags have a clear link to the schemas of the governed data and can apply at multiple levels of granularity. Moreover, our approach enjoys many of the same benefits as the systems with a high level of abstraction thanks to their hierarchical definition. Note that since general purpose approaches like XACML [32] and OPA can technically operate at any level of abstraction, we cannot directly compare them here.

ASPECT 3, POLICY COMPLEXITY: Systems and languages with a high policy complexity notably include XACML [32] and OPA/Rego. These languages have a large learning curve because of their many features and constructs that allow for writing complex yet expressive policies. On the other end of the spectrum we find languages like LEGALEASE [38] but also the simple query-like policies from Beedkar, Quiané-Ruiz, and Markl [3]. LEGALEASE has a low complexity because of its intuitive data types and clearly defined semantics that are intended to be close to the natural language constructs used in privacy policies. The query-like policies from Beedkar, Quiané-Ruiz, and Markl [3] are similarly intuitive because they are understandable by anyone who understands SQL and because they include very little language features. A language that is somewhat in between is the model from Wang et al. [46] that for example supports boolean operators between clauses.

Our policy model is a little more complex because authoring policies for this model requires a little more understanding of the policy processing that is not immediately clear at first sight (e.g., conflict resolution and attribute semantics) compared to the mentioned low-complexity approaches. Furthermore, our model is missing some constructs that would bring it closer to natural language such as negations and exceptions, see Section 8.1.4. Still, our policy model is confined to practically only consist of attribute-value pairs and decisions and thus there is very little structural difference between policies. Therefore, once the reader is familiar with the policy processing and the individual attribute semantics, all policies will be trivial to understand.

ASPECT 4, POLICY EXPRESSIVENESS: Languages like XACML [32] and OPA/Rego are highly expressive (but have high complexity as we have seen). Because they are very general purpose and can serve at a low level, one can draw a parallel to the field of programming languages and compare these policy languages to Assembly Language. They can be used to write very detailed policies, but can also serve as the foundation for other policy systems. Other data-centric policy systems trade some expressiveness in favor of lower complexity, by adding syntactic sugar and making more assumptions, somewhat comparable to higher-level programming languages.

One can look at expressiveness from various different viewpoints. For example, just like QOMPLIANCE, the policy model of LEGALEASE is quite simple with a very limited set of constructs. LEGALEASE does support some more advanced constructs compared to QOMPLIANCE such as exceptions and negations. However, QOMPLIANCE supports a wider range of attributes and explicitly intended to be extensible, which increases the expressive power of the policies. The Data Capsule system [46] even takes the policy model a step further compared to LEGALEASE (on which their design was based) by adding boolean operators. Deep Enforcement [24] and Compliant Geo-distributed Query Processing [3] only have very simple policy models with small sets of attributes and possibilities. Therefore, we consider QOMPLIANCE to live close to LEGALEASE and the Data Capsule system towards the middle of the expressivity spectrum.

ASPECT 5, ENFORCEMENT VS. AUDIT: Most systems that we have surveyed simply target enforcement without mentioning auditing as a requirement or a goal. The exception to this is LEGALEASE/GROK by Sen et al. [38], which takes a passive approach by retrospectively automatically auditing data flows. Although QOMPLIANCE has a clear focus on enforcement, we do not position the system entirely on the enforcement side of this axis for two reasons. Firstly, QOMPLIANCE is not solely responsible for all enforcement. Although any requests that do not conform to the decisions and requirements set by policies are denied, enforcement of certain requirements is left to the data processor (e.g., data location). Secondly, QOMPLIANCE explicitly takes auditability into account, with features like timestamping, cryptographic signatures and evaluation logs.

ASPECT 6, STATIONARY VS. MOVING: On one end of this spectrum we find approaches that explicitly target moving data. For example, the Data Capsule paradigm by Wang et al. [46] targets makes strong assumptions, but clearly targets moving data by accompanying the data with the policies that belong to it. LEGALEASE/GROK [38] also targets moving data by tracking data flows, and the work by Cory [12] is targeted towards data location management. On the other end we find systems that are simply aimed at access control at the database level such as Deep Enforcement by Khaitzin et al. [24] or general policy languages like OPA/Rego and XACML [32]. Our approach has a clear focus on moving data with its geolocation features and SQL-based data movements. However, the fundamental approach for QOMPLIANCE can technically also serve for stationary data by routing all queries through it before database access. By ignoring its geolocation features, QOMPLIANCE can function as a compliance system closer to how Deep Enforcement [24] works.

As we can see when re-evaluating our work compared to the related work over these axis, most of these systems target a particular niche. This is often defined by a large set of assumptions about the system's application. Although our comparison framework allows for comparing these systems on some reoccurring aspects, it is difficult to make direct comparisons between many different approaches. We have drawn parallels to programming languages, in particular in Aspect 4. The spectrum seems to mainly consist of either low-level general purpose Assembly-like languages (e.g., OPA/Rego and XACML), or very high-level but domain specific languages (e.g., LEGALEASE/GROK). There is a gap in the middle here that can be compared to higher-level but general purpose programming languages like Java. We have shown that QOMPLIANCE lives in the middle of this spectrum using this comparison framework according to Contribution 2.

7.3 PERFORMANCE

Although high performance is not a main objective of this work, it is still an important aspect to consider. Because our system can provide direct feedback to the user, a response should be generated in a reasonable amount of time, as we set out in Objective 7. In this case, a maximum reasonable response time would be the length of an HTTP request (i.e. a few seconds).

7.3.1 *Variables and Bottlenecks*

The policy matching and especially the conflict resolution is likely to be a bottleneck, along with the database querying for policies. In the absolute worst-case scenario, the conflict resolution algorithm has to compare all policies that have been retrieved from the database and their context attributes. A few optimizations can be made here to cache some intermediate results, but ultimately this comparison has to be done because of how our proposed conflict resolution works. Still, this is only becoming a problem with a very large number of applicable policies. Note that in our implementation, we already only retrieve policies by tags that apply on the query. Furthermore, we only have to execute the conflict resolution algorithm for the policies that actually apply based on all context attributes.

The compliance checking has the following variables that can influence the processing time:

1. *The number of policies.* As already briefly discussed above, this is likely to be the most influential factor. Note however that the number of policies can be interpreted in two ways: the number of policies managed by the system or the number of policies applicable on the input. Even though most of the policy matching occurs at the application side, the system uses the tags applicable on the input to only get policies from the database that apply on these tags. This greatly reduces the amount of policies that need to be considered by the matching algorithm. Furthermore, for algorithms further down the line (e.g., conflict resolution and requirements evaluation), even less policies need to be considered because likely only a subset of the set of retrieved policies will apply. Still, for conflict resolution the algorithm has exponential runtime for the number of policies.

2. *The number of attribute values per policy.* If more attribute values apply, more checks have to be done to see if a policy matches and if that policy conflicts with other policies. Furthermore, if we add more alternative values (remember that values for a single attribute have OR-semantics), more policies will apply on the same input. Adding context attributes is more likely to significantly influence the processing time compared to requirement attributes because their evaluation is generally more involved. Furthermore, context attributes are involved in conflict resolution whereas requirement attributes are not.

3. *The implementation (and size) of the attribute-level algorithms* (i.e. matching, conflict resolution, etc.). If some algorithms with high complexity are required for evaluating an attribute value, this will need to be done for all policies and possibly even for all combinations of applicable policies. We do not test their individual impact since it is trivial to derive this from the algorithms, but we show how a combination of different attribute types interact with randomly generated policies.
4. *The number of data stores, datasets and columns* that are referenced by the set of tags that apply on the query. If this number is large, more schema information needs to be retrieved and ingested by Calcite. However, Calcite's processing time is out of our hands and we are not going to evaluate its performance. The only optimization made here is that we only retrieve the schemas that are applicable on the query by performing some early parsing, but the actual validation is left to Calcite.
5. *The size of the input SQL* can also influence the processing time, but similar to the previous point this is left to Calcite and thus we do not explicitly evaluate its performance for this research.

7.3.2 *Method*

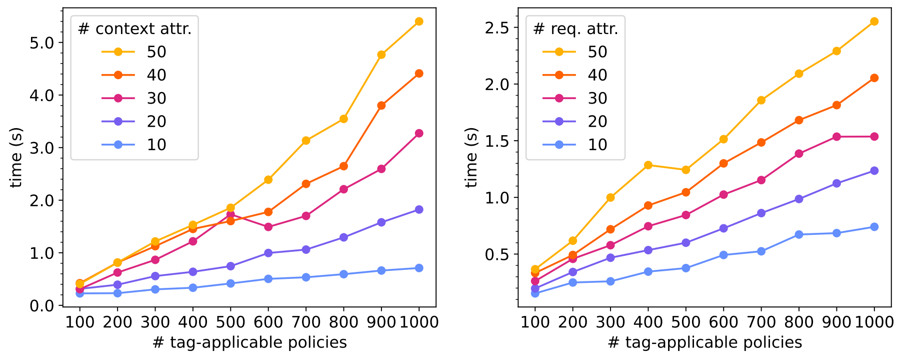
For testing the performance of our system we use the data generator that is part of our implementation, along with the schema from the TPC-W database benchmark. TPC-W is a well-known benchmark in the database space modeling an e-commerce environment [29]. We use this schema because it is a common application that many are familiar with. Its schema consists of 8 tables with 4 to 22 columns per table. We add the same schema to 5 different data stores. This allows for creating entirely contained sets of policies on the same schemas, which in turn allows us to set the exact number of tag-applicable policies. Using 5 contained sets of policies enables us to run 5 experiments with the same parameters but with a different set of policies. Because policies and their attributes are randomly generated, we run 5 experiments on different sets of policies with the same parameters to reduce major deviations introduced by the randomness by averaging the processing times over these 5 runs.

We will try various different values for the most influential parameters in order to evaluate how they affect response time. We measure the processing time as the moment right before sending the validation request to the moment that we receive a HTTP 200 response (that the system will only send if everything went right). The experiments in this chapter were executed locally on a MacBook Pro (i7-7820HQ, 16GB) with a PostgreSQL 13 instance running in a Docker container. Furthermore, we use Java/JDK 17. The jars for the two system components are started and managed by our Python evaluation script, which is also responsible for managing the experiment execution flow and processing the results.

The script initializes the data manager with the corresponding parameters for a particular experiment, which in turn will generate the data. After the initialization is done and the data is generated, the script fires a first warmup request. Then, we measure the mean processing time over 5 requests that each address a different data store for the same set of parameters. The generated policies contain randomly chosen context attributes and requirement attributes from a predefined set of values or trees depending on the attribute type.

7.3.3 *Results*

To understand the influence of the number of attribute values per policy, we plot the processing times for various numbers of applicable policies and various numbers of attribute values against each other. We run two experiments: one where we keep the number of requirement attribute values consistent (at 10) and vary the number of context attributes, and the other way around. This is done for various amounts of applicable policies, while keeping all other parameters at their default values. Figure 7.2 shows the results of these experiments. As expected, adding requirements has less of a performance impact compared to adding context attributes.



(a) Varying the number of context attribute values per policy.

(b) Varying the number of requirement attribute values per policy.

Figure 7.2: Total processing time for various numbers of tag-applicable policies against various numbers of attribute values while keeping the other parameters consistent.

Interestingly, one can also see in these plots the processing time seems to scale approximately linearly with the number of context attributes and requirement attributes (for the same number of applicable policies). This is somewhat surprising because evaluating hierarchical attributes involves comparing subtrees from two policies with the input. However, if the input and the attribute value trees do not grow too big, it seems that this influence is negligible. We expect that with the proposed higher-level attributes for this system, the performance will stay within reason. Furthermore, note that enum and tag reference attributes can be evaluated in constant time.

To analyze the differences in processing time between numbers of context attributes in a bit more detail, Figure 7.3 splits the processing time into three different components. The two major contributors to the processing time once we get into larger sets of policies and attributes are the policy retrieval from the database and the policy matching (which includes conflict resolution). This shows that the database querying is also a major factor in the processing time, and both contributors seem to grow fairly linearly with the number of context attributes.

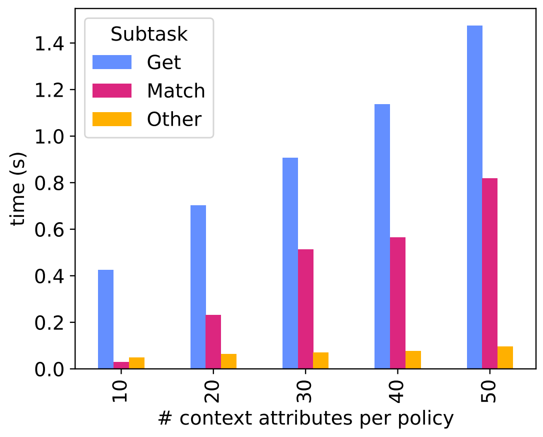
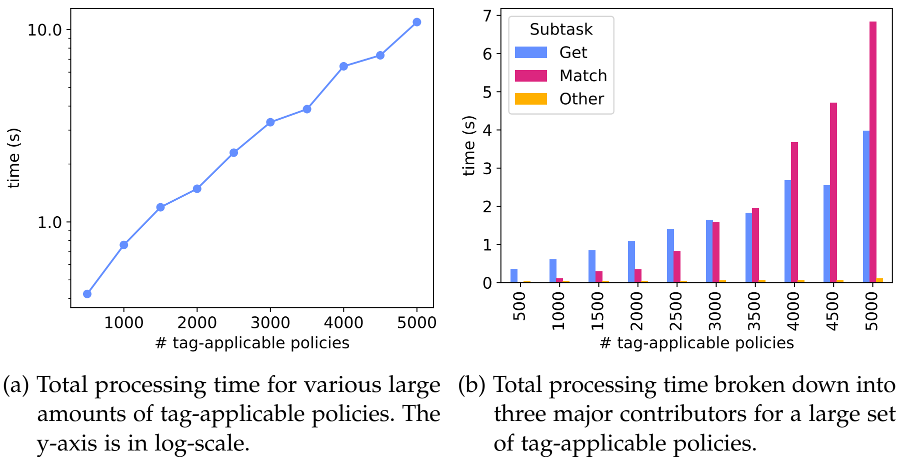


Figure 7.3: Processing times for the three major contributors to the processing time for a set of 600 tag-applicable policies with various amounts of context attributes. The 'get' subtask is the time it takes to retrieve the policies from the database based on their tags, 'match' is the policy matching and conflict resolution and 'other' includes all other time, which together makes up the total processing time.

However, the figures we have seen up until now do not clearly show an exponential growth like we would expect. This becomes more clear from Figure 7.4a, which shows the processing time for a higher number of tag-applicable policies while keeping all other parameters constant. For these results, policies were generated with 10 context attributes and 10 requirement attributes. Note that the y-axis is in log-scale, thus the fairly straight line indicates that indeed the number of tag-applicable policies have an exponential relationship with the total processing time. Important to remember here is that we again consider the tag-applicable policies here. There can be many more unrelated policies in the database which likely do not have a major influence on the total processing time.



(a) Total processing time for various large amounts of tag-applicable policies. The y-axis is in log-scale.

(b) Total processing time broken down into three major contributors for a large set of tag-applicable policies.

Figure 7.4: Total processing time for large amounts of tag-applicable policies with other parameters consistent (10 context and 10 requirement attributes per policy).

On this particular configuration and input, on average about 10-15% of the tag-applicable policies were considered a full match on the input and were thus passed on to the conflict resolution algorithm. In the case of 5000 tag-applicable policies, an average of around 800 policies were evaluated to match the input. This is an unrealistically high number of policies for the intended policy model and use case. We expect the amount of applicable policies on a SQL query not to be exorbitant, because in that case the high-level approach of our system will be hard to maintain and may not be suitable for an implementation that requires policies at that scale. Therefore, we would argue that even though this implementation has an exponential relationship with the number of applicable policies, the point at which this becomes a problem is beyond what would be a reasonable set of policies. Thus, we do not expect the response time to become unreasonable for an HTTP request, which satisfies Objective 7.

8 DISCUSSION

In this discussion chapter, we discuss go into more detail about some notable results from Chapter 7. We also discuss other notable considerations and limitations, and we give recommendations for future research.

8.1 SYSTEM SCOPE & ASSUMPTIONS

The design decision to cover the middle ground between high-level and low-level policies and policy languages (as discussed in Section 3.1) has resulted in a number of assumptions and choices. As it turns out, this positioning in the design spectrum was not an easy space to cover and one size does not seem to fit all. Like other systems that we have evaluated from the related work, our system and policy model also has its limitations because of design decisions. However, in Chapter 7 we have shown that our system has a use case in which it can operate well. The assumptions and choices made for this work have been described as much as possible throughout this work, but in this section we will discuss some especially important examples that are worth considering when implementing or extending QOMPLIANCE.

8.1.1 *Combining Attribute Values*

We realize that the design decisions made in this work may not work in every environment. A notable example is the semantics of attributes in policies that list multiple values. Whereas other systems like XACML [32] allow for complex functions within the policies to compare attributes, we opted to move some of this logic to the application level to simplify the policies without compromising too much on policy expressiveness. However, this moves the decision making about the policy structure and evaluation from the policies themselves to the application design, meaning that instead of the policy author, the system's designer is now responsible for these decisions. In QOMPLIANCE, there is an OR-relationship between values for the same context attribute (e.g. a policy with tags *PII* and *Health Data* will apply if either tag is applicable, they do not have to be both applicable). This is different for requirements, which have AND-semantics (as explained in Section 5.5.1) because all requirements have to be considered.

The decision to give context attributes OR-semantics has been made because it was the more suitable approach for our purposes because it enables one policy to govern a broader range of contexts. However, for other applications of our system, one could imagine how policy authors might want to use AND-relationships between values for the same context attribute instead, to write more specific policies. Although AND-semantics can technically be simulated by writing two policies (e.g. one for *PII* and one for *Health Data*), this may be undesirable if this is a recurring pattern in the policies that an organization wants to implement. Therefore, QOMPLIANCE has been designed with extensibility at the application level in mind. In the case of this example, changing the behavior to AND-semantics is as easy as replacing line 13 of Algorithm 2 with: '$allowableValues_{atId}$ contains all $atVals$'. This can even be done at the level of an individual attribute.

8.1.2 *Conflict Resolution & Indeterminate Decisions*

Our proposed conflict resolution algorithm, that is based on how specific policies are, is easy to understand but it cannot resolve all conflicts. If policies are considered equally specific by our algorithm, there is currently no way to tell the system what to do. This is opposed to languages like XACML which support different rule combining algorithms within the policies themselves [32]). This is a consequence of the design decisions made to do all of the policy matching and conflict resolution at the application level. Remember that in case a conflict cannot be resolved, QOMPLIANCE prefers the policy that conforms to the (configurable) system's default decision. Again, the conflict resolution strategies can be changed at the application level though by overriding the relevant methods, if another approach is deemed more suitable when implementing the described system.

Similarly, how the system deals with INDETERMINATE policies is also worth considering when implementing this system. Although we propose INDETERMINATE policies to always take precedence, an organization may want to take a more lenient approach, for example by allowing a transformation irregardless of the INDETERMINATE policy if the outcome would have been ALLOW.

8.1.3 *Tagging*

The foundation of the proposed declarative policies relies on metadata tags on the governed data. This layer of abstraction provides many benefits such as decoupling the policies from the governed data definition and making policies more understandable. However, tags have their challenges and limitations too, some of which we have already discussed in Chapter 3. Most importantly, we assume that tags are consistently and reliably applied. Because of this layer of abstraction, any decisions and guarantees made by QOMPLIANCE regarding the data are only as strong as the tags allow for. Thus, if sensitive information is not tagged as such, policies that should apply on this data may not actually be applied. Therefore, QOMPLIANCE cannot provide strong guarantees about the data itself. For example, if an adversary incorrectly tags data, QOMPLIANCE will not be able to correctly apply the right policies.

Furthermore, when facing large amounts of data, it may be cumbersome to correctly tag all data. Similarly, another factor to consider is the level of tags that the organization wishes to apply. If the organization wishes to create highly specific policies, tagging the data will be more work. However, it is important to recognize that in systems with a low policy abstraction level, similar and even more problems will exist. In these systems, if new data gets added or removed, the policies themselves have to be updated and it might be hard to find the appropriate policies that should apply on new data if the system manages a large set of policies. QOMPLIANCE aims to make this process easier with the tag hierarchy and by allowing tags to apply to different data levels (i.e., data store, dataset, column). Furthermore, the metadata can be separately maintained from the policies as opposed to systems with a low level of abstraction. Similar assumptions of reliance on metadata and tags have been made in previous work, notably GROK by Sen et al. [38]. Their work also suggests techniques for automating the data tagging. Automatic data tagging could also serve as an interesting topic for future work.

8.1.4 *Exceptions & Negations*

The proposed policy model does not allow for explicit exceptions and negations in requirements, which are features in some related work like OASIS [32] and Sen et al. [38]. For instance, it is not possible to specify a policy that explicitly says "if the data is in NL, it cannot be moved to the US" (see also Listing 7.2). This decision was made to simplify the policy model, but it introduces some limitations to the policies that can be written. Similarly, if a policy specifies a requirement, another policy cannot provide an exception for this requirement (e.g. a 'without' requirement that another policy should override). Note that policy exceptions are possible with context attributes and thus policy applicability by adding a more specific policy, this only concerns requirements. These missing features somewhat limit the expressivity of our policy model, although in many cases they can be worked around thanks to the hierarchical attributes such as in Listing 7.2. Future work can look into making the policy model more expressive by adding these kinds of constructs and possibly also look into performing user experiments for optimizing human comprehension.

8.2 SCALABILITY & PERFORMANCE

Although the results from our experiments show that the system's performance is as expected and sufficient for the use cases that we intend the system to be used for, further research could look into optimizing the algorithm design in this work. Exponential worst-case time complexity is not ideal but due to time restrictions for this thesis we were not able to get to optimizing the reference implementation.

Furthermore, the metadata database querying has a fairly significant impact on the processing time for larger sets of policies. Because of how we mapped the universal data model to a database design, two table joins need to be performed to join the context and requirement attributes on the policies. This database design can theoretically be simplified to improve performance.

Another interesting design tradeoff to look at from a performance perspective is the attribute evaluation logic. In this work, the attribute evaluation logic (i.e., attribute matching) is implemented at the application level for flexibility and extensibility. However, one could also consider pushing down this logic to the database level so that the policy matching can be entirely done by the database. This could improve performance at the expense of the system's flexibility. One exception to this is that Qompliance already does the tag matching at the database level because it is so fundamental to the design.

Lastly, it is important to recognize that Qompliance was designed to be able to statically evaluate policies, meaning that no access to the governed data is required. This has a number of benefits, especially with regards to the scalability of the system. The system proposed in this paper can be horizontally and individually scaled or even distributed at every level, down from the database up to the compliance engine. Even the evaluation algorithms can be parallelized, which can bring down the processing times for very large numbers of applicable policies. Further research could look into scaling and distributing the policy evaluation across systems and regions.

We also note that it is difficult to compare evaluation performance with other systems, or even with a baseline. We have concluded in Section 7.2 that, although these systems have some aspects in common, related systems target very different applications and scopes in a manner somewhat comparable to programming languages. Therefore, it is difficult to compare the performance between systems or for example with a common benchmark that can serve as a baseline. Future work could elaborate upon our work towards modeling the data-centric policy systems design space and possibly even design a benchmark for this space or subsets of the space.

8.3 FUTURE WORK

Apart from the suggestions that we have already made in the previous section, we also give two notable suggestions for future work that go beyond the current scope of the system.

8.3.1 *Query Rewriting*

One of the original ideas for this research was to include query rewriting as a way to enforce requirements on data transformations through the SQL. This way, by intercepting and modifying the SQL before further processing, little to no modification to existing data processing systems is needed. Similar techniques have been applied before in previous research, see Section 2.5. Suitable requirements for this idea were attributes that mandate that certain data is not included in the SQL or that certain data needs to be aggregated or anonymized in some other way for a transformation to be allowable. It is important to recognize that these modifications generally break the semantics of a query. Therefore, before proceeding with the rewritten query, it should first be presented to the query author to get confirmation whether the newly proposed compliant query is satisfactory. Qompliance is in the ideal position to give this feedback to the query author.

Let us take the 'without' requirement as an example. For typical `SELECT`, `FROM`, `WHERE` queries, this rewriting is fairly trivial: remove all references to data that is not allowed to be in the query from all places. It will break the semantics of the query but as a result we still have a presentable and valid query that can be returned to the user. However, this quickly gets more complicated with more sophisticated queries. Take subqueries as an example: the outer query may use columns from the inner query. These references will then also need to be removed. Although this would technically still be possible, one can imagine that if a query needs to be rewritten to the point that an outer query that depends on the result of an inner query cannot use this result anymore, the meaning of the rewritten query will likely be far from what was originally intended. From an automated enforcement perspective, one could argue that this still is desirable behavior. But for the purposes of this system, we want the user to have the ultimate control over what the result looks like.

Therefore, the original idea was to have the user verify whether the rewritten query suggestion is still in line with what the goal of the query was. However, the rewritten query suggestions are likely quite difficult to understand when the rewriting is done using rudimentary approaches, especially for complex queries with many interdependencies. Alternative approaches derived from related work were also considered, such as wrapping the entire query in another projection that only includes allowable columns, and replacing cell values with `NULLs` or other values. Still, both of these approaches suffer from similar problems: the former can be exploited to reveal information about the data, while the latter is in many cases likely to also break the meaning of queries, in particular when further processing expects a certain data to be populated. Furthermore, most related work only provides semantics-preserving approaches for basic queries or have made other tradeoffs or assumptions. We are currently not aware of related work that can make (semantics-breaking) query rewrites or suggestions for the modifications that our system intends to do.

Another intended application for query rewriting is the static enforcement of selecting rows that conform to a particular condition. This would be particularly useful for conditionals checking like a customer's opt-in (without any data access), as is a common notion in many regulations (see Section 2.1).

Ultimately, query rewriting was deemed out of scope for this research given the time constraints. Our alternative approach presented in this research is based on the idea that it is more intuitive to output a list of requirements that should be implemented before the query is allowable. This list of outcomes can then be parsed by a user-facing front-end that can visually aid the user with rewriting the SQL to become compliant, for example using code highlighting. Future work could look at building an editor-like interface that can assist the user with rewriting the query to make it compliant. Still, further research could also look into more advanced query rewriting strategies that attempt to provide the user with (possibly multiple) sensible alternative query suggestions that are easy to understand.

8.3.2 *Compliance as Optimization Constraint*

An interesting path for further research could be to develop a more close integration with the data processor. The data processor can then use the compliance evaluation output as input to the query plan optimization algorithm. This can for example be used to optimize the query plan based on allowable data locations, an idea proposed by Beedkar, Quiané-Ruiz, and Markl [3].

9 CONCLUSION

Throughout this work, we have introduced QOMPLIANCE, an approach towards data-centric policy compliance on SQL-defined data movements. SQL is a common denominator between many different databases and data processing workloads such as analytical queries and batch processing. Because SQL is a suitable point for early compliance detection in a data processing pipeline, we have devised a system that can statically evaluate the compliance of a SQL query to a set of policies. This enables query authors to get early feedback about the compliance of their intended query.

To simplify policy authoring and management, we introduce a data-centric policy model that allows for defining declarative policies that regulate how data can be used without directly referencing the data schemas. Based on related work and privacy legislation like GDPR, we derive a set of attributes that can be included in these policies to define the context in which they apply and to define requirements that should be fulfilled. The policy model and evaluation are based on Attribute-Based Access Control and other data-centric policy-based systems, extended with properties like NON-DECIDING policies to allow for high-level declarative policies. We propose an extensible application-level approach for defining and evaluating these attributes. By making more assumptions about how policies are structured and processed compared to languages like XACML [32], policy complexity is reduced. Furthermore, we show basic algorithms for matching and evaluating these policies on an SQL input. We also propose a set of requirement attributes that can be used to enforce requirements, for example, on the environment during or after the data processing or on the SQL itself.

We have qualitatively evaluated the use of this approach in a use case covering the significant features of QOMPLIANCE. This use case demonstrates the intended scope and implementation of the approach presented in this work. Furthermore, we have proposed a comparison framework for the design space and have qualitatively compared our approach with other data-centric compliance systems. Lastly, we have shown through empirical experimentation that the system can give feedback within a few seconds for a manageable set of policies, even though the processing time is exponentially proportional to the number of policies that apply on an input.

QOMPLIANCE is a novel *middle-of-the-road* approach that draws inspiration from various approaches towards access control and (privacy) policy compliance. By balancing the scope of the system and its policy model, we have addressed a gap that we have identified in the state of the art. Many considerations and assumptions go into the design of access control and compliance systems, and no one size will fit all. However, with QOMPLIANCE we hope to offer an interesting and flexible alternative approach towards data-centric policy compliance that may spark further ideas.

APPENDICES

A SOURCE CODE

The source code for the reference implementation described in Chapters 6 and 7 can be found on GitHub:
`https://github.com/doudejans/qompliance`

B ADDITIONAL FIGURES

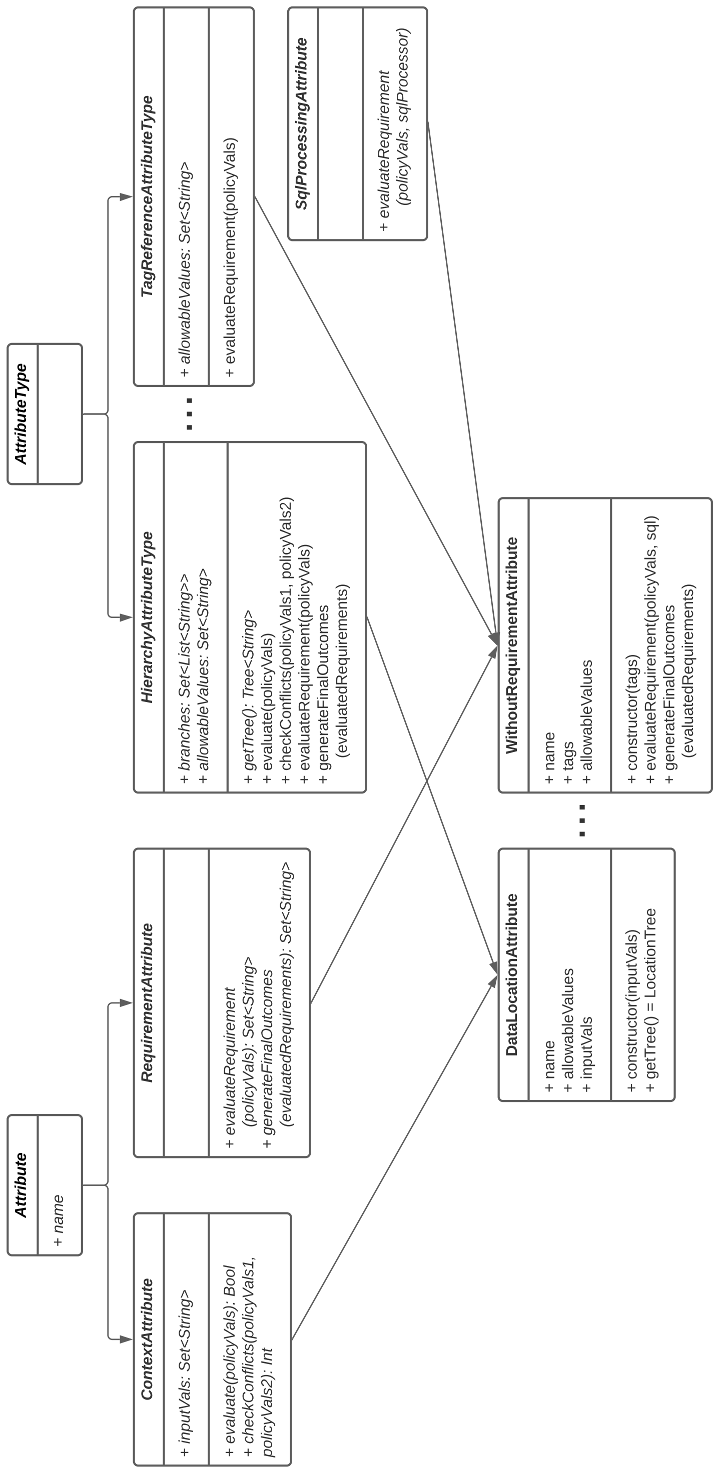

Attribute
+ name
ContextAttribute
+ inputVals: Set<String>
+ evaluate(policyVals): Bool
+ checkConflicts(policyVals1, policyVals2): Int
RequirementAttribute
+ evaluateRequirement (policyVals): Set<String>
+ generateFinalOutcomes (evaluatedRequirements): Set<String>
AttributeType
HierarchyAttributeType
+ branches: Set<List<String>>
+ allowableValues: Set<String>
+ getTree(): Tree<String>
+ evaluate(policyVals)
+ checkConflicts(policyVals1, policyVals2)
+ evaluateRequirement(policyVals)
+ generateFinalOutcomes (evaluatedRequirements)
TagReferenceAttributeType
+ allowableValues: Set<String>
+ evaluateRequirement(policyVals)
SqlProcessingAttribute
+ evaluateRequirement (policyVals, sqlProcessor)
DataLocationAttribute
+ name
+ allowableValues
+ inputVals
+ constructor(inputVals)
+ getTree() = LocationTree
WithoutRequirementAttribute
+ name
+ tags
+ allowableValues
+ constructor(tags)
+ evaluateRequirement(policyVals, sql)
+ generateFinalOutcomes (evaluatedRequirements)

Figure B.1: Full version of attributes class diagram from Section 6.3.

C YAML POLICY LANGUAGE SPECIFICATION

This document describes the YAML implementation of the policy model. The YAML implementation serves as a policy language and is used as the format for submitting policies to the system. This specification uses the example attributes used in the thesis, but can be trivially extended to support other attributes.

C.1 SPECIFICATION

A policy consists of the following basic elements:

- A unique name
- Context attributes that define the applicability of a policy
- A decision about whether the transformation/movement on which the policy applies is allowed or not
- Requirement attributes that dictate what should happen if a policy applies on the request (only allowed for `allow` and `nondeciding` policies)

Attributes can have different types which in turn have their own semantics. In the following sections, we will discuss these basic elements, and their supported attributes.

C.1.1 *Name (required)*

Every policy should have a name that can be used as a natural identifier, for referring to the policy in other places or for example in (audit) logs. This name should be unique.

Syntax:

```
name: <name>
```

C.1.2 *Context (required)*

The 'context' section of a policy lists all context conditions that should hold for a policy to be applicable. It can contain the following attributes: tag, role, purpose, data location and storage classification. A policy does not have to specify all attributes. Between the attributes in the context themselves there is a logical conjunction (AND): they all have to apply in some way for a policy to apply. Between attribute values for all attributes there is a logical disjunction (OR): any value has to apply on the input for the attribute to apply. For example, if a policy specifies tags and purposes, some tags and some purposes of the policy should be applicable on the input for the policy to be applicable on the input.

Note that exact semantics of what constitutes an attribute 'match' can differ between attribute types. Especially if a policy lists multiple attribute values per attribute, this introduces the challenge of how to consider these multiple values (e.g., all values should apply or at least one). Theoretically, the system supports defining attribute types which use different semantics when determining a match. However, the context attribute types in this proposed model only really consider the attribute values using OR-semantics. In short, this means that *for all attributes in the policy, at least one value should apply on the input*.

Syntax:

```
context:
  tag:
  role:
  purpose:
  data-location:
  storage-classification:
```

C.1.2.1 *Tag*

The most powerful contextual attribute in the system are tags. They play a central role in the data-centric approach of this system. Because of this, they have a few special traits that other attributes do not have (and they are treated a bit differently internally), more on this later.

Tags are used for restricting what data a policy should apply to. Tags serve as a layer of abstraction between the 'low-level' data model of the data that the system is governing, and the 'high-level' policies. This decouples the data definition from policies, and together with the implied meaning of the tags it allows the policy author to write declarative policies. For this to work well, all input data for the transformation has to be tagged with predefined tags that can apply to datastore(s), table(s) and column(s).

The decision to use tags comes with a number of benefits and drawbacks. We will list some here which should be taken into account when implementing the system.

Benefits include:

- *Flexible:* Tags can apply at any level of detail, both the meaning of a tag and the level at which data is tagged (datastore, table, column).
- *Decoupling policy from data definition:* Policies do not directly reference the schema of the governed data (like in many other systems). This means that new schemas can easily be added to the system without rewriting policies, and the other way around.
- *Easier to understand policies:* Because of this decoupling, policy authors can write more declarative policies by using the meaning of the tags, all without knowledge of the data in the system. Policies will also be easier to understand for people unfamiliar with the

data at all: a policy that simply references a 'PII' tag will be easier to understand than a policy referencing all columns that are considered PII.

- *Tags/metadata are common:* The use of tags and metadata in general is well-established in the industry. Many systems use metadata and tags for managing data, and a lot of research has been done towards managing and proactively generating metadata (which can be useful for this system as well). The tags in this system can also be used by other systems or the other way around.

Drawbacks include:

- *Picking the right tags:* Assigning the right tags can be difficult, especially if there are many tags, and there is more potential for conflict.
- *Adds a layer of abstraction:* Using tags creates an implicit layer between the policies and the data which could make it less obvious to see what policies apply on what data. This might influence the 'effectiveness' of the compliance system (i.e., whether all data that should be governed by a policy is in fact linked to this policy so that it can be checked). Moreover, these tags have to actually be assigned, either automatically or by someone with knowledge about the data. However, a clear user interface for managing this layer could help with managing this problem.
- *No support for direct references in policies:* In some cases, it could be powerful to write very specific policies with highly specific conditions on the data. By not supporting direct references to the data, the policy model loses some power and expressiveness. However, with specific enough tags these references can technically be emulated if desired.

Tags have to be predefined in the system, and data has to be tagged before the policies can be enforced. Tags are stored in a hierarchy, meaning that tags can have parents and children. This hierarchy can be used to organize the tags, and is also used for conflict resolution. An important difference to note is that this hierarchy of tags does not imply that if a certain tag is used in a policy, its children are also used for matching policies. This is an exception because for all other hierarchical attributes, this is in fact the case. This decision was made because it makes policies more understandable. With attributes like a geolocation, it is very clear that if a country is included in a policy, all cities in this country are implicitly also included. However, with tags this behavior can quickly get confusing, thus we opted to require exact tag matches between policies and the tagged data.

Remember that, as with all context attribute values, tags in policies always have an OR relationship meaning that it is considered to be a match for the tag attribute if any of the tags apply.

Syntax:

```
tag:
  - <tag>
```

C.1.2.2 *Role*

The role of a user is used for simple role-based access control, which is a typical feature in many access control systems. This is a hierarchical attribute, meaning that the finite set of role names should be predefined in a tree, where roles can also inherit from other roles. These role names can for example be mapped from LDAP groups.

Syntax:

```
role:
  - <role>
```

This attribute is optional. If roles are not included in a policy, the policy applies to all roles.

C.1.2.3 *Purpose*

The submitter of the job should specify the purpose for the transformation (from a fixed hierarchy of options). This is a hierarchical attribute, meaning that the finite set of purposes should be predefined in a tree, where purposes can also inherit from other purposes.

Syntax:

```
purpose:
  - <purpose>
```

This attribute is optional. If purposes are not included in a policy, the policy applies to all purposes.

C.1.2.4 *Data location*

The data location is another important attribute in the system, which enables the geolocation features of this system. It allows policy authors to write policies that govern how data should be managed across different borders and in different jurisdictions. Together with the data location requirement attribute, it can also be used to ’steer’ data to the right location.

This is again a hierarchical attribute, meaning that the locations have to be predefined in a tree. Implicitly, these trees can be based on the hierarchy of locations, e.g., Amsterdam is in the Netherlands, which is in Europe.

Syntax:

```
data-location:
  - <data-location>
```

This attribute is optional. If data locations are not included in a policy, the policy applies to all locations.

C.1.2.5 *Storage classification*

The storage classification context attribute can be added to a policy to restrict on what types of data stores a policy applies on. The classifications are labels that are attached to data stores in the metadata. Classifications can for example be useful to select data stores that have a particular compliance certification, are disk encrypted or are permitted to store sensitive data. This attribute's requirement counterpart can be used to 'steer' data to a data store with a certain classification.

Storage classifications have an enum data type, meaning that all possible values are a simple predefined list of values.

Syntax:

```
storage-classification:
  - <storage-classification>
```

This attribute is optional. If storage classifications are not included in a policy, the policy applies to all storage classifications.

C.1.3 *Decision (required)*

A decision determines whether the policy allows the input to be processed or not, given that the policy context applies on the input. This is comparable to and derived from traditional access control where this would control whether someone has access to the data or not. However, because these decisions are not enforced at data access but rather at the point of data transformation or movement, this decision should be interpreted a bit differently. This can still enable 'traditional' access control if all data access is required to go through SQL queries via our system.

A decision can be one of: `allow`, `deny`, `nondeciding`.

A `nondeciding` policy is a policy which does not make an actual decision about whether a policy allows a certain input. The main benefit of a `nondeciding` policy is that it still allows the policy to specify requirements. A `deny` policy cannot specify requirements because since the transformation is not allowed, requirements cannot be enforced anyways. One could for example use a `nondeciding` policy to put requirements on the data location, without attaching this to decisions about specific data. A `nondeciding` policy does not influence the final decision and lets other policies determine the decision outcome. If all policies are `nondeciding`, the system's (configurable) default decision is used.

Syntax:

```
decision: <allow/deny/nondeciding>
```

Note that although every policy has to have one of the three decision values, a policy in the YAML implementation does not have to explicitly list a decision. In this case, the decision will be set to `nondeciding` since this naturally follows from how you would read such a policy.

C.1.4 *Require*

The 'require' section of a policy lists all requirements that are required to be enforced once a policy is determined to be applicable on the input (based on the policy's context). A policy should either have decision `allow` or `nondeciding` to be allowed to list requirements. It can contain the following requirement attributes: data locations, storage classifications, without and aggregate. Requirements are entirely optional, meaning that a policy does not have to specify all attributes or any requirements at all. Requirement attributes all have to be enforceable for a policy to be considered valid and applicable. If a policy matches with a certain input and the requirements are validated, but the system determines that they cannot be enforced in some way, the policy decision will be set to `indeterminate`.

Note that the semantics of the requirements and what constitutes a valid requirement can vary between requirement types and what operations they enforce. Conflict resolution, handling multiple attribute values and the semantical meaning of the output of a requirement can also differ per requirement.

Syntax:

```
require:
  data-location:
  storage-classification:
  without:
  aggregate:
```

C.1.4.1 *Data location*

Together with the data location context attribute, this requirement enables the geolocation features of this system. It allows policy authors to write policies that govern where data is being processed and stored.

This attribute has the same properties as its context counterpart, meaning that the values use the same predefined tree.

Syntax:

```
data-location:
  - <data-location>
```

C.1.4.2 *Storage classification*

The storage classification requirement is another powerful requirement which can be used to dictate where data can be stored. The storage classifications are labels that are attached to data stores in the metadata. These can be used to select data stores that conform to a particular classification. Classifications can for example be useful to select data stores that have a particular compliance certification, are disk encrypted or are permitted to store sensitive data.

Storage classifications have an enum data type, meaning that all possible values are a simple predefined list of values.

Syntax:

```
storage-classification:
  - <data-location>
```

C.1.4.3 *Without*

The ’without’ requirement can be used to mandate that data is not accessed in the transformation and is not included in the final result, and thus the final SQL. This is useful to write policies that restrict what data can be accessed without preventing the data access entirely. It can be used to prevent data from being queried/joined and stored together. The possible values of this attribute are the set of tags that are registered in the system. This requirement is only considered to be satisfied if all tags mentioned in the policy do not reference any data touched by the SQL.

The requirement operates in the following way: first the tag is resolved to see what data it applies on. Then, the query is being checked for any references to the data in this set. If any of this data is included, a rewritten query suggestion will be made to the user. The input will not be allowed until this requirement is satisfied, either by rewriting the query or by accepting the query suggestion.

Syntax:

```
without:
  - <tag>
```

C.1.4.4 *Aggregate*

The aggregate requirement is used similarly to the ’without’ requirement, but instead only allows a SQL query where the referenced data has been aggregated (or not present at all). This is another way to restrict what data ends up being used, but is a bit more lenient than entirely excluding the data from a column. The possible values of this attribute are the set of tags that are registered in the system. This requirement is considered to be satisfied if all tags mentioned in the policy refer to data that is either in the SQL as aggregated columns or not in the SQL at all.

The requirement operates in the following way: first the tag is resolved to see what data it applies on. The set of data references to columns that are actually in the query and need to be aggregated according to the policy is the intersection between the data references in the query and in the set of resolved tags. Then we subtract the set of data references in the query that are in an aggregation function. If there is any column references left, these have not been aggregated and thus the user needs to update the query to reflect this requirement.

Syntax:

```
aggregate:
  - <tag>
```

C.2 EXAMPLES

In this section we show various interesting examples to demonstrate the YAML policy syntax.

```yaml
name: Keep data from leaving the EU
context:
  data-location:
    - EU
require:
  data-location:
    - EU
```

```yaml
name: Marketing department can access customer data without PII
context:
  tag:
    - customer_data
  role:
    - Marketing Dept
decision: allow
require:
  without:
    - PII
```

```yaml
name: Medical information should be kept in HIPAA compliant storage
context:
  tag:
    - medical
require:
  storage-classification:
    - HIPAA
```

```yaml
name: Data scientists can access all data for specific purposes
context:
  tag:
    - sales_data
    - customer_data
    - financial_data
  purpose:
    - analytics
    - research
  role:
    - Data Science Dept
decision: allow
require:
  without:
    - PII
```

BIBLIOGRAPHY

[1] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi. "Extending relational database systems to automatically enforce privacy policies." In: *21st International Conference on Data Engineering (ICDE'05)*. Apr. 2005, pp. 1013–1022.

[2] Paul Ashley, Satoshi Hada, Günter Karjoth, Calvin Powers, and Matthias Schunter. *Enterprise privacy authorization language (EPAL)*. Tech. rep. 2003, p. 31.

[3] Kaustubh Beedkar, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. "Compliant Geo-distributed Query Processing." In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD/PODS '21. New York, NY, USA: Association for Computing Machinery, June 2021, pp. 181–193. URL: https://doi.org/10.1145/3448016.3453687.

[4] Ji-Won Byun and Ninghui Li. "Purpose based access control for privacy protection in relational database systems." In: *The VLDB Journal* 17.4 (July 2008), pp. 603–619. URL: https://doi.org/10.1007/s00778-006-0023-0.

[5] California State Legislature. *California Consumer Privacy Act*. June 2018. URL: http://leginfo.legislature.ca.gov/faces/codes_displayText.xhtml?division=3.&part=4.&lawCode=CIV&title=1.81.5.

[6] Francesca Casalini and Javier López González. *Trade and Cross-Border Data Flows*. Tech. rep. Paris: OECD, Jan. 2019. URL: https://www.oecd-ilibrary.org/trade/trade-and-cross-border-data-flows_b2023a47-en.

[7] S. Ceri and G. Gottlob. "Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries." In: *IEEE Transactions on Software Engineering* SE-11.4 (Apr. 1985), pp. 324–345.

[8] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. "HoTTSQL: proving query rewrites with univalent SQL semantics." In: *ACM SIGPLAN Notices* 52.6 (June 2017), pp. 510–524. URL: https://doi.org/10.1145/3140587.3062348.

[9] Pietro Colombo and Elena Ferrari. "Privacy Aware Access Control for Big Data: A Research Roadmap." In: *Big Data Research* 2.4 (Dec. 2015), pp. 145–154. URL: https://www.sciencedirect.com/science/article/pii/S2214579615000489.

[10] Pietro Colombo and Elena Ferrari. "Towards a Unifying Attribute Based Access Control Approach for NoSQL Datastores." In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. Apr. 2017, pp. 709–720.

[11] Pietro Colombo and Elena Ferrari. "Access control technologies for Big Data management systems: literature review and future trends." In: *Cybersecurity* 2.1 (Dec. 2019), p. 3. URL: https://cybersecurity.springeropen.com/articles/10.1186/s42400-018-0020-9.

[12] Nigel Cory. *Cross-Border Data Flows: Where Are the Barriers, and What Do They Cost?* Tech. rep. Information Technology and Innovation Foundation, May 2017. URL: https://itif.org/publications/2017/05/01/cross-border-data-flows-where-are-barriers-and-what-do-they-cost.

[13] *Data Protection Commissioner v Facebook Ireland Limited and Maximillian Schrems*. July 2020. URL: https://eur-lex.europa.eu/legal-content/en/TXT/?uri=CELEX:62018CJ0311.

[14] Henry DeYoung, Deepak Garg, Limin Jia, Dilsun Kaynar, and Anupam Datta. "Experiences in the logical specification of the HIPAA and GLBA privacy laws." In: *Proceedings of the 9th annual ACM workshop on Privacy in the electronic society - WPES '10*. Chicago, Illinois, USA: ACM Press, 2010, p. 73. URL: http://portal.acm.org/citation.cfm?doid=1866919.1866930.

[15] European Parliament and Council of the European Union. *General Data Protection Regulation (EU) 2016/679*. Apr. 2016. URL: http://data.europa.eu/eli/reg/2016/679/oj.

[16] Kaniz Fatema, David W. Chadwick, and Stijn Lievens. "A Multi-privacy Policy Enforcement System." In: *Privacy and Identity Management for Life*. Ed. by Simone Fischer-Hübner, Penny Duquenoy, Marit Hansen, Ronald Leenes, and Ge Zhang. Vol. 352. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 297–310. URL: http://link.springer.com/10.1007/978-3-642-20769-3_24.

[17] Nils Gruschka, Vasileios Mavroeidis, Kamer Vishi, and Meiko Jensen. "Privacy Issues and Data Protection in Big Data: A Case Study Analysis under GDPR." In: *2018 IEEE International Conference on Big Data (Big Data)*. Seattle, WA, USA: IEEE, Dec. 2018, pp. 5027–5033. URL: https://ieeexplore.ieee.org/document/8622621/.

[18] Paolo Guagliardo and Leonid Libkin. "A formal semantics of SQL queries, its validation, and applications." In: *Proceedings of the VLDB Endowment* 11.1 (Sept. 2017), pp. 27–39. URL: https://doi.org/10.14778/3151113.3151116.

[19] Vincent C. Hu, David Ferraiolo, Rick Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. Tech. rep. NIST SP 800-162. National Institute of Standards and Technology, Jan. 2014, NIST SP 800–162. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-162.pdf.

[20] Vincent C. Hu, D. Richard Kuhn, David F. Ferraiolo, and Jeffrey Voas. "Attribute-Based Access Control." In: *Computer* 48.2 (Feb. 2015), pp. 85–88.

[21] Joint Task Force Interagency Working Group. *Security and Privacy Controls for Information Systems and Organizations*. Tech. rep. National Institute of Standards and Technology, Sept. 2020. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf.

[22] G. Karjoth and M. Schunter. "A privacy policy model for enterprises." In: *Proceedings 15th IEEE Computer Security Foundations Workshop. CSFW-15*. Cape Breton, NS, Canada: IEEE Comput. Soc, 2002, pp. 271–281. URL: http://ieeexplore.ieee.org/document/1021821/.

[23] Günter Karjoth, Matthias Schunter, and Michael Waidner. "Platform for Enterprise Privacy Practices: Privacy-Enabled Management of Customer Data." In: *Privacy Enhancing Technologies*. Ed. by Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Roger Dingledine, and Paul Syverson. Vol. 2482. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 69–84. URL: http://link.springer.com/10.1007/3-540-36467-6_6.

[24] Ety Khaitzin, Julian James Stephen, Maya Anderson, Hani Jamjoom, Ronen I. Kat, Arjun Natarajan, Roger Raphael, Roee Shlomo, and Tomer Solomon. "Deep Enforcement: Policy-based Data Transformations for Data in the Cloud." In: *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, Renton, WA, USA, July 8, 2019*. Ed. by Christina Delimitrou and Dan R. K. Ports. USENIX Association, 2019. URL: https://www.usenix.org/conference/hotcloud19/presentation/khaitzin.

[25] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David J. DeWitt. "Limiting Disclosure in Hippocratic Databases." In: *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. Ed. by Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer. Morgan Kaufmann, 2004, pp. 108–119. URL: http://www.vldb.org/conf/2004/RS3P3.PDF.

[26] Majid Bashir Malik, M. Asger Ghazi, and Rashid Ali. "Privacy Preserving Data Mining Techniques: Current Scenario and Future Prospects." In: *2012 Third International Conference on Computer and Communication Technology*. Allahabad, Uttar Pradesh, India: IEEE, Nov. 2012, pp. 26–32. URL: http://ieeexplore.ieee.org/document/6394662/.

[27] Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. "Formalisation and Implementation of the XACML Access Control Mechanism." In: *Engineering Secure Software and Systems*. Ed. by Gilles Barthe, Benjamin Livshits, and Riccardo Scandariato. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 60–74.

[28] M. J. May, C. A. Gunter, and I. Lee. "Privacy APIs: access control techniques to analyze and verify legal privacy policies." In: *19th IEEE Computer Security Foundations Workshop (CSFW'06)*. July 2006, 13 pp.–97.

[29] D.A. Menasce. “TPC-W: a benchmark for e-commerce.” In: *IEEE Internet Computing* 6.3 (May 2002), pp. 83–87.

[30] M. Negri, G. Pelagatti, and L. Sbattella. “Formal semantics of SQL queries.” In: *ACM Transactions on Database Systems* 16.3 (Sept. 1991), pp. 513–534. URL: `https://doi.org/10.1145/111197.111212`.

[31] Qun Ni, Elisa Bertino, Jorge Lobo, Carolyn Brodie, Clare-Marie Karat, John Karat, and Alberto Trombeta. “Privacy-aware role-based access control.” In: *ACM Transactions on Information and System Security* 13.3 (July 2010), 24:1–24:31. URL: `https://doi.org/10.1145/1805974.1805980`.

[32] OASIS. *eXtensible Access Control Markup Language (XACML) Version 3.0*. 2013. URL: `https://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf`.

[33] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. “The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases.” In: *CoRR* abs/1405.3631 (2014). URL: `http://arxiv.org/abs/1405.3631`.

[34] Siani Pearson and George Yee, eds. *Privacy and Security for Cloud Computing*. Computer Communications and Networks. London: Springer London, 2013. URL: `http://link.springer.com/10.1007/978-1-4471-4189-1`.

[35] R. L. Rivest, A. Shamir, and L. Adleman. “A method for obtaining digital signatures and public-key cryptosystems.” In: *Communications of the ACM* 21.2 (Feb. 1978), pp. 120–126. URL: `https://doi.org/10.1145/359340.359342`.

[36] Anders Rundgren, Bret Jordan, and Samuel Erdtman. *JSON Canonicalization Scheme (JCS)*. June 2020. URL: `https://www.rfc-editor.org/info/rfc8785`.

[37] Matthias Schunter and Paul Ashley. “The Platform for Enterprise Privacy Practices.” In: (2002).

[38] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Tsai, and Jeannette M. Wing. “Bootstrapping Privacy Compliance in Big Data Systems.” In: *2014 IEEE Symposium on Security and Privacy*. San Jose, CA: IEEE, May 2014, pp. 327–342. URL: `http://ieeexplore.ieee.org/document/6956573/`.

[39] Raghav Sethi et al. “Presto: SQL on Everything.” In: *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 2019, pp. 1802–1813. URL: `https://doi.org/10.1109/ICDE.2019.00196`.

[40] William Stallings. “Handling of Personal Information and Deidentified, Aggregated, and Pseudonymized Information Under the California Consumer Privacy Act.” In: *IEEE Security & Privacy* 18.1 (Jan. 2020), pp. 61–64. URL: `https://ieeexplore.ieee.org/document/8965262/`.

[41] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. "Data warehousing and analytics infrastructure at facebook." In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. SIGMOD '10. New York, NY, USA: Association for Computing Machinery, June 2010, pp. 1013–1020. URL: https://doi.org/10.1145/1807167.1807278.

[42] Damiano Torre, Ghanem Soltana, Mehrdad Sabetzadeh, Lionel C. Briand, Yuri Auffinger, and Peter Goes. "Using Models to Enable Compliance Checking Against the GDPR: An Experience Report." In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Sept. 2019, pp. 1–11.

[43] Lydia de la Torre. "A Guide to the California Consumer Privacy Act of 2018." In: *SSRN Electronic Journal* (2018). URL: https://www.ssrn.com/abstract=3275571.

[44] Michael Carl Tschantz, Anupam Datta, and Jeannette M. Wing. "Purpose Restrictions on Information Use." In: *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*. Ed. by Jason Crampton, Sushil Jajodia, and Keith Mayes. Vol. 8134. Lecture Notes in Computer Science. Springer, 2013, pp. 610–627. URL: https://doi.org/10.1007/978-3-642-40203-6_34.

[45] Ashish Vulimiri, Carlo Curino, Philip Brighten Godfrey, Thomas Jungblut, Konstantinos Karanasos, Jitendra Padhye, and George Varghese. "WANalytics: Geo-Distributed Analytics for a Data Intensive World." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, May 2015, pp. 1087–1092. URL: https://doi.org/10.1145/2723372.2735365.

[46] Lun Wang, Joseph P. Near, Neel Somani, Peng Gao, Andrew Low, David Dao, and Dawn Song. "Data Capsule: A New Paradigm for Automatic Compliance with Data Privacy Regulations." In: *arXiv:1909.00077 [cs]* (Aug. 2019). URL: http://arxiv.org/abs/1909.00077.

[47] Martin A Weiss and Kristin Archick. *US-EU data privacy: from safe harbor to privacy shield*. Tech. rep. Congressional Research Service, 2016.

[48] Rigo Wenning and Sabrina Kirrane. "Compliance Using Metadata." In: *Semantic Applications: Methodology, Technology, Corporate Use*. Ed. by Thomas Hoppe, Bernhard Humm, and Anatol Reibold. Berlin, Heidelberg: Springer, 2018, pp. 31–45. URL: https://doi.org/10.1007/978-3-662-55433-3_3.