Technische Universität Berlin

Chair of Database Systems and Information Management

Master's Thesis

Anonymized Access Control for Distributed Event Stores

Henri Tyl Allgöwer
Degree Program: Computer Science
Matriculation Number: 454925

Reviewers
Prof. Dr. Volker Markl
Prof. Dr. Odej Kao

Advisor
Rudi Poepsel Lemaitre

Submission Date
13.12.2023

Hereby I declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

Berlin, December 1, 2023

. . . .
Henri Tyl Allgöwer

Zusammenfassung

Tipps zum Schreiben dieses Abschnitts finden Sie unter [49]

Abstract

The abstract should be 1-2 paragraphs. It should include:

- a statement about the problem that was addressed in the thesis,
- a specification of the solution approach taken,
- a summary of the key findings.

For additional recommendations see [49].

Acknowledgments

For recommendations on writing your Acknowledgments see [50]. Thank you to the chair at Database Systems and Information Management (DIMA)

Contents

List of Figures

List of Tables

List of Abbreviations

ACL	Access Control List
CCPA	California Consumer Privacy Act
CLI	Command Line Interface
DASH	Data Anonymization Stream Handler
DIMA	Database Systems and Information Management
DAC	Discretionary Access Control
DES	Distributed Event Store
GDPR	General Data Protection Regulation
ICD	International Statistical Classification of Diseases and Related Health Problems
IoT	Internet of Things
JAAS	Java Authentication and Authorization Service
KNN	K-Nearest Neighbors
MAC	Mandatory Access Control
PII	Personally Identifiable Information
RBAC	Role Based Access Control
SSE	Sum of Squared Errors
SASL	Simple Authentication and Security Layer

List of Algorithms

1 Introduction

Distributed event stores capture, store, and process real-time data streams in distributed environments. They have been widely adopted across various sectors, including Fortune 100 companies, governments, healthcare, and transportation industries [4] for their efficient, scalable, and fault-tolerant system design. In the modern data-driven world, the growing demand for comprehensive data privacy policies has resulted in increasingly stringent regulations from governments worldwide [10, 29]. However, the underlying infrastructure for distributed event stores to adequately support these policies is markedly lacking [9]. This disparity poses a unique challenge, particularly when considering the demands of modern database systems to maintain high performance, characterized by low latency and high throughput.

The present work aims to examine what techniques can be effectively employed to ensure privacy and security measures, such as access control and data masking, in distributed event stores. Additionally, it aims to explore what strategies can be developed and implemented to ensure that these privacy and security measures have a minimal impact on the performance of distributed event stores.

While there exists a body of work focusing on anonymization and data masking for data streaming [7, 53, 39], there is a noticeable gap in research specifically targeting distributed event stores. Furthermore, although there are enterprise technologies for managing data flowing into such systems [12], there is limited literature on techniques designed for data already within distributed event stores. Most notably, the concept of integrating Role Based Access Control (RBAC) within this framework, where the role assigned determines the level of anonymity accorded to the data, is a completely novel approach.

The introduction of access control coupled with anonymization in distributed event stores holds the potential to contribute to more advanced, efficient, and secure data handling. By making such tools accessible and cost-effective, companies might be more inclined to prioritize and invest in user data privacy.

The overarching goal of this thesis is to design, implement, and evaluate a management framework for distributed event stores. This framework aims to incorporate RBAC, granting administrators the capacity to define levels of data anonymization and specify masking functions. By associating levels of anonymization with the number of additional data streams per original data stream, the framework will facilitate nuanced and customizable data privacy measures. Moreover, this plugin will facilitate the assignment of consumers to streams based on their specific roles, further enhancing granular control over data access and privacy within distributed event stores.

The scope of this thesis will primarily encompass the exploration of privacy and security measures in the context of distributed event stores. The thesis will delve into the administrative aspects of these technologies, including the implementation and management of access control and data masking functions. Particular attention will be paid to the role of administrators and the impact of their decisions on the performance of these systems. This includes the computational implications of various anonymization techniques. It will involve an in-depth study of different strategies' performance impacts, aiming to minimize overhead while maintaining robust data privacy. At the same time, this thesis will not specifically address or align with any particular data privacy policy such as the General Data Protection Regulation (GDPR) [10]. While the thesis is designed around general principles of data privacy and security, it will not provide policy-specific solutions or address the nuances of any specific regulatory framework. Furthermore, the anonymization of data flowing into or out of distributed event stores is beyond the scope of this thesis. The focus will primarily be on the data within the systems, and not on the methods and protocols for handling data entering or leaving these systems.

This thesis introduces the Data Anonymization Stream Handler (DASH), a system architected to process data streams into multiple anonymized versions. This design aims to provide users of distributed event stores with a nuanced granularity of anonymization. A comprehensive survey of anonymization techniques is presented, categorizing these methodologies for practical application in various scenarios. DASH includes an extensive library of anonymization techniques, offering users the flexibility to tailor the anonymization process to their specific requirements. Additionally, role-based access control is made available as a separate component to assign anonymized versions to streams. This combination of access control with anonymization variety is not only theoretically robust but also practically applicable. Its synergy is highlighted in our theoretical framework chapter, where we showcase a real-world example. Furthermore, this thesis presents a theoretical model designed to simplify future implementations of anonymized access control across various distributed event stores.

In the context of this research, we have tested and evaluated DASH in an extensive data pipeline environment. We have found ...

The thesis is structured as follows: in Chapter 2 we conduct a literature review.

The theoretical framework is constructed in Chapter 3. Subsequently, Chapter 4 examines in detail the implementation. Tests and their evaluations are addressed in Chapter 5. Related Work is the subject of Chapter 6. Finally, Chapter 7 presents the conclusion and offers an outlook on future work.

2 Literature Review

This chapter presents a comprehensive review of the important literature relevant to the key topics of this thesis. Initially, it examines the evolution of access control. This is followed by an in-depth exploration of distributed event stores, elucidating their development, their strengths and weaknesses, and their prominence in modern distributed database systems. Subsequently, the following section explores data anonymization, highlighting general techniques as well as those specifically tailored for streaming data.

2.1 Access Control

Access control is a fundamental concept in information security, defining the actions that a subject, typically a user or an automated process run by a user, is authorized to perform on an object, which could be data, files, or other system resources. These actions include a variety of operations, traditionally including but not limited to, reading, writing, and executing files or data. As a critical component of system security, the methodologies and principles have been subject to extensive research and evolution over several decades. Ever adapting to the change of requirements and emerging security concerns.

Sahndu and Samarati's seminal work [42] lays the foundational framework for understanding access control in the broader context of information security. They emphasize access control as an integral component of overall security, closely linked with authentification and auditing. They go on to describe the basic principles of access control, distinguishing between policies and mechanisms, focusing on the former. First, they explain the Access Control Matrix and their derivatives Access Control Lists (ACLs) and Capabilities lists and ultimately Authorization Relations. The Access Control Matrix is a basic security model where rows represent subjects and columns represent objects. Each cell in this matrix specifies the actions a subject can perform on an object. However, this matrix often contains many empty cells as not all subjects interact with all objects. ACLs offer a streamlined approach by listing objects and specifying which users have what permissions. Conversely, Capabilities Lists provide a subject-focused perspective, listing subjects and their permissions for different objects. Furthermore, they analyze the prevailing standards of their time: Discretionary Access Control (DAC) and Mandatory Access Control (MAC). In DAC permissions are assigned by the resource owner typically in the form of ACLs. A resource owner is also at liberty to delegate the task of granting permissions to another subject. Thus, this approach does not facilitate control over the dissemination of information, however, the decentralized approach provides flexibility. MAC on the other hand relies on a central authority to decide on all matters related to permissions. With roots in the US military, it drew inspiration from the need-to-know principle. In MAC the flow of information is strictly regulated. Overall, the authors criticize both approaches to access control regarding their adaptability and scope. The mandatory approach was considered too rigid, the discretionary model was largely confined to research applications due to its cooperative yet autonomous focus. They appreciate the newcomer RBAC in the space, for its discretionary flexibility and its mandatory strictness, providing a wide range of applicability, especially in commercial enterprises. A key benefit of RBAC lies in its facilitation of smooth transitions of permissions and privileges when a user's role within an organization changes. Moreover, RBAC simplifies the implementation of separation of duties, through mutually exclusive roles. Sahndu elaborates on RBAC further together with Coyne [41] explaining that finding a consensus in the form of a RBAC standard requires a multidimensional view, stating that considerations regarding the nature of privileges and permissions, hierarchical roles, user assignments, privilege and permission assignment, role usage, role evolution as well as object attributes have to be made.

A comprehensive understanding of RBAC requires the distinction between user groups and roles. Permissions based on user groups are a long-established practice - but do not offer the same range of functionality as role-based permissions. Ferraiolo and Kuhn [17] identify two distinct differences, the first being that groups function as a discretionary mechanism, unlike roles. Access rights are assigned at the liberty of the object owner, with groups comprising users to whom the owner grants access. In contrast, roles represent a more abstract categorization of the user, allowing a single user to be associated with multiple roles. Unlike group-based access, permissions in a role-based model are assigned based on the roles themselves, not at the discretion of the resource owner. The second key difference lies in the operations associated with the permission. Groups are assigned classical file permissions as is common for an operating system e.g. read, write, execute, and own. Roles, on the other hand, refine this approach by defining 'transactions' - the authorization to execute a specific function on a set of data items. This allows for a much more nuanced and finetuned approach to access control, aligning more closely with practical requirements and daily operations.

Building on these insights, Sandhu further solidifies the concept of RBAC in his pivotal work 'Role-based access control' [40] laying the cornerstone for the formulation of a standard. In 2001, Ferraiolo et al. [18] proposed a NIST standard for role-based access control, with Sandhu notably listed as a contributing author. Their goal was to bring clarity and establish well-founded, common terminology in the field of role-based access control. Their standard introduces four levels of RBAC, each building upon the other. The foundational level, termed, 'Core RBAC', includes basic data elements, namely users, roles, objects, operations, and permissions. Users are assigned to roles. Roles in turn are assigned permissions. Permissions are permissible operations to objects. In any given session, a user operates under a specific subset of their assigned roles. This core concept is then expanded to include hierarchies of roles, static separations of duties, and, ultimately, dynamic separations of duties.

Access control methodologies continue to evolve, with fine-grained access control emerging as a particularly popular approach due to its enhanced granularity in permission settings. Wang et al. [51] examined the correctness of fine-grained access control, formulating the requirements of them being sound and secure to achieve maximum information. Other concepts include purpose-based access control for privacy protection [6].

An especially interesting inquiry relevant to this thesis is raised by Chaudhuri et al. in [8]. They ask whether there is common ground between database access control and privacy. Despite their apparent relation, these two are seldom addressed together. In their journal article, they expand on the differential privacy notion with noisy views. Differential privacy requires that computations are formally equivalent when performed with or without any single record. Noisy views refer to the technique of adding noise to aggregate data in a database, thereby enhancing privacy. This innovative combination implemented on a database server level seems promising in bridging the gap between access control and privacy. This approach of harmonizing access control and privacy aligns closely with the objectives pursued in this thesis.

2.2 Distributed Event Stores

In modern times data is being produced at an unprecedented rate and volume. A significant portion of that data is so-called 'streaming data'. Streaming data is continuously generated, often in high volumes and at high velocity, from various sources. Its distinguishing characteristic, however, lies in its continuous flow and boundless nature. It additionally requires real-time or near-real-time processing as relevancy is key. Sources of streaming data include Internet of Things (IoT) devices, log files, financial transactions, and social media platforms. One of the key challenges of streaming data is the need for systems that can process and analyze the data as soon as it arrives. This difficulty is exacerbated by the volume, velocity, and variety (the three Vs of Big Data) of the data. Moreover, streaming data often requires a different approach to data management and storage. Since the data is continually flowing, it is infeasible to store all incoming data indefinitely.

Traditionally, relational databases have been the go-to with Oracle [13] and MySQL [36] as the dominating database management systems [44]. Over time, relational databases have evolved. Initially, their primary focus was enabling high-speed transaction processing, as illustrated by Selinger et al. in 1979 [43]. The emergence of unstructured data, as is common in streaming data, for instance, has led to a shift away from relational databases to column-based or more generally NoSQL databases. Prominent examples of such databases include Apache Cassandra [19], Cloud Bigtable [33], Amazon Redshift [2], and MongoDB [27]. The immediate processing and decision-making requirements of streaming data can oftentimes not be met by these types of databases. Specifically, managing time-ordered events and executing temporal queries present significant challenges. Additionally, streaming data necessitates robust transactional guarantees in addition to complex event processing capabilities. This poses another challenge to NoSQL databases. Ultimately, the high throughput and low latency demands on top of the aforementioned requirements are enough to require a new model entirely.

The architectural pattern of event sourcing emerges. It involves storing the state changes of an application as a sequence of events. Instead of keeping the current state of data in a database, every change (or event) that affects the system's state is captured and stored. These events are immutable, meaning once they are stored they cannot be changed. A principal advantage of event sourcing thus lies in the reproducibility of system changes. Events can be replayed to reconstruct the system state at any point in time. This is particularly useful for debugging, auditing, and understanding the sequence of actions that led to a particular state.

Numerous entities have integrated event sourcing into their database designs. Notably, Greg Young, a pioneer of event sourcing, developed EventStoreDB [52]. Nowadays, there are commercial as well as open-source solutions available, offering comprehensive capabilities for event storage. To facilitate scalability and fault-tolerance most opt for a distributed approach. Prominent Distributed Event Stores (DESs) include Apache Kafka [4], Amazon Kinesis [1], RabbitMQ [34] and Apache Pulsar [20].

2.3 Anonymization

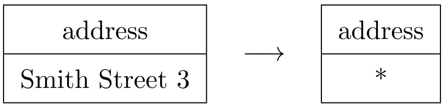
Protecting privacy is an increasing concern for companies that rely on processing data. With extensive legislation in place across the globe, data scientists and data engineers alike are tasked with securing their users' data. The key requirement for privacy is restricting the reidentification of an individual with a record. Typically, the attributes in a single record can be categorized into direct identifiers also called Personally Identifiable Information (PII), indirect identifiers called quasi-identifiers, and the remaining, sensitive data. Anonymization takes on different forms in achieving the task of preventing reidentification. One approach is to use cryptographic methods to make it unreadable for individuals without access to the key. Depending on the concrete cryptographic function, this can be an expensive yet secure way of doing it. While Cryptography is a fascinating field at the intersection of computer science and mathematics, its application for streaming data appears limited due to its costly nature, with the cost in performance making it unattractive for distributed event stores. An alternative, yet effective approach involves employing masking functions, which alter the original data values, thereby obfuscating them and making it more challenging for unauthorized entities to discern the true information. In contrast to cryptographic functions, this is generally a cheaper approach, but the resulting protection is highly dependent on the masking function and its application. In the following, we survey the most prevalent masking functions, describing their operational principles and applications. Subsequently, we address particular considerations that are to be made in the context of data streaming, underscoring how these principles adapt to the dynamic nature of such environments.

2.3.1 Masking Functions

Initially, we establish a basic understanding of key terms essential for this discussion. In this context, a *datum* refers to a single piece of information e.g. a singular entry of a database or any one event of a data stream. Synonymous with it also used the word *tuple*. A tuple is composed of one or multiple *attributes*. The names of the attributes are typically used as keys for identification of the attribute within the tuple. It is customary for individual tuples to be part of a bigger collection. In the static context, they are collected in databases and grouped in tables. Data streams work analogous to dynamic operations. All tuples of the same database table or data stream are required to follow the same pattern. This refers to the sequence of attributes of each tuple. This pattern is fixed in a data schema associated with the table or stream. Sometimes it is appended to each datum in the form of a header. As this substantially increases the size of each datum it is more common to define it once in the initialization step of the database table or data stream.

Having established the terminology let us begin by introducing prevalent masking functions:

Suppression aims to effectively delete the value of a tuple's attribute by replacing it with a meaningless character, most commonly the asterisk *. It is important to note, that actually removing the attribute from the tuple or replacing it with a null value would violate the data schema and thus negatively impact operability. The asterisk does the trick while maintaining the data schema. To work, Suppression only requires a non-empty set of keys for the attributes that are supposed to be suppressed as parameters. Naturally, it leads to total information loss of the specified fields. This can be particularly useful for fields containing PII like a person's home address as shown in Figure 1.



address
Smith Street 3

→

address
*

Figure 1: Example of suppression of an attribute.

A similar approach is applied in **Blurring**. Here, the value of an attribute is replaced with arbitrary characters, typically Xs. It is distinguishable from Suppression in that not all characters of the value have to be replaced and even the amount of characters can remain the same. Imagine a user at the checkout of an online store that they have already purchased goods at prior to this session. Here the credit card information of that user was saved as part of the agreement from the previous session. The user is then given the option to use that credit card again, with it being specified as a sequence of blurred characters with only the last three digits in plain text as shown in Figure 2. This allows the user to double-check the card information without exposing the credit card number to the network, screen captures, or bystanders. This operation also leads to high information loss but retains some usability of the original value. The parameters for Blurring include the keys to blur as well as optionally the number of characters and whether the amount of characters is to be maintained. Note, that setting the parameters to blur all characters and reduce the amount to one is equal in functionality to Suppression.

Substitution replaces the value of the specified attribute with a predefined substitute. Figure 3 shows an example where a name is switched out with an

credit card		credit card
1234 5678 9123 4567	⟶	XXXX XXXX XXXX X567

Figure 2: Example of blurring of an attribute.

arbitrary fake name from a provided substitution list. While this masking function leads to substantial information loss, it seemingly maintains the integrity of the data from an outsider's perspective. This can make the data easier to work with, while still ensuring anonymity. As parameters, the keys for the attributes that are supposed to be substituted are required in tandem with the intended substitutes.

name		name
Martin Smith	⟶	John Doe

Figure 3: Example of substitution of an attribute.

An alternative approach is **Tokenization**. Here, values are also substituted, but not with some arbitrary replacement. Instead, the substitute is a specific token. These tokens can be reversed to restore the original value as long as a key is known with which the token was created. There are different approaches to achieve this. The first one coming to mind is a database mapping token to the hash of the original value. Only access to the database as well as the hashing function will yield the correct original value from the token. Another approach is to omit the database and instead use a more sophisticated cryptographic algorithm to create the token, effectively encrypting the data. An additional, less computationally intensive method involves the use of a hashmap, where original values are associated with randomly generated character sequences. This approach, while simpler, still provides a level of security by obfuscating the original values. Each method has its trade-offs: while the database and cryptographic approaches ensure a robust security level, they necessitate significant storage and computing resources. On the other hand, the hashmap approach, though less resource-intensive, might offer a slightly reduced security level. This makes the choice of method dependent on the sensitivity of the data and the available resources. For instance, highly sensitive information, such as passwords, might necessitate more resource-intensive methods, as depicted in Figure 4.

One of the most common masking functions is **Generalization**. Here, the value of an attribute is abstracted to a more general value. This effectively reduces

password		password
MyPassword1	→	d9f4c3b72c7935e5

Figure 4: Example of tokenization of an attribute.

the information - but does not remove it entirely. For example, a datum with a residency field could generalize the exact location to a broader one. Instead of Berlin, it would read Germany as shown in Figure 5. This could then be even further generalized to Europe and so forth. Typically, entire generalization hierarchies are provided as parameters to facilitate this. These hierarchies must be exhaustive of all possible arising values if no default is provided as a generalization is ambiguous. These complete generalization hierarchies are required as parameters in addition to their respective attribute key.

residency		residency
Berlin	→	Germany

Figure 5: Example of the generalization of an attribute.

A special case of Generalization is **Bucketizing**. It functions similarly, but exclusively for numerical values. Ranges replace specific values in the tuple. Figure 6 shows an example where the age 27 is bucketized to the range [20 - 30]. Only a moderate amount of information is lost with this masking function. To deploy, it requires the bucket sizes as well as the attribute keys.

age		age
27	→	[20 - 30]

Figure 6: Example of bucketizing of an attribute.

Finally, there are the **Noise Methods**. Again, these only apply to numerical data. The idea is to modify the original values by adding noise. Typically, this noise is chosen randomly from a distribution. The standard deviation will then define how much each data point can diverge from the original value. Utilizing the normal distribution with mean 0 would ensure that the data over time would retain its mean and variance. The result will invalidate individual tuples but preserve the overall spread of the data in the long run. As an example, Figure 7 shows noise added to two attributes of a tuple. Note that with no loss to the generality, one is decreased, while the other increases.

height	weight
185	83

→

height	weight
181	84

Figure 7: Example of adding noise.

The aforementioned masking functions are applied to a single value within a record. There are more complex masking functions that go beyond this. For instance **Conditional Substitution** extends this by requiring a certain condition to be met for a masking to occur. Also, masking functions are considering multiple records at the same time. **Aggregation** is a familiar example where values of multiple records are aggregated with a specified method. One of the widely accepted models for preserving the privacy of data subjects in the dataset is **k-anonymity**. Introduced by Sweeney [46], the k-anonymity model necessitates that any record in a collection is indistinguishable from at least $k-1$ other records regarding their indirect identifiers. This model has been revised in the past, with further requirements as in the diversity of distinct sensitive values and maintaining the spread within the dataset within each subgroup of values. All these more complex masking functions rely upon and utilize the masking functions depicted in these subsections. They are the foundation of masking functions overall. The anonymization techniques we have just described provide further protection and are of great use at a cost of decreased performance. In Section 3.2 we delve deeper into these more complex anonymization techniques and provide insight into their application, especially in the context of this thesis.

2.3.2 Data Streaming

In the context of Distributed Event Stores, which operate on streaming data, additional challenges emerge, as previously established. Simple masking functions are well-suited for data stream handling, as they modify single records and integrate seamlessly into the data pipeline. Complex masking functions, such as those performing aggregations based on multiple tuples, require a distinct approach. An unbounded stream must be discretized into finite sets of records for these functions to operate effectively. Modern data stream handlers, such as DES, typically employ 'windowing' techniques. Windowing segments an unbounded stream into discrete windows of record collections. These may vary in size, overlap, and basis (periods, sessions, or states). The scope of this concept is extensive, encompassing a vast array of methodologies and applications. Researchers have developed specialized adaptations of established concepts to address the unique challenges of streaming data. For instance, targeted solutions for k-anonymity within data streams include KIDS [53] and CASTLE [7].

3 Theoretical Framework

This chapter lays the theoretical foundation for addressing the challenges and intricacies of anonymized access control in distributed event stores. It begins with an exploration of managing different anonymization granularity. In this process, it describes one specific use case in detail, underscoring the necessity and practicality of various levels of anonymization, illustrated through a real-world scenario. The following section presents a categorical survey of anonymization techniques. This survey aims to provide a comprehensive overview of existing methods, structuring these techniques in a way that showcases their complexities as well as their potential applications. The focus here is on categorizing and understanding the broad spectrum of anonymization methods, setting a theoretical basis for their practical implementation. In the final section, we discuss a concrete theoretical model capable of fulfilling the requirements for varying levels of anonymization granularity for distributed event stores. It will go in-depth on the model architecture as well as its functional and non-functional requirements.

3.1 Managing Different Anonymization Granularity

When planning to integrate anonymization techniques into existing systems, there are many things to consider. First one must understand the data flowing through the system. Does it include PII? Is there further sensitive data? What part of it is necessary for system maintenance? Maybe there is additional data collected for statistics. Bearing this in mind the next thought would be what needs to be anonymized. For this privacy agreements with the user must be taken into account. There may be additional government regulations in place like the California Consumer Privacy Act (CCPA) in the United States of America or the GDPR in the European Union. The next course of action is deciding on specific masking functions. As was discussed in 2.3 and further detailed in the subsequent section there is a plethora to choose from. It is important to note that all forms of anonymization lead to a loss of information. Choosing Blurring or Suppression, two methods that replace attributes with a placeholder, for all critical data fields derived in the previous assessment, will ensure that all privacy concerns are addressed, it will also diminish all intelligence gained from collecting this data in the first place. It is questionable if not collecting this type of data in the first place would then be the better solution as it would save storage and computing power. An alternative approach would be to invest heavily in IT security ensuring that no intruder with malicious intent can gain access to sensitive data. Keeping in mind, however, that social engineering attacks are nowadays the most common and effective strategy giving a guarantee of safety can be impossible. It also stands to reason that the more employees a company has the risk of social engineering attacks increases. Both of these radical approaches do not seem to adequately solve the problem. Fortunately, there is a way to navigate between these two extremes. An attentive observer of the company's data operations will likely notice that data can and should be restricted similarly to permissions: Only the data needed to fulfill the user's duty should be accessible to the user. In addition, there are anonymization techniques, which do not lead to total information loss like the two mentioned before. Generalization for example can be employed to significantly reduce the re-identification of individuals, while simultaneously retaining some information. With data restriction and different anonymization techniques in mind, there is a middle ground to be found that maximizes security and minimizes information loss. Consider the following example:

3.1.1 Use Case Example

Hospitals depend on the collection, management, and analysis of data to administer the best and most accurate care to their patients. In a modern hospital, all data would be stored in a centralized hospital database. Here all data for individual patients are brought together. Table 1 shows an exemplary table of patients in the endocrinology ward of a German hospital. Note that the tuple has been shortened to enhance its readability as well as prepended with a header of the corresponding database table. A more detailed version is shown in Table 11 in Appendix A.

The header of table 1 shows eleven attributes. First the person id (*pid*), which is the primary key for each patient as it uniquely identifies the patient in the database. Then *name*, zip code (*zip*), *sex* and *age* are included as additional personal information. This would typically also include the full address not just the zip code, contact information, height as well as weight to adapt the dosage of the medication. The subsequent two attributes include information about the patient's insurance information. Insurance companies in Germany are uniquely identified with a nine-digit institutional identifier. In the case of the first entry the insurance company (*ins. co.*) has the value 101575519, which matches the identifier of the Techniker Krankenkasse (TK). Each client is then assigned a number unique to

pid	name	zip	sex	age	ins. co.	ins. no.	diag.	gluc.	hba1c	med.
1	F. Ott	10969	M	28	TK	K15489	E10	22.1	8.74	Insulin
2	L. Lieb	34127	F	59	AOK	Y41271	E11	16.3	7.61	Metformin
3	T. Zeit	70192	M	15	TK	Z17291	E10	23.8	8.13	Insulin
4	H. Lang	80923	F	21	TK	I79435	E10	18.9	7.99	Insulin
5	J. Putz	91757	D	24	IKK	Q29751	E10	21.2	6.04	Insulin
6	I. Spies	60819	M	68	TK	J33921	E11	19.1	5.07	Metformin

Table 1: Example table of diabetes patients

that insurance company called the insurance number (*ins. no.*). It always starts with a letter followed by digits. Finally, the datum references the medical information. It starts with the diagnosis (*diag.*) classified according to the International Statistical Classification of Diseases and Related Health Problems (ICD). E10 is the label for Type 1 Diabetes Mellitus; E11 is the label for Type 2 Diabetes Mellitus. The most important medical measurement for the treatment of this disease is the current amount of glucose (*gluc.*) in the blood. This determines the quantity of medication (*med.*) to be administered to the patient. For Type 1 Diabetes this is Insulin, for Type 2 it is Metformin. Lastly, the table includes an attribute called *hba1c*. This is the body's three-month average of blood glucose. Through which diabetes is diagnosed. In this case, it is also symbolic for all additional diagnostic findings. Glucose and HbA1c are intentionally distinguished as separate attributes in this dataset, despite both being blood-derived metrics, due to their distinct measurement methodologies and relevance in immediate treatment contexts. Glucose can be ascertained with a single drop of blood, providing critical information for immediate treatment. Conversely, HbA1c is derived from a complete blood count and does not require instant action.

In a hospital setting, numerous actors engage with the aforementioned dataset. The most straightforward and prominent is the doctor. She will need all data to fulfill her duties. The doctor's letter contains all personal information. The medical data is needed for diagnosis and treatment. She will also need to keep the insurance information in mind as the covered treatment options are oftentimes different for each company. Additionally, she will need to write the patient's insurance information on the prescriptions. Only the pid could be omitted, but is debatable if the overhead is worth it, considering the pid can be easily inferred with all the given information. Therefore, no anonymization of the doctor's data makes the most sense.

Supporting the doctor is the nurse staff. One of their main tasks is to monitor patients and administer medication. To accomplish this they require the diagnosis, medication, and in this case the glucose data. As the HbA1c value is not relevant for the immediate treatment it can be safely omitted. Again insurance information is necessary as nurses typically do have the liberty to administer medication according to their judgment. This is especially important when considering how understaffed hospitals in Germany are most of the time. On the other hand, the patient's insurance number does not play into this. As nurses also interact directly with the patients they need some basic personal information like name and sex. Pid and zip, however, are not required. Therefore, the data for the nurse staff can be anonymized as shown in Table 2 without limiting the nurses or losing valuable information.

pid	name	zip	sex	age	ins. co.	ins. no.	diag.	gluc.	hba1c	med.
*	F. Ott	*	M	28	TK	*	E10	22.1	*	Insulin
*	L. Lieb	*	F	59	AOK	*	E11	16.3	*	Metformin
*	T. Zeit	*	M	15	TK	*	E10	23.8	*	Insulin
*	H. Lang	*	F	21	TK	*	E10	18.9	*	Insulin
*	J. Putz	*	D	24	IKK	*	E10	21.2	*	Insulin
*	I. Spies	*	M	68	TK	*	E11	19.1	*	Metformin

Table 2: Data available for the nurse staff. Note that PID, zip, insurance number, and additional medical information have been suppressed as indicated by their cell's light red background.

In tandem with the stay and medical treatment of the patient, the administration of the hospital will want to collect the money from the patient's insurance. The insurance company together with the patient's personal insurance number will suffice as identification. Administered medication will be imperative as this dictates the amount of money the hospital will get in addition to the fees for the stay. For this, the diagnosis will typically have to be added as a suitable reason. No further information is required. Limiting the amount of data here is crucial as the data is exported to a third party. Which means that additional regulations will take effect. Minimizing the data leaving the hospital minimizes security risks. With these strict rules in place, the data can be adjusted as seen in Table 3.
Note at this point that an unauthorized entity, who has gained access to both the data of the nurse staff and that of the administration, would struggle to correlate the entries. The shared available data fields insurance company, diagnosis,

and medication are likely generic enough to not point to a singular but to many patients.

pid	name	zip	sex	age	ins. co.	ins. no.	diag.	gluc.	hba1c	med.
*	*	*	*	*	TK	K15489	E10	*	*	Insulin
*	*	*	*	*	AOK	Y41271	E11	*	*	Metformin
*	*	*	*	*	TK	Z17291	E10	*	*	Insulin
*	*	*	*	*	TK	I79435	E10	*	*	Insulin
*	*	*	*	*	IKK	Q29751	E10	*	*	Insulin
*	*	*	*	*	TK	J33921	E11	*	*	Metformin

Table 3: Data available for the administration. Note that only the insurance information, medication, and diagnosis are not suppressed.

Diabetes, which afflicts over ten percent of the global population and demonstrates a rising prevalence, stands as one of the most common chronic diseases worldwide [16, 38]. Given its mostly non-lethal progression and lifetime dependency on medication, it has given rise to a substantial market. As the cause, optimal treatment, and cure remain subject to research, data from especially newer diabetes patients is in hot demand. To provide this data to research institutes following the regulations in place the hospital must ensure that no concrete patient can be reidentified. Here, advanced anonymization techniques such as K-Anonymization come into play. Each attribute of the data entry can be assigned to one of three categories: personally identifiable, quasi-identifying, and sensitive attributes. To achieve k-anonymity each entry must suppress the personally identifiable attributes - while keeping the sensitive attributes untouched. Most importantly the quasi-identifiable attributes of each data entry must be the same for at least k - 1 other entries of a data set. This is typically achieved with generalization of these attributes until k entries are found. In this use case, the personally identifiable attributes are *pid*, *name*, and *insurance number*. The quasi-indentifying attributes are *zip*, *sex*, *age*, and *insurance company*. The medical data comprises sensitive attributes. A K anonymous version of this data extry is depicted in Table 4.

While the aforementioned diabetes patient use case scenario may appear unique and specific, the aspects and nuances are applicable in numerous contexts. The distinct data requirements for doctors, nurses, administration, and research are

pid	name	zip	sex	age	ins. co.	ins. no.	diag.	gluc.	hba1c	med.
*	*	XXXXX	M	[10 - 70]	TK	*	E10	22.1	8.74	Insulin
*	*	XXXXX	M	[10 - 70]	TK	*	E10	23.8	8.13	Insulin
*	*	XXXXX	M	[10 - 70]	TK	*	E11	19.1	5.07	Metformin
*	*	XXXXX	{M, F, D}	[10 - 70]	ins. co.	*	E11	16.3	7.61	Metformin
*	*	XXXXX	{M, F, D}	[10 - 70]	ins. co.	*	E10	18.9	7.99	Insulin
*	*	XXXXX	{M, F, D}	[10 - 70]	ins. co.	*	E10	21.2	6.04	Insulin

Table 4: K-anonymized data available for external research. The sensitive medical attributes remain unchanged as indicated by the white cell background. Unlike the personally identifying attributes, which have been suppressed as denoted by the red cell background. The yellow cell background highlights the generalized quasi-identifiable attributes. Note that the entries of the first group have unchanged values for the attributes *sex* and *ins. co.*. As all three original entries shared the same value it did not need to be generalized.

anticipated to persist, albeit adapted, throughout the entire healthcare industry. It is also viable in different sectors. Imagine security levels in government matters, trade secrets, and specific customer knowledge in corporations or secrecy of correspondence for the transportation industry. Distributed event stores are utilized across all of these sectors with major players relying on distributed event stores for their everyday needs [4].

3.2 Categorical Survey of Anonymization Techniques

Data anonymization is a multifaceted process tailored to meet diverse application requirements and user needs. This section first categorizes the anonymization techniques based on their scope of operation, simultaneously setting them into context with each other through a hierarchical structure. It will then continue to go in-depth on each category, highlighting the intricacies of that category alone and providing insights, implementation considerations, and use cases to prominent examples. The aim is to provide a structured understanding of their application in various contexts.

The categorization of anonymization techniques can be delineated as follows:

- **Value-Based** Handles one tuple at a time and replaces the values of attributes independently.
- **Tuple-Based** Operates on individual attributes of a single tuple - but considers the values of the entire tuple for the change.
- **Attribute-Based** Extends the view from one tuple to a larger collection or table of data. Evaluate the values of singular attributes of the entire set and collectively make changes to that attribute accordingly.
- **Table-Based** Covers a table of data and perceives all attributes of each tuple. Adaptions to multiple attributes simultaneously are common. It can be argued that methods falling under this category are not masking functions but algorithms utilizing many masking functions to achieve anonymization on a table level.

To better illuminate the relationship and hierarchy of the aforementioned categories as well as provide some examples, refer to Figure 8. These categorizations are critical for selecting suitable anonymization strategies in various real-world applications. Having outlined the structure and dimensions of the various masking functions, it is now time to take a closer look at the functionality and use cases, beginning with the value-based masking functions.

3.2.1 Value-Based Masking Functions

Value-based masking functions form the foundation of data anonymization. These functions focus on altering single values within a tuple. They play a pivotal role in the construction of more complex anonymization methods. Although integral, their application in isolation often falls short of providing robust protection, due to their limited scope of operation, which is restricted to a single attribute. Utilizing them together in the context of more complex anonymization strategies is essential in providing effective data protection. In Section 2.3.1 we have already covered these simple masking functions and refer the reader to that for more detail. For the sake of completeness and quick reference, we provide a short overview of the most prominent value-based masking functions:

- **Suppression** Replaces a value with a generic value.
- **Blurring** Alters a value by partial suppression.
- **Substitution** Replaces one value with another valid value given a lookup table.
- **Tokenization** Replaces one value with a random value. The mapping is stored in a database.

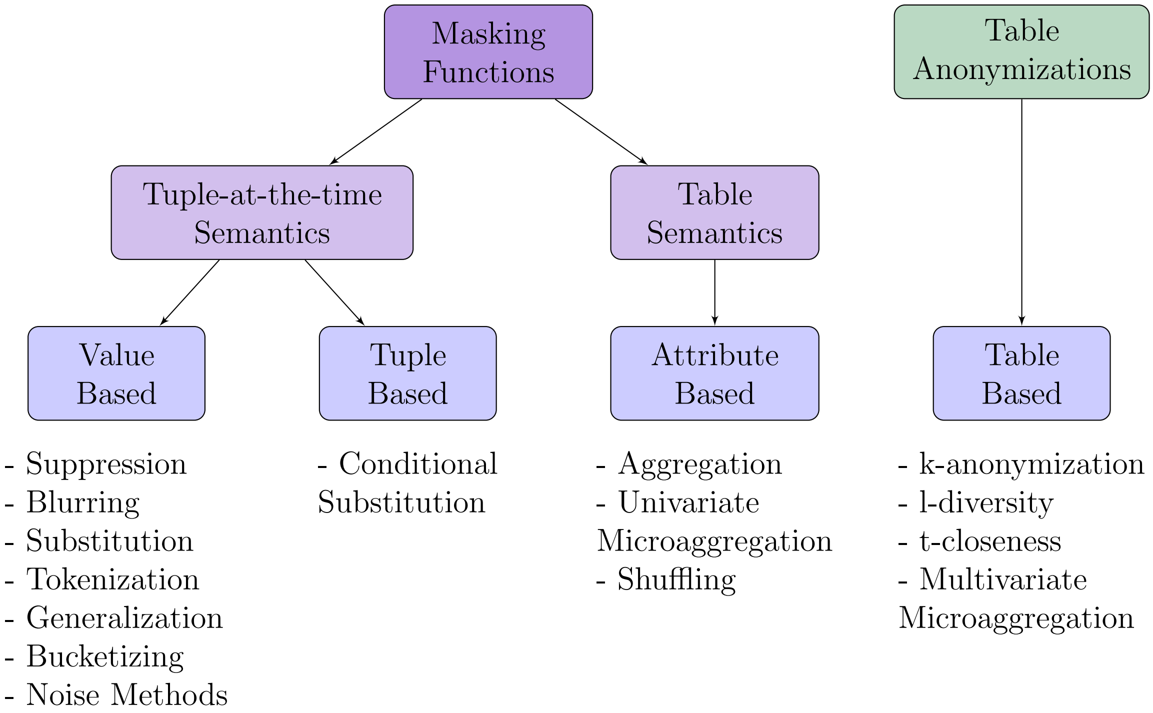


Figure 8: Hierarchy of masking functions

- **Generalization** Generalizes a value using a predefined generalization hierarchy.
- **Bucketizing** Specialized version of Generalization referring to numerical values. They are generalized by discretizing the value range and replacing each value with the corresponding range.
- **Noise Methods** Adds noise to each value, where the noise can be additive or multiplicative.

3.2.2 Tuple-Based Masking Functions

Extending the scope of operation from independent attributes to the tuple as a whole yields the tuple-based masking functions. The most notable candidate is **Conditional Substitution**. As can be expected the change to the tuple is identical to that of the value-based Substitution. The value of the attribute specified by the parameter is changed according to the given substitution dictionary. The key difference, however, is that a change only occurs if a certain condition is met. These are additionally provided as a parameter. They can be specified in a variety of different possible formats like a direct match, a numerical range, or even a regular expression. Before showcasing some examples consider the following definitions:

Let T be a tuple defined as a fixed sequence of attributes a_0 to a_n, $T = (a_0, a_1, \ldots, a_n)$. Also, let i, $j \in \{0, 1, \ldots, n\}$.

Further, a masking function for conditional substitution mf_{cs} can be defined based on a condition c evaluated on the attribute a_i, with a substitute s for the attribute a_j:

$$mf_{cs}(i, c, j, s) = \begin{cases} (a_0, \ldots, a_{j-1}, s, a_{j+1}, \ldots, a_n) & \text{if } c \text{ matches on } a_i \\ T & \text{else} \end{cases}$$

Note, that a match on c can take on different forms. An example of a conditional substitution masking function for a direct match on the value of one of the attributes is shown in Figure 9. It shows an excerpt of a database table with three entries. Remember that all tuples of a single database table share the same data scheme. For better understanding, the header with the attribute's labels is shown in the first row of each table in the figure. It shows that the data in this table has two attributes, *rank* and *salary*. The masking function $mf_{cs}(0,'Manager', 1,'*')$ is applied to every tuple in the table. It can be read as: if the attribute with index 0 matches on 'Manager', substitute the attribute with index 1 with '*'. Following this logic only one entry in the output was changed, as there was just one match.

rank	salary
Worker	62 000
Assistant	45 000
Manager	135 000

$$\xrightarrow{mf_{cs}(0,'Manager',1,'*')}$$

rank	salary
Worker	62 000
Assistant	45 000
Manager	*

Figure 9: Conditional substitution with a direct match condition taking the entire tuple into consideration.

Another example is shown in Figure 10. The condition in this case is a range. Naturally, it can only be applied to numerical attributes since there is a clear ordering of values. In the example, the database table shows two attributes: *name* and *age*. The masking function employed is $mf_{cs}(1, [0, 18], 1,'minor')$. Subsequently, the output shows two changes in the table as two entries match the condition. Note that this time the condition is evaluated on the same attribute as the substitution e.g. $i = j$.

name	age
John	45
Frederik	7
Samatha	15

$\xrightarrow{mf_{cs}(1,[0,18],1,'minor')}$

name	age
John	45
Frederik	minor
Samatha	minor

Figure 10: Substitution with a range condition on the basis of only a singular attribute in the tuple.

Finally, Figure 11 shows $mf_{cs}(0,'@example\backslash.com',1,0)$ being used as masking function. The attribute *Points* for all entries with the domain 'example.com' in the *Email* attribute, determined through the regular expression $'@example\backslash.com'$ is set to 0.

Email	Points
user1@example.com	150
service@mail.org	325
john@example.com	25

$\xrightarrow{mf_{cs}(0,'@example\backslash.com',1,0)}$

Email	Points
user1@example.com	0
service@mail.org	325
john@example.com	0

Figure 11: Regular expression as a condition for the substitution as part of a tuple-based masking function.

3.2.3 Attribute-Based Masking Functions

Following the hierarchy of masking functions depicted in Figure 8 this concludes the masking functions with Tuple-at-the-time semantics. Next, the focus shifts towards operations on tables, more specifically on collections of tuples. Before, the masking functions were applied to each tuple individually. The focus is to add more context to the anonymization and in return diminish the information loss caused by anonymizing altogether. Also, it allows the better handling of outliers. Returning to the generalization example from Figure 5, where the residency 'Berlin' was generalized to 'Germany'. If in an entire table there is only one German resident, it would be easy for an unauthorized party, who has gained access to this table, to reidentify the individual from the output. All the masking operation would have done is cost resources and lead to information loss within the system. This is of course not desirable. Fortunately, edge cases like these are unlikely for data with low spread or high density. Even in data sets with high spread and the likelihood of outliers, situations like the aforementioned can be easily avoided with knowledge of the data and the correct choice of masking function for the appropriate edge cases e.g. Suppression. An approach that more complex anonymization techniques like the ones discussed in Section 2.3.2 employ inherently. The subsequent subsection will go into more depth on this as well. For the moment, it is crucial to understand that more input can lead to less information loss and more anonymization reliability. It does, however, come at a higher computing cost as the findings in Chapter 5 show. More input in the context of attribute-based masking functions refers to more values from multiple tuples for the same attribute. Within the scope of this thesis, it is essential to highlight the particular interest in data streams being the foundation of distributed event stores as opposed to static databases or bounded datasets. Data streams inherently pose additional challenges due to their unbounded nature. In Section 2.3.2 this has been addressed already and windowing was introduced as a means of discretizing data streams. In this context 'collections of tuples', 'table', and 'window' all refer to this same discretization: a finite number of tuples as a working base obtained through windowing operations. It is worth mentioning that the parameters for windowing like window size can be included in the optimization of the anonymization itself. This particular nuance is looked into in Section 5.2

Building on this understanding, **Aggregation** as an attribute-based masking function leverages this principle. In essence, aggregation combines multiple values into one single value. This in turn will represent the attribute for all input tuples. This effectively reduces the information of an individual entry but preserves the underlying trend. By merging data it significantly reduces the reidentification probability of a single individual through this attribute. There is a variety of aggregation techniques, but they all have one thing in common. They work only on numerical values. Figure 12 shows an example. The table contains six tuples, each tuple only has one attribute, *age*. The right side of the arrow illustrates the result. Note that the amount of tuples remains unchanged, but now all tuples have the same value for the attribute. Depending on the aggregation type, which is written underneath each column, the values have been combined. The first column shows the sum, the second the median, the third the average, the fourth the maximum value found, the fifth the minimum value found, the sixth the number of input tuples, and the last the most frequent value in the input table.

A special form of aggregation is **Univariate Microaggregation**. This masking function specifically optimizes the goal of minimizing information loss. The method accomplishes this by clustering similar data. The aggregation is then only applied to these groups. This approach yields a table with greater variance in data

age
23
45
26
32
26
27

$\longrightarrow$

age						
180	32	30	45	23	6	26
180	32	30	45	23	6	26
180	32	30	45	23	6	26
180	32	30	45	23	6	26
180	32	30	45	23	6	26
180	32	30	45	23	6	26
Sum	Median	Average	Max	Min	Count	Mode

Figure 12: Aggregation operations on the "age" attribute.

than what is observed with standard aggregation. Also, outliers have less effect on the overall table. Of course, the clustering of data will require additional resources. One approach to univariate microaggregation is offered by Hansen and Mukherjee [21]. They formulate the univariate microaggregation problem as a table consisting of n entries to be divided into subgroups with at least k observations with minimized spread of the original data within the subgroups. The authors transform the univariate microaggregation problem into an equivalent graph problem. They then prove that this graph problem can be solved using a shortest-path algorithm in polynomial time. The authors commence by collecting all n values for the given attribute and sorting them in ascending order to form a vector V with $|V| = n$. Each V_i corresponds with an original value from the table, now at position $i, i \in [1, n]$. They go on to construct a directed weighted graph $G_{k,n}$. The nodes are labeled with i and added with a source node with label 0. The graph has a directed edge $e(i, j)$ from each node i to node j if $i + k \leq j < i + 2k$. The edge $e(i, j)$ is then associated with the group of values, called the 'corresponding group', in V with index h where $i < h \leq j$. The edge $e(i, j)$ is then assigned a weight equal to the Sum of Squared Errors (SSE) of the corresponding group. Finally, they prove that the shortest path in $G_{k,n}$ is the optimal solution of the univariate microaggregation problem. Algorithm 1 shows pseudocode for their proposed solution.

Let us break this down. Remember that the core idea is to minimize information loss. Less spread in a group that is subject to aggregation is key here. Therefore, ordering the elements before grouping neighbors ensures that for any two elements within a group, there does not exist another element in another group that lies between them. The question now is where to start and end each group. They then go on to construct a graph by adding nodes for each value in V. Intuitively, these are all possible starting positions of subgroups. The only constraint is that

Algorithm 1: Pseudocode for the univariate microaggregation approach by Hansen and Mukherjee [21]

```
Parameters: $T$: Table to be anonymized
            $a$: Attribute in $T$'s Schema subject to microaggregation
            $k$: Minimum observations per group
Output    : The shortest path in the form of a list of edges
$V \leftarrow$ Values of column $a$ in $T$ sorted in ascending order
$n \leftarrow |V|$
Initialize $G_{k,n}$ as an empty graph
Add a source node $s$ to $G_{k,n}$ with label 0
for $i$ from 0 to $n$ do
    for $j$ from $i + k$ to $\min(i + 2k - 1, n)$ do
        Compute the corresponding group $CG$ from $V[i : j]$
        Compute the mean $\bar{x}$ of $CG$
        Compute the SSE for $CG$
        Add a directed edge $e(i, j)$ to $G_{k,n}$ with weight SSE
    end
end
Apply Dijkstra's algorithm on $G_{k,n}$ starting from $s$
return The list of edges forming the shortest path in $G_{k,n}$
```

each group must at least contain k observations. The compound inequality $i+k \leq j < i+2k$ for the creation of edges enforces this constraint and additionally limits the number of elements within a group to $2k - 1$. This is important as larger groups lead to more information loss. However, limiting the group size to exactly k is not feasible, as n is not required to be a multiple of k. If $n \mod k \neq 0$ at least one group must contain more than k elements. Now each node i is connected via an edge $e(i, j)$ to all nodes j that are at least k and at most $2k - 1$ elements away from it. The corresponding groups to each edge $e(i, j)$ are then evaluated for their closeness to each other. This is done by calculating the within-group squared error SSE. First, the mean $\bar{x}$ of the values in the corresponding group cg is calculated. Then for each value x_i in cg the squared error sqe_i is calculated, $sqe_i = (x_i - \bar{x})^2$. The SSE for a corresponding group cg with h values is then equal to the sum of sqe_i for all x_i in cg: SSE $= \sum_{i=1}^{h} sqe_i$. With the directed weighted graph $G_{k,n}$ defined and an artificial source node labeled with 0 connected to the first k to $2k - 1$ nodes properly weighted, a shortest path algorithm is applied. In Algorithm 1 Dijkstra [15] is used. Finally, the optimal solution is the grouping according to the corresponding groups of all the edges of the shortest path.

Shuffling is another attribute-based masking function. The technique involves rearranging the values of the specified attribute. The value of the attribute for an individual tuple is exchanged with that of another tuple in the same table. Thus, from an observing standpoint, the overall data remains the same. No information is lost. In addition, reidentification based on the value of this attribute is no longer possible as the likelihood of the value belonging to the original tuple is slim. Consequently, shuffling diminishes the direct applicability or interpretability of the changed tuple. It becomes impossible to maintain valid statistical correlations among multiple attributes within a tuple. Also, users of the data can no longer rely on the veracity of individual data in the table. The negative impact on the statistical value and tuple authenticity intensifies with increasing data spread. Mixing outlying values into otherwise inlying tuples can prove to be detrimental when performing data analysis. Taking into account Shuffling's strengths and weaknesses a primary area of application is with personally identifying attributes. In this context, outliers are nonexistent, and ideally, statistics based on these attributes are rare or even absent. Consider the example shown in Figure 13. It depicts a table consisting of five tuples with three attributes *name*, *residency*, and *purchase* respectively. Shuffling is separately applied to each of the two attributes containing PII, *name* and *residency.* Note, that names do not correspond with the residencies anymore, this is due to them being shuffled independently. Also, observe that the residency of the second tuple is unchanged. When shuffling each element is switched with an element at a random index in the set. Therefore, an element may be shuffled with itself. The implementation itself can then be enhanced with a seed to effectively determine the randomness if that is to be desired. The overall result in this example shows that no person can be reidentified from a purchase, while the overall data volume and variety have been maintained. From a statistical standpoint, no harm has been done either as the primary interest does not lie in personally identifiable attributes. It is worth noting that while names are shuffled, they persist in the dataset. In situations requiring extreme confidentiality, this might not provide sufficient anonymization. Another risk emerges if another database table with shared attributes exists; such tables could be used to cross-reference tuples, potentially nullifying the effectiveness of this masking function.

3.2.4 Table-Based Anonymization Techniques

Extending the scope of operation from individual attributes within a larger collection of tuples to the entirety of the tuple with all its attributes, it is in this context the concept of table-based anonymization techniques emerges. It is essential at this point to remark that these table-based anonymization techniques rely

name	residency	purchase
Jane	Hillside	Laptop
Alice	Cityburg	Teapot
John	Lakefront	Desk
Eve	Townsville	Camera
Bob	Villagetop	Bookshelf

$\longrightarrow$

name	residency	purchase
John	Townsville	Laptop
Jane	Cityburg	Teapot
Alice	Villagetop	Desk
Bob	Hillside	Camera
Eve	Lakefront	Bookshelf

Figure 13: Arbitrary customer data where the values of the attribute "name" as well as "residency" are shuffled throughout the table.

on a fundamentally different premise than masking functions. While masking functions' main aim is to obscure or change pieces of data, table-based anonymization techniques operate on a holistic approach. The individual datum is only regarded as part of the larger set, which in turn is to be remolded to fit into predefined data privacy policies. To elaborate on this difference further think of the underlying methodology of the masking functions thus far. They all intend on masking sensitive information be it in the form of an attribute or tuple, they make specific changes to it to ensure that individual values are no longer recognizable. They suppress, obscure, and generalize the original values. The main intent here is to mask or hide the sensitive information, essentially camouflaging it to deter any reidentification. On the other hand table based anonymization techniques do not target individual data points. They use more sophisticated methods of analyzing the data set as a whole and transforming the entire table ensuring that it aligns with established privacy standards. To facilitate the comprehension of these concepts, consider Table 5 as a working example throughout this subsection. It depicts a table with five columns. In typical relational databases, attributes can be categorized into three main types: Personally Identifiable Information (PII), Quasi-identifiers, and Sensitive attributes. Personally Identifiable Information (PII) is defined as any information that can be used to explicitly identify an individual. In this example, the *name* attribute serves this purpose. Quasi-identifiers are attributes that can be used to implicitly identify an individual when correlated with other information. In the current example, the attributes *residency* and *age* function as quasi-identifiers. Sensitive attributes contain information collected about an individual but must be protected from being traced back to the individual once anonymized. In this table, *diagnosis* serves as a sensitive attribute. Finally, the example contains a nameless leading column, marked as 'Row Index'. While not part of the table's data, it serves as a reference to facilitate the reader's understanding, especially as rows will be rearranged and anonymized subsequently.

Row Index	PII	Quasi-Identifier		Sensitive
	name	residency	age	diagnosis
1	Ahmed	Berlin	27	Covid
2	John	Glasgow	45	Asthma
3	Thomas	Munich	32	Covid
4	Anna	Madrid	59	Diabetes
5	Winston	Manchester	20	Covid
6	Kim	Stuttgart	54	Covid
7	Miguel	Barcelona	34	Asthma
8	Farah	London	41	Diabetes
9	Jane	Bilbao	70	Diabetes

Table 5: Working example for table-based anonymization containing a table with four attributes marked above with their corresponding category. Additionally, there is a leading column, which is not part of the data - but serves as reference.

The most notable concept is ***k*-anonymity**. In the definition by Sweeney from 2002 [46] k-anonymity demands that any one entry in a table must be indistinguishable from $k-1$ others regarding its quasi-identifiers. Table 6 shows a 3-anonymized version of the working example. The entries were grouped into sizes of $k = 3$ according to their similarity in quasi-identifiers. These then were generalized to encompass all values in a group making them indistinguishable. The *name* was suppressed, while the *diagnosis* remained unchanged. Now no individual can be reidentified given this table. Over the years several algorithms have emerged to efficiently transform a set of data to conform to k-anonymity. Notabely, this includes Datafly by Sweeney herself [45] as well as Mondrian [31] and Incognito [30] by LeFevre et al. Each of these algorithms has its computational complexity, quality of results, and applicability to different kinds of data. Datafly for example is more suited for categorical data, while Mondrian is more optimized for numerical data whereas Incognito focuses on the optimization for computational efficiency. As this thesis’s focal point is streaming data another algorithm emerges as the most relevant - ’CASTLE: a δ-constrained scheme for k_s-anonymizing data streams’ by Cao et al. [7].

CASTLE

CASTLE stands for Continuously Anonymizing STreaming data via adaptive clustering. Streaming data differs from static databases in two key aspects: first, it is unbounded, necessitating an approach that can handle data of indeterminate size. The second distinction is that streaming data is append-only; that is, entries in a data stream are not modified or deleted once added. However, the significance of entries may diminish over time, a factor that the authors of CASTLE address by focusing on the *freshness* of the data. They do this by defining a streaming variant of k-anonymity called k_s-anonymity. Each tuple t of a stream S consists, as always, of personally identying attributes $p_1, \ldots, p_n$, quasi-identifiers $q_1, \ldots, q_n$ and sensitive attributes $s_1, \ldots, s_n$. Additionally, it is given another attribute, the position p in S. The anonymized result of the stream S is termed S_{out}. Given a position P, the authors consider S_{out} *k_s anonymized up to P* if the collection of all tuples with $t.p \leq P$ in S_{out} is k-anonymized. To further facilitate the requirement of freshness they add a δ-constraint to their scheme. It says that any such stream S satisfied the δ-constraint if all tuples with a position less than $t.p - \delta$ are already in S_{out}. Keep in mind that there is a time difference, the processing time, between S and S_{out}. The algorithm consumes the plaintext stream S one tuple at a time and produces the anonymized stream S_{out} in batches. The δ constraint limits the number of tuples that are being processed by CASTLE at any one time and subsequently ensures that fresh data is produced to S_{out}.

Minimizing information loss is important and for k-anonymity the only leeway lies within the quasi-identifiers as they are generalized. Sensitive attributes remain untouched and PII is suppressed. To be able to minimize information loss, it first has to be quantified. For a numerical attribute q_i and the generalization g to the range $[l, u]$, from the domain $[L, U]$ of q_i, the corresponding information loss is defined by the authors as follows:

$$VInfoLoss(g) = \frac{u-l}{U-L}$$

The information loss for categorical attributes is calculated according to their generalization hierarchy. Figure 14 shows the generalization hierarchy of the attribute *residency* used in the working example (Table 5)

The information loss for the generalization g of a categorical attribute to a node v in the categorical attribute's generalization hierarchy is defined as follows:

$$VInfoLoss(g) = \frac{|S_v|-1}{|S|-1}$$

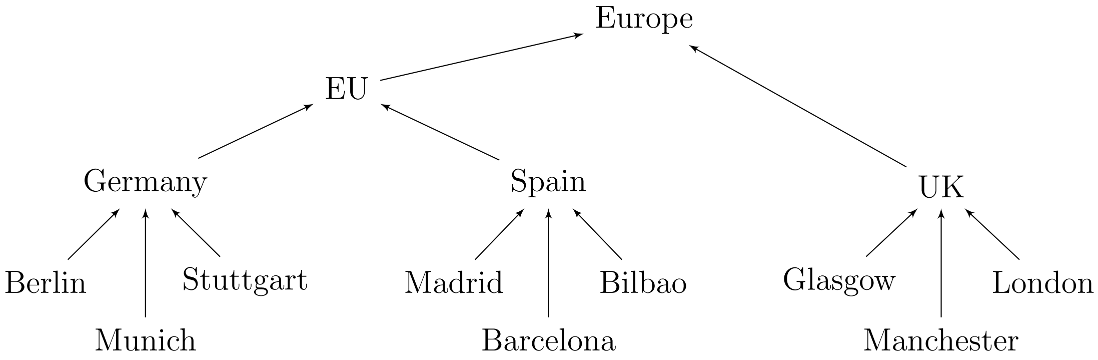


Figure 14: Generalization hierarchy for the attribute *residency*.

Here, $|S_v|$ is the number of leaf nodes of the subtree rooted at v and $|S_v|$ is the overall number of leaf nodes in the generalization hierarchy. No generalization, e.g. the attribute remaining the value of a leaf node in its generalization hierarchy, would lead to $\frac{1-1}{|S|-1} = 0$ no information loss. On the other hand, a generalization leading to complete information loss e.g. generalization to the root of the generalization tree is equal to $\frac{|S|-1}{|S|-1} = 1$. An example of the attribute *residence* and its generalization hierarchy as it is shown in Figure 14 would be the generalization to 'Germany' leading to an information loss of $\frac{3-1}{9-1} = \frac{2}{8}$.
Overall, the information loss G due to the generalization of its n quasi-identifiers is defined as follows:

$$VInfoLoss(G) = \frac{1}{n} \sum_{i=1}^{n} VInfoLoss(g_i)$$

The CASTLE scheme revolves around the clustering of data. Each cluster encompasses some tuples and is defined by the common generalization of the quasi-identifiers of its members. As mentioned before the algorithm consumes one tuple t at a time from the input stream. Information loss is key to the assignment of t to a cluster. For all clusters, the enlargement cost is determined. The enlargement cost is the additional information loss through further generalization of the cluster to include t. It is calculated for a cluster C as follows:

$$Enlargement(C, t) = \sum_{i=1}^{n} (InfoLoss(\hat{g}_i) - InfoLoss(g_i))$$

The current generalization of C for the attribute i is denoted as g_i and the generalization to include t_i as $\hat{g}_i$. Consider again the working example of Table 5. The generalization of the quasi-identifier *residency* is in accordance with Figure 14 and for *age* it is in buckets of size 5 for the domain [0,100]. Imagine the tuples are consumed in order of their row index and the current tuple t has position 6. At this point there are three Clusters, C_1, C_2 and C_3 with generalizations $G_1 = (Germany, [25, 35])$, $G_2 = (UK, [20, 45])$ and $G_3 = (Madrid, [55, 60])$ respectively. To include t the enlargement cost for G_1 is $Enlargement(G_1, t) = (2/8 - 2/8) + ((55 - 25) - (35 - 25))/(100 - 0) = 0 + 20/100 = 0.2$. Intuitively, this is to be expected as 'Stuttgart' is within the 'Germany' generalization and the age range has to be extended by 20 within a domain of 100. Similarly, the other enlargement costs are calculated: $Enlargement(G_2, t) = (1-2/8)+10/100 = 0.85$; $Enlargement(G_3, t) = (5/8 - 0) + 5/100 = 0.675$. As $Enlargement(G_1, t) < Enlargement(G_3, t) < Enlargement(G_2, t)$, t is assigned to G_1.

With every incoming tuple t the freshness of the data already within the system is evaluated. This is enforced by the δ constraint. Should the system include a tuple with a position less than $t.p-\delta$ it is marked as expired. Before any new tuple is consumed, the expired tuple is subject to be output. To adhere to k-anonymity in the output stream at least $k - 1$ other tuples must be indistinguishable from the expired tuple in the output. Therefore, not only the expired tuple itself - but its corresponding cluster is ejected with the generalization of the cluster. Here, the size of the cluster is essential to consider. It must be at least k to ensure k-anonymity. If the size of the expired cluster, however, is smaller than k it must be merged with another cluster. Again the information loss is the key metric. The merge of the expired cluster with all existing clusters is evaluated by its enlargement cost. This cost is calculated similarly as for the enlargement of a cluster to encompass a tuple. The cluster with the smallest enlargement cost is chosen and the merged cluster with their shared generalizations is subject to be output. Furthermore, a cluster should also not be too large, when it is output. The bigger the cluster the more generalization it is likely to have undertaken, and thus more information has been lost. Any cluster larger than 2k in size is therefore split. CASTLE reorganizes the cluster into subclusters with at least size k, whose generalizations lead to less information loss. It draws inspiration from K-Nearest Neighbors (KNN) algorithms to efficiently find dense subclusters. CASTLE then outputs the subclusters. Any output cluster is not deleted, it is emptied of its tuples (to save storage), but the generalizations can still be useful and are saved for reuse in a set of reused clusters KC. If a cluster is expired but is not of size k, its generalizations can be checked against the KC, should it fit any of the already output clusters it can be output with the generalizations of that cluster as it can be sure that at least k other tuples exist in the output with the same generalizations. The reuse technique mitigates the need for expensive cluster merges and splits. However, the authors of CASTLE realize that it also leaves the output open to attacks due to the overlapping of clusters. Say two clusters C_1 and C_2 were output with their only difference being the generalization of the *age* interval being [25-

35] and [20-35] respectively. If at a later stage, a single tuple is output with the gernalization of C_2 an attacker watching the output of the stream could infer, that the tuple's age is actually within the interval [20-25]. If the age was in [25-35] it would have been output with that generalization as the information loss of that cluster is lower. To avoid this, reused clusters are examined for overlap, and if it exists a random cluster's generalizations are chosen instead of the one with minimal information loss.

They optimize further by taking into consideration the unbound nature of data streams. It can be expected that the distribution of the data will not remain constant over time. Therefore, a fixed value τ for the maximum enlargement cost is not ideal. Instead, it reflects the information loss of the last μ output clusters, where μ is an input parameter. This will reflect the current expected variance of data and assist in maintaining a constant amount of clusters with ideally similar spread. They go even further with this concept by introducing another input parameter β limiting the maximum amount of clusters at any one moment in the system. This not only prevents large numbers of clusters, which need storage as well as computation power to calculate enlargement costs every time a new tuple arrives. But also reduces the amount of small-sized clusters and subsequent expensive merge operations. If an incoming tuple would require a new cluster, but β has been reached, it will instead be pushed to the existing cluster with minimal enlargement cost regardless of τ.

Another consideration they made is regarding outliers. In the context of CASTLE, an outlier can be understood as a cluster, with a small size (less than k) and a long lifetime. When a tuple in such a cluster expires, a merge of clusters would lead to potentially high information loss across multiple clusters. To avoid this the tuple is instead suppressed and output on its own. This of course means maximum information lost for that one tuple, but it is worth considering what the merge would have cost.
We refer to the original paper CASTLE by Cao et al. [7] for the pseudocode of the algorithm as it goes in-depth on its various components. For the working example, the output for a 3-anonymized version is shown in Table 6.

While k-anonymity offers protection against identity disclosure, it is insufficient for guarding against attribute disclosure. Specifically, k-anonymity does not impose diversity on sensitive attributes within each equivalence class (i.e., a group of at least k elements indistinguishable concerning their quasi-identifiers). As a result, an attacker can still infer the sensitive attribute of an individual with a high degree of confidence if all k records in the same equivalence class share the same sensitive value. Recall the 3-anonymized version of the example in Table 6. All entries within the first group share the same value for the sensitive attribute

	name	residency	age	diagnosis
1	*	Germany	[25 - 55]	Covid
3	*	Germany	[25 - 55]	Covid
6	*	Germany	[25 - 55]	Covid
2	*	UK	[20 - 45]	Asthma
5	*	UK	[20 - 45]	Covid
8	*	UK	[20 - 45]	Diabetes
4	*	Spain	[30 - 70]	Diabetes
7	*	Spain	[30 - 70]	Asthma
9	*	Spain	[30 - 70]	Diabetes

Table 6: 3-anonymized version of the working example. Suppressed fields are in red cells, and generalized fields are in yellow.

diagnosis. To mitigate this vulnerability, ***l*-diversity** is introduced as an enhancement to k-anonymity. It requires that each equivalence class contain at least l distinct values for the sensitive attributes, thereby adding a layer of protection against attribute disclosure. Concretely, Machanavajjhala et al. define the principle of l-diversity in their paper ”l-Diversity: Privacy Beyond k-Anonymity” [35] as follows: ”A q^*-block is l-diverse if it contains at least well-represented values for the sensitive attribute S. A table is l-diverse if every q^*-block is l-diverse.” In this context, a ”q^*-block” refers to an equivalence class and, in its simplest form, ”well-represented” is to be understood as *distinct* values according to their definition. This is also the definition used by Cao et al. in CASTLE when illuminating their extension of the algorithm proposed by them to allow for the adherence of l-diversity. The extension refers only to a small change within the delta constraint logic of their algorithm as well as a renewed cluster splitting approach. Essentially, it ensures that clusters that are meant to be returned are certain to contain at least l different values for the sensitive attributes. If this is not the case clusters are merged and subsequently bucketized following the values of the sensitive attributes. Should a bucket contain k elements it is ready to be split from the rest and output. Take a look at Table 7, which shows a 2-diverse version of the working example. The groups have been adjusted to ensure that they each contain at least two distinct sensitive values. This has come at the cost of generalizing the *residency* attribute one more time.

Adhering to the l-diversity principle, however, does not protect against attack-

	name	residency	age	diagnosis
1	*	EU	[25 - 55]	Covid
6	*	EU	[25 - 55]	Covid
7	*	EU	[25 - 55]	Asthma
2	*	UK	[20 - 45]	Asthma
5	*	UK	[20 - 45]	Covid
8	*	UK	[20 - 45]	Diabetes
3	*	EU	[30 - 70]	Covid
4	*	EU	[30 - 70]	Diabetes
9	*	EU	[30 - 70]	Diabetes

Table 7: 2-diverse version of the working example. Suppressed fields are in red cells, and generalized fields are in yellow.

ers with background knowledge. Similarity attacks for example, which exploit the inherent characteristics and structural patterns within the data to compromise their anonymity, are still effective. In the context of the 2-diverse table presented earlier (see Table 7), the susceptibility to similarity attack becomes evident. Take, for instance, the first equivalence class characterized by residency in the EU and age between 25 and 55 years. The diagnoses within this class - Covid and Asthma - are both lung-related conditions. An attacker could easily infer that an individual belonging to this group is highly likely to be suffering from a respiratory disease. This is addressed by ***t*-Closeness** as it considers the distribution of sensitive attributes. Li et al., who have coined the term in their paper 't-Closeness: Privacy Beyond k-Anonymity and l-Diversity' [32], define the t-closeness principle as follows: "An equivalence class is said to have t-closeness if the distance between the distribution of a sensitive attribute in this class and the distribution of the attribute in the whole table is no more than a threshold t. A table is said to have t-closeness if all equivalence classes have t-closeness.". This principle serves as an enhanced metric for privacy preservation by addressing the distributional skewness in sensitive attributes that might otherwise lead to privacy leaks. The advantage of t-closeness stems from the fact that the overall distribution of the sensitive attributes is mimicked in each equivalence class, thereby minimizing the changes in attribute disclosure through background knowledge or data patterns. This principle applied to the working example is shown in Table 8. As the overall distribution of respiratory diseases in the original table (Table 5) is 2/3 the entry with row index 4 instead of 7 is exchanged within the first and third group. This leads to a preservation of the distribution at a minor cost of generalization in the *age* quasi-identifier.

	name	residency	age	diagnosis
1	*	EU	[25 - 60]	Covid
4	*	EU	[25 - 60]	Diabetes
6	*	EU	[25 - 60]	Covid
2	*	UK	[20 - 45]	Asthma
5	*	UK	[20 - 45]	Covid
8	*	UK	[20 - 45]	Diabetes
3	*	EU	[30 - 70]	Covid
7	*	EU	[30 - 70]	Asthma
9	*	EU	[30 - 70]	Diabetes

Table 8: T-closeness version of the working example. Suppressed fields are in red cells, and generalized fields are in yellow.

An honorable mention for the table-based anonymization techniques is **Multivariate Microaggregation**. This method extends Univariate Microaggregation by simultaneously considering multiple attributes and aggregating them in a conjunctive manner. To achieve this, it groups records into clusters of at least k similar records, where k is the microaggregation factor, and subsequently replaces each record in the cluster with the cluster's centroid. Although this ensures that no individual record can be uniquely identified, it incurs a substantial computational cost. The computational intensity is such that, to date, no efficient algorithm for this problem has been identified. Oganian and Domingo-Ferrer have proven that multivariate microaggregation is an NP-hard problem [37].

3.3 Model for Anonymization in Distributed Event Stores

The opening chapters of this thesis underscored the rising ubiquity of distributed event stores in the current era of data streaming. It has been observed that the optimization of performance often takes precedence over privacy concerns in the design and implementation of stream processing systems. This has been mostly due to the lack of privacy regulations and the additional computational costs associated with implementing robust privacy measures. It is worth noting, however, that this favoritism is not universal. Especially in the healthcare and government sectors the focus on privacy has always been prevalent. Nevertheless, advances in technology and shifts in regulatory focus are increasingly leading to solutions that aim to balance performance with privacy. The rise of the demand for enhanced privacy in all sectors further underscores the need for the development of robust, efficient solutions specifically tailored to comply with a variety of regulations. Thus, earlier sections of this chapter focused on the need for various degrees of anonymization granularity. Through a use-case example, the practical implications and advantages of different levels of anonymization were shown. In light of these discussions, the two research questions formulated in the Introduction of this thesis gain renewed focus:

1. What techniques can be effectively employed to ensure privacy and security measures, such as Role-Based Access Control and data masking, in distributed event stores?
2. What strategies can be developed and implemented to ensure that these privacy and security measures have a minimal impact on the performance of distributed event stores?

Addressing these questions requires a complex system that meets a comprehensive set of requirements, both functional and non-functional, which will be thoroughly examined in the subsequent subsections.

3.3.1 Architecture

Before getting into any specifics, let us first lay the foundation of the envisioned system architecture. Figure 15 shows a high-level version of the UML Component Diagram, displaying the key components and their interactions.

At the center of the architecture, the Data Officer serves as the primary and sole user of the system. The Data Officer interacts with components of the system via interfaces. In the diagram, the interaction points are represented by lines terminating in either circles or semicircles. These symbolize required and provided interfaces respectively. The role of the Data Officer embodies either an individual or a group within an organization responsible for data management. This role, therefore, depends on the DES as their primary source of data. The Data Officer is reliant on the proper functionality and accessibility of these stores. This dependency is represented by the dotted arrow in the diagram. The people with the

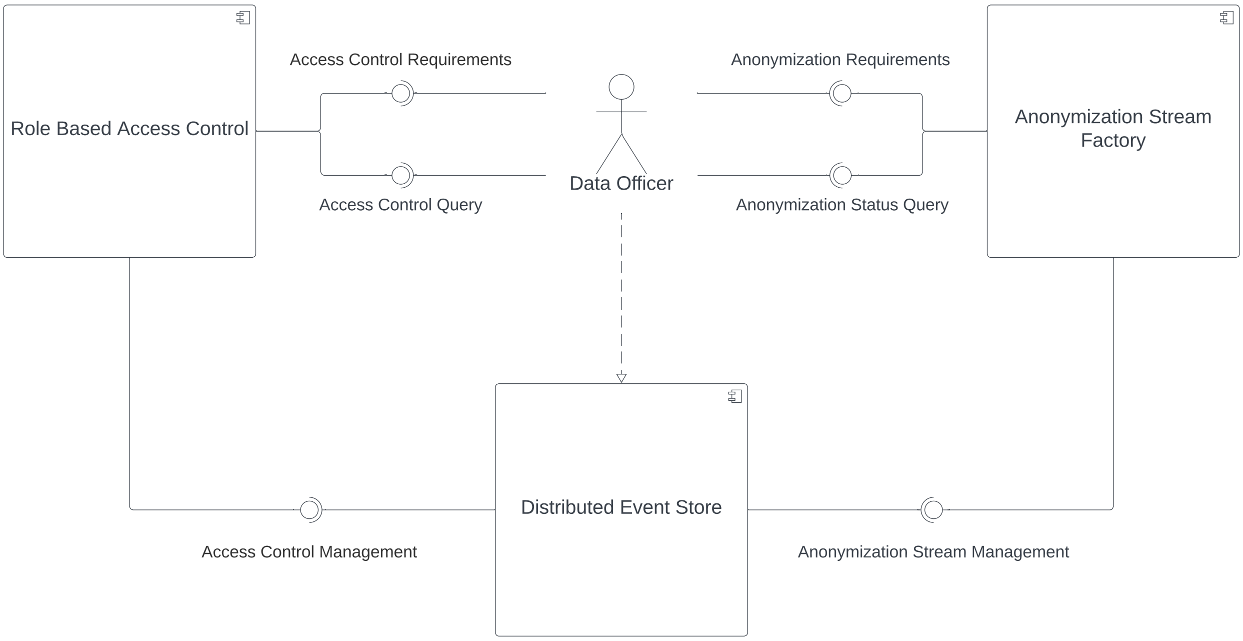

Role Based Access Control
Access Control Requirements
Access Control Query
Data Officer
Anonymization Requirements
Anonymization Status Query
Anonymization Stream Factory
Distributed Event Store
Access Control Management
Anonymization Stream Management

Figure 15: UML Component Diagram of the anonymization system for distributed event stores.

Data Officer role are expected to be technically skilled individuals with extensive knowledge of the data flowing through the DES. This is critical as the Data Officer has the authorization to assign and withdraw access control to all consumers and producers of data within the DES. Additionally, these people will have to think of and define the levels of anonymization appropriate for their use case, a non-trivial task as we have seen in Section 3.1. In the context of the anonymization system, however, the Data Officer does not operate directly with the DES. Instead, these interactions are abstracted away to two intermediate independent components - one responsible for Role Based Access Control and one we call the Anonymization Stream Factory. They communicate with each other by passing requirements and status updates. These are explicitly two different interactions because of their usage. The requirements are meant to be large one-time occurrences at the setup stage of the operation. They dictate the layout of the initial anonymization structure. During runtime, the Data Officer can monitor the processes through queries. This is the other type of interaction. For the RBAC, one can conceptualize the requirements as a collection of individual access control queries. As such the interactions merge on the RBAC side of the system. The interactions for the Anonymization Stream Factory paint a similar picture. The two components process these incoming requests. In the case of the requirements, they transform these into a form, which is interpretable by the management component of the DES.

With a general understanding of the overarching components and interactions, we can look deeper into each component. Figure 16 provides an in-depth look at the component responsible for enabling RBAC. Note that this is not the only possible approach to facilitate role-based access control.

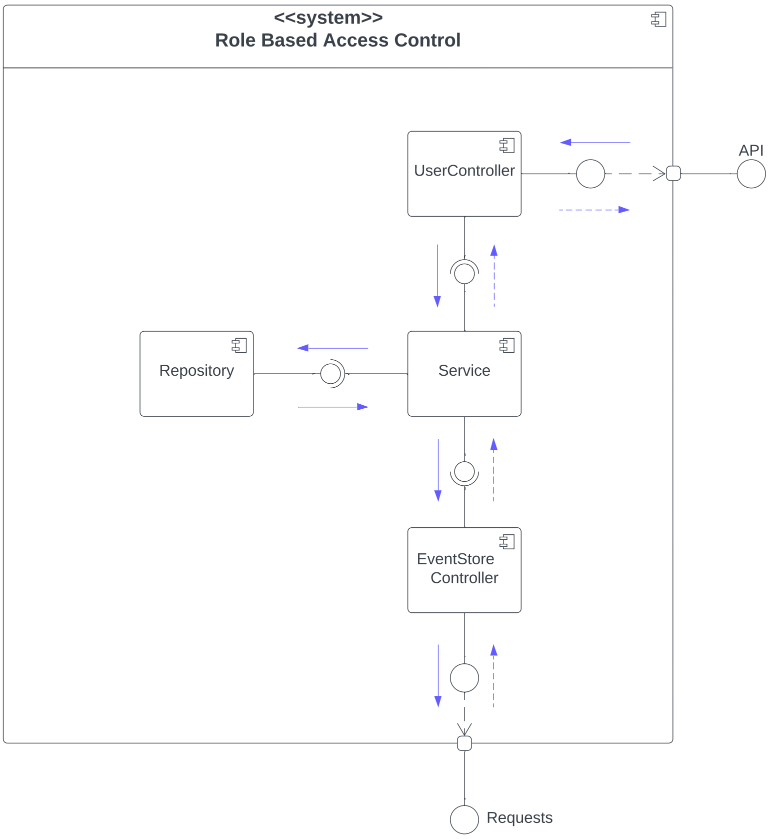


Figure 16: UML Component Diagram of the component responsible for facilitating Role Based Access Control. Additionally, blue arrows have been added symbolizing data flowing. Dashed lines indicate simple response codes.

The Component Diagram shows the system operating with a Controller-Service-Repository setup. The Controller is responsible for interactions with the Client. As there are two Clients, the Data Officer and the Distributed Event Store, two separate controllers distribute the responsibility. Blue arrows in the diagram indicate the data flow. They have been added to the UML diagram to highlight a distinct feature of this system. Data is generally only flowing in one direction.

These instructions, provided by the Data Officer, are translated by the RBAC system into requests directed to the DES. The only data flowing back are simple responses without payload as illustrated by the dashed arrows. The external interactions are handled by the controllers. These then pass on the data to the service. Here the business logic of the system is executed. Incoming requests are addressed and, either transformed and forwarded to the DES or immediately taken care of. Hereby, the repository is utilized. The roles and their access clearance must be saved somewhere. If the DES would already have the role and their access clearance, the RBAC system would be redundant. As this is not the case the management of roles falls to the responsibility of this component. An API request for the addition of a role would be handled by the RBAC component and by this component alone. The modification of a role, however, would need to be forwarded to the DES if a user occupies this role. It would convert this role modification into modified Access Control Lists of each user with that role and pass these on. Additions or modifications to users are stored within the component and also forwarded, for analogous reasons. Altogether this leads to an overall better user experience for the Data Officer. They would only have to deal with the abstraction of access control lists in the form of roles, simplifying requests and making them less error-prone in the process. One important additional functionality the RBAC component has to offer is the assignment of roles to the consumption of streams. Users will be able to access different versions of the data depending on their role. The access control aspect of this will be managed by this component as well. The functionality of providing differently anonymized versions of data is provided by another component - the Anonymization Stream Factory.

The Anonymization Stream Factory is the heart of the system. Its Component Diagram is depicted in Figure 17. Within this component, the specifications provided by the Data Officer are implemented. After thoroughly analyzing their data needs, a plan detailing the anonymization granularity was formulated and transformed into a comprehensive set of requirements. These requirements are subsequently transmitted to the entry point of the Anonymization Stream Factory. The requirements include all the necessary information of the stream that is supposed to be anonymized. It also includes the wishes for the various levels of anonymization. Together they are sent to the entry point of the anonymization stream factory.

Internally, these are received by the system's central component, the Manager. This component is responsible for external communication as well as internal communication. With that, its main task is forwarding data. First, the requirements are passed on to the Requirements Processor. As the requirements are expected to come in one request, These requirements are translated into a format comprehensible to this component and then deconstructed into their parts. Remember,

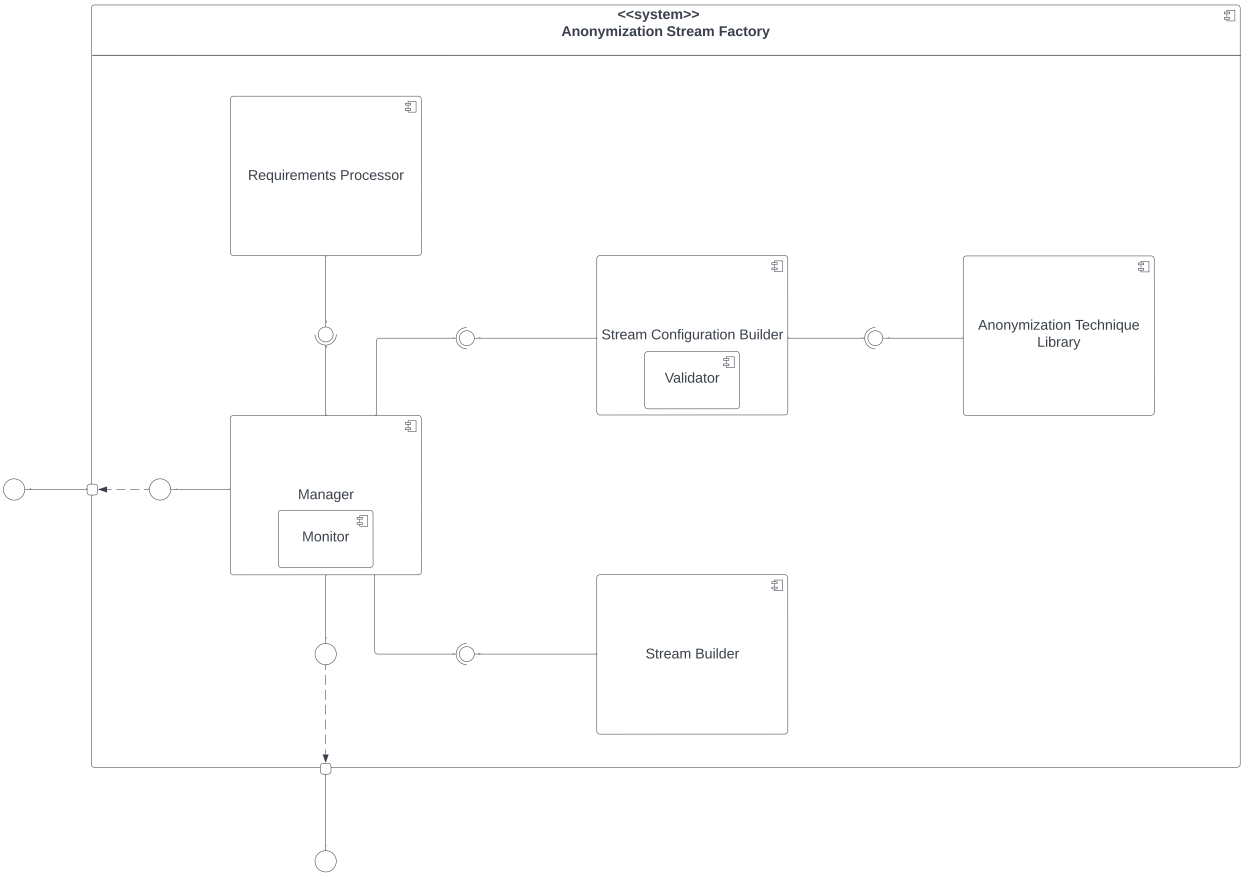

<<system>>
Anonymization Stream Factory
Requirements Processor
Stream Configuration Builder
Validator
Anonymization Technique Library
Manager
Monitor
Stream Builder

Figure 17: UML Component Diagram of the anonymization system for distributed event stores. Additionally, blue arrows have been added symbolizing data flowing. Dashed lines indicate simple response codes.

these are the original stream configuration and the various anonymized stream desires. These desires will then be passed individually on to the Stream Configuration Builder. It verifies the availability of the desired anonymization techniques for the stream by interacting with the Anonymization Technique Library. An immutable data component within the system. In Section 2.3 we have seen that each anonymization technique can be finetuned through parameters. While these can be syntactically checked with the requirements processor, their semantics need to be validated as well. For this, the configuration builder must include some sort of validation component. The validation must be properly done at this stage as catching these errors should be done before the stream's runtime. They should also be well communicated back to the Data Officer. Regardless of the outcome, the results are relayed back to the Manager. Either with an error message, that is to be forwarded back to the user or with the defined stream configurations. These then can be passed on to the Stream Builder. Here the actual streams are built, they will consume the original data stream and transform it according to the stream configuration, then produce a new anonymized stream. This ready-to-go stream will be passed back again to the Manager. Subsequently, the Manager forwards the processed stream to the DES. It should either be able to monitor the status of the streams by itself or answer the status queries by the data officer with forwarded messages from the DES.

Finally, let us take a look at the internal architecture of the Distributed Event Store. Figure 18 shows its Component Diagram. At the core are the individual Event Stores. Each is responsible for storing and managing data (in this context also often to referred as "Events"). They are distributed, forming a loose cluster of individually not connected stores. They can be spread across multiple physical machines and even different geographical locations. The Event Stores are controlled by the administration component of the system. It monitors and logs the stores as well as gives commands like adding or removing individual event stores. The administration also decides on the partitioning and aggregation logic. The main benefits of a distributed system are scalability and reliability. It accomplishes horizontal scalability with ease through the extension of additional event stores. The workload is then balanced across these stores to mitigate bottlenecks. Reliability is facilitated through replicability. If any one event store fails, there is another with replicated data, which can take its place until the original event store is back online. This leads to increased fault tolerance, data durability, and higher availability. All this relies on the proper replication and distribution logic. The strategy is chosen by the Administration and forwarded to the Partitioner component to put into effect. It takes the data collected by the producers and shards it e.g. splits it into multiple pieces. Shards are replicated and sent to other destinations. The Aggregator does the composite process. From the Administration, it is familiar with the distribution logic and can piece together the data from multiple sources into one output. This output is relayed to the consumers of the data.

The Administration component is configurable from the outside through an interface. Normally, there is only one expected interface. The reason for the two interfaces shown in the diagram is simply due to it being a part of a bigger Component Diagram shown in Figure 15. It is administrated from two other components, the RBAC system and the Anonymization Stream Factory, both dictated by the Data Officer. The full Component Diagram spans multiple pages and can be found in Appendix 26. In reality, it would be safe to assume that the other components interact with the Distributed Event Store through one administration interface. Through this interface, for example, access control can be assigned or data streams

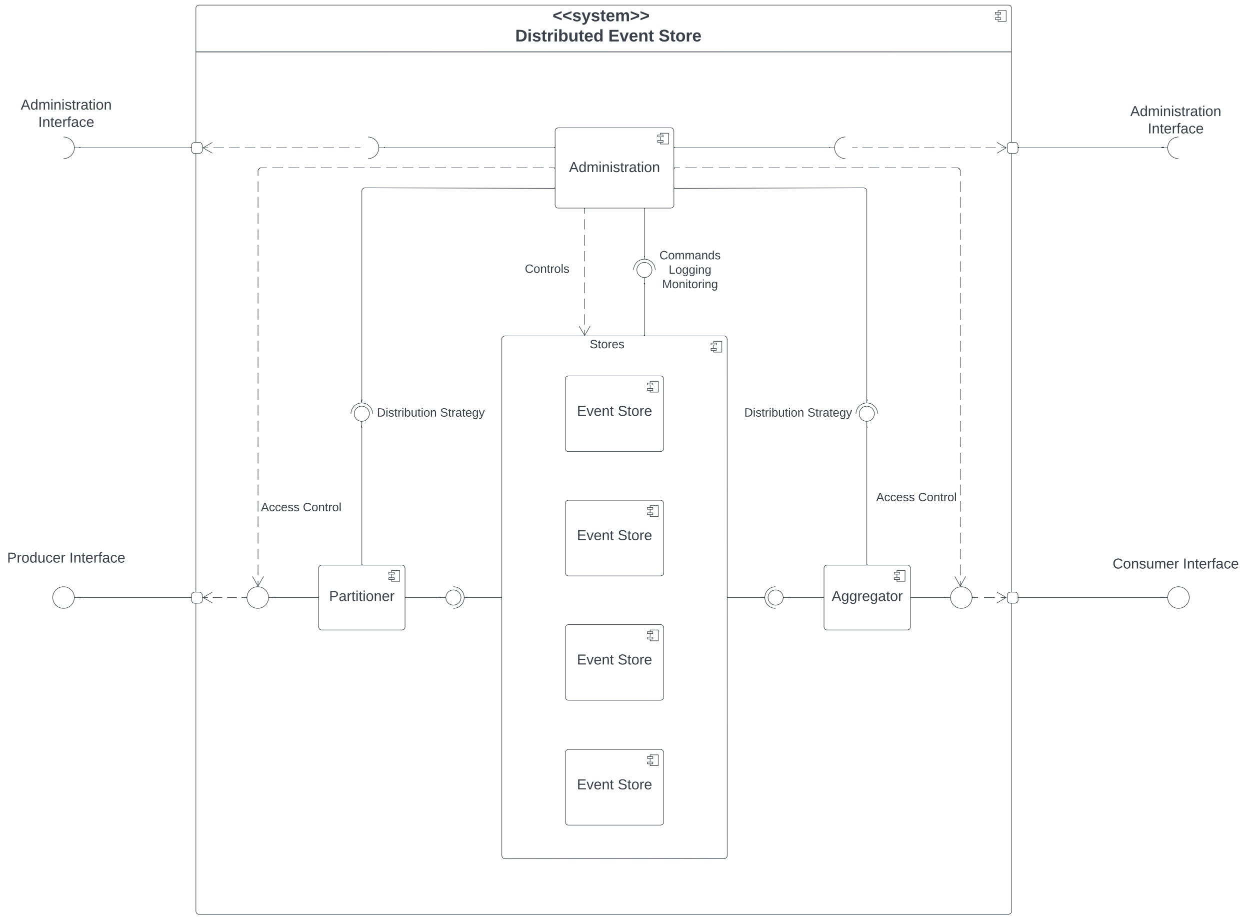


Figure 18: Architecture of a Distributed Event Store illustrated with a UML Component Diagram.

added. The access control would then be enforced at the system's data ports.

3.3.2 Functional Requirements

The architectural layout presented until this point describes the system as a whole, its components, and their interactions, thereby serving as a blueprint for the envisioned infrastructure. Let us translate this vision into actionable and measurable goals:

1. **Data Stream Integration**
   Distributed Event Stores are specialized towards operating on streaming data; appropriate integration is essential to the success of the system. The

system is anticipated to receive streaming data as the input that requires anonymization. Consequently, the anonymized output data must likewise be in a streaming format. The system must facilitate the ingestion, processing, and management of data streams in real-time, ensuring both the efficiency and reliability of data handling.

2. **Administrative Control**
   The goal of the system is to provide granular anonymization to its user, the Data Officer. Here, the system must provide tools for the Data Officer to govern the system's operation. This specifically includes access control management and oversight of the data processing pipeline. The system must incorporate robust monitoring tools and provide interfaces for the user.
3. **Adaptability**
   With any large software endeavor comes changes in requirements over time. The system must be adaptable to that change. New anonymization techniques must be addable to the system. Existing ones must be customizable. It must be possible to modify the system ports to enable connections to different distributed event stores. The system's architecture must be designed with flexibility at its core, accommodating changes in requirements and technological advancements with minimal disruption.

3.3.3 Non-functional Requirements

Finally, let us address the non-functional requirements. Although non-functional requirements do not dictate system functionality, they are imperative for the operational integrity of the system. Reiterating that the strengths of Distributed Event Stores lie in their ability to efficiently handle large amounts of data in real-time, the addition of the proposed system mustn't diminish these. Therefore, the three main non-functional requirements of the system are essential to maintain the strengths of the underlying distributed event stores:

1. **Performance**
   The system should minimize performance overhead. This can be evaluating the latency and throughput with and without the anonymization system in place. Additionally, storage is always a concern. However, given the fact that the system's central idea is duplicating the data and providing different anonymized versions through different streams, much more storage capacity can be expected.
2. **Reliability**
   The system must ensure the availability of anonymized data counterparts

concurrently with non-anonymized data. System failure carries significant risks. Furthermore, it must be guaranteed that non-anonymized data is under the strict proposed access control rules. The plain text data being available to unauthorized persons would jeopardize the entire system.

3. **Scalability**
   It should be possible to easily scale the system horizontally, mirroring the core attributes of distributed systems. Accommodating increasing data loads and parallel processing demands without compromising the efficiency of the anonymization process, is what the system should strive to do.

4 Implementation

The previous chapters have established the foundation for this thesis. The need for anonymization granularity in distributed event stores has been discussed in chapter 1. The existing literature and research were provided in chapter 2 with an emphasis on anonymization techniques specifically for streaming data. Chapter 3 served as an outline of the expected functionality of the system. This chapter focuses on the practical steps taken to materialize the aforementioned system, detailing the technological choices, development processes, and the resulting product.

This chapter begins with an explanation of the choice of Apache Kafka as a Distributed Event Store, Kafka's architecture, and where the implementation ties in. The subsequent section will focus on the administration of Apache Kafka. This administration is managed through the Apache ZooKeeper framework. Abstracting from the existing ACLs to enable Role Based Access Control is achieved by the RBAC system. How this is done will be the center of the next section. Next, the focus shifts to the central component of the implementation: the Anonymization Stream Factory. We will go in-depth on its various components, illuminate the thought process, highlight its core features, and explain how it can be adapted going forward. For effective testing of the system, a data pipeline has been constructed using the aforementioned components. To complete the pipeline we have added a Data Generator, a Kafka Connector, and a Kafka Consumer. They will be the focal point of the Data Pipeline section. Finally, the concluding section will demonstrate the integration of these components into a cohesive system, facilitated by Docker for ease of use and deployment.

4.1 Distributed Event Store - Apache Kafka

Out of the multitude of distributed event stores available, Apache Kafka has been selected for this research. Its scalability and reliability, while maintaining low latency and high throughput have established Kafka as the de facto standard in message broking. Combined with its compatibility with essentially all stream processing frameworks, albeit offering its stream processing framework Kafka Streams, Kafka is one of the leaders in big data processing technologies worldwide. Its utilization spans multiple sectors, including Fortune 100 companies, governments, and healthcare. Furthermore, as an open-source technology, Apache Kafka offers a rich set of tools and an active community. This allows a more thorough exploration of practical implementations simultaneously staying grounded in real-world applicability. Although there are noteworthy competitors such as Apache Hadoop, Amazon Kinesis, or RabbitMQ, ultimately Kafka aligns best with the requirements at hand, particularly due to its extensive support for research and development endeavors.

4.1.1 Architecture

So let us take a look at Kafka's architecture shown in Figure 19. There is quite a bit of Kafka terminology involved, in the following written in cursive.

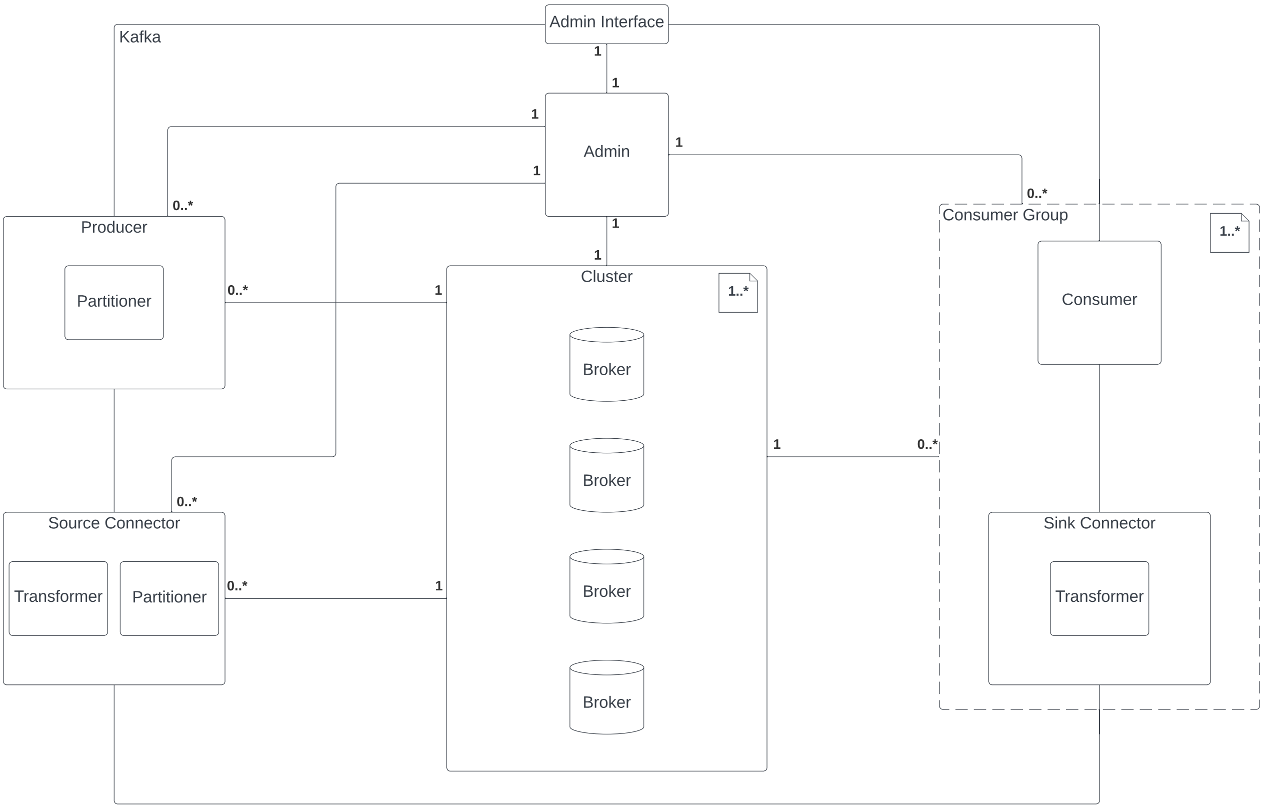


Figure 19: Apache Kafka Architecture and Terminology. The links between components have been enhanced with cardinalities to provide more context.

At the core is the *Cluster*, containing at least one *Broker* (database), this is denoted with the note in the top right corner reading '1..*'. As a cluster, they function as distributed storage. However, they deal exclusively with streaming data. A singular stream of related events is termed *Topic*, produced and distributed across the individual Brokers according to strategies delineated by the administration component. For topic production, a *Producer* must be developed by the user to format the raw data. Alternatively, users can employ *Connectors* that interface external systems with Kafka, thus relieving users of data transformation tasks. These connectors are available in a plug-and-play fashion, and the *Kafka Connect* framework allows for the creation of custom connectors. *Consumers* retrieve topics, operating collectively within *Consumer Groups*. Kafka provides three distinct delivery semantics — 'at least once', 'at most once', and 'exactly once' — which refer to how often a singular event is delivered to each consumer group. On the consumer side, similarly to Producers, a *Kafka Consumer* requires user development, although connectors for integrating Kafka with external systems are available. Note that custom Consumers and Connectors can coexist in on Consumer Group. The administration is available through a *Kafka Admin* framework. However, simply using the Kafka Admin framework requires significant development overhead. Instead, other frameworks are brought in to adopt this task.

4.1.2 Administration

There are several tasks that the administration unit of Kafka has to accomplish. These administrative tasks must be made configurable for the system administrator within the Kafka environment. The responsibilities include the following:

1. **Cluster configuration**
   It must be in close communication with the brokers. It must be possible to add and delete individual brokers to the cluster.
2. **Topic Control**
   Topics must be creatable and deletable. Topics are associated with producers and consumer groups. In particular, the offsets for consumer groups must be monitored. Typically, this is done in an additional "consumer offsets" topic. Here consumer groups are mapped to their offset. Consumer groups periodically publish on this specific topic whenever they consume messages from their respective topics.
3. **Partition Strategy**
   The partition strategy for producers must be communicated. Partitions need to be assigned to brokers.

4. **Replication Strategy**
   To ensure reliability partitions need to be replicated. There is a partition leader stored on one broker and replications on several other brokers. The administration must be able to select the replication strategy.
5. **Access Control**
   Access control must be enforceable for producers, consumers, and particularly for administrative operations.
6. **Fault-tolerant**
   The administration component itself must be robust and reliable.

While technically all these requirements can be met simply using the Kafka Admin framework, Kafka themselves do not recommend it. Up until Kafka Version 3.3 (released in August 2022), Kafka advised the delegation of the administration to another framework called *Apache ZooKeeper*. ZooKeeper is an open-source centralized service for synchronizing distributed systems, maintaining configuration information, and providing group services. It facilitates the communication of the various components with each other through hierarchical namespaces. Each namespace corresponds to a node within a hierarchical tree structure. A node on its own can simultaneously have children and data. As ZooKeeper is designed for configuration information the data associated with a node is expected to be small (Byte to Kilobyte in size). For instance, ZooKeeper creates nodes for each broker and stores its configuration as data. In a separate node, it stores information about each topic including its partitions, the assignments of partitions to brokers, and the replication logic. These correlations enable ZooKeeper to maintain system operations and adapt to changes. Each node also encompasses a ACL that governs the access to the node's data and its children. The change of the ACL of a topic node for instance would affect the access control for the consumption of that topic. ZooKeeper relies on replication to ensure fault tolerance. In addition, watches are created to detect failures in individual nodes. With all this in place, ZooKeeper makes the following guarantees: Sequential Consistency, Atomicity, Single System Image, Reliability and Timeliness. Altogether, it provides all responsibilities as described in the prior.

Apache Kafka, however, has created its own administration component called *KRaft*. Instead of administering the cluster externally, it delegates this responsibility to the brokers. They store the necessary metadata for maintenance and administration alongside the topic partitions. Again replication and partition are utilized to ensure durability. This approach significantly reduces overhead, as it obviates the need to maintain and store separate ZooKeeper nodes. KRaft has been introduced in Kafka Version 3.3 and is production ready as of Version 3.5 (released June 2023). However, it may be reasonably anticipated that the adoption of this new administrative approach within production servers will occur over several years.

4.1.3 Configuration

In the Implementation, we use the current newest stable version of Kafka 3.5.1. We opt to use ZooKeeper as Administration as in current production environments it is the standard. For the test suite, a data generator is created, and a custom Kafka Connector is developed to produce data for the cluster. Additionally, a Kafka Consumer is instantiated to validate the anonymized output. The implementation employs the Admin, Connect, and Streams APIs, all of which are version 3.5.1. Further details on the implementation will be provided in the appropriate sections.

4.2 Role Based Access Control

This section presents the implementation of a system designed to enable the Role Based Access Control (RBAC) policy for Apache Kafka, as directed by the Data Officer. It follows a streamlined variant of the Repository-Service-Controller software development architecture. It includes a Controller for Kafka, a Command Line Interface (CLI) for the Data Officer as well as a database - but skips the service in favor of more straightforward transactional processing.

Inherently, Kafka incorporates mechanisms that enforce access control on specific resources or their groups. However, the specification of policies is limited to ACLs in Kafka. Nevertheless, Kafka allows changes to these ACLs to be administered programmatically by an authenticated and authorized user. This is where our RBAC system comes into play.

The system abstracts Kafka's Object-oriented access security model, realized through ACLs, transitioning it to a role-based access control paradigm. Here, the authorized permissions of actions on objects are no longer defined for subjects e.g. users, but to roles. Subsequently, roles are assigned to users, granting them the cumulative permissions associated with each assigned role.

Realizing this abstraction requires memory of the mappings of permissions to roles, users to roles, and overall users as well as roles. To achieve this, the RBAC system incorporates an SQLite [48] database, consisting of three tables: `Users`, `Roles`, and `UserRoles`. We chose SQLite for its in-memory data storage, its simplicity in both administration and development - and its high reliability. The permissions are stored as text in the `Roles` table alongside the role name and an autoincremented unique role ID. The username is stored in the `Users` table together with an autoincremented unique user ID. This relationship is represented in the `UserRoles` table, where both role and user IDs function as foreign keys, and the combination of userId and roleId forms the primary key. This allows one user to be assigned to many roles.

Administrative operations are executed by the Data Officer via a CLI, developed using the JCommander framework [5] and accessed through the `rbac_cli.sh` shell script located in the system's root directory. The commands in Table 9 are available to the Data Officer:

Command	**Parameters**
addUser	`--userName <userName>`
deleteUser	`--userName <userName>`
addRole	`--roleName <roleName> --permissions <topic1,...>`
deleteRole	`--roleName <roleName>`
addPermission	`--roleName <roleName> --topicName <topicName>`
removePermission	`--roleName <roleName> --topicName <topicName>`
assignRole	`--userName <userName> --roleName <roleName>`
removeRole	`--userName <userName> --roleName <roleName>`
userStatus	`--userName <userName>`
roleStatus	`--roleName <roleName>`
roles	Lists all roles
users	Lists all users
userRoleStatus	Displays the status of user roles
exit	Exits the system
help	Prints this table

Table 9: List of Available Commands in the RBAC System

It is important to note that in this context, permissions correspond directly to topic names. The reason is that in our application solely the consumers of topics are restricted. A permission here can therefore be understood as the permissible action 'consume' for the object 'topic' by the subject 'user'.

The commands originate at the Data Officer, find their way into the RBAC system through the CLI, and are applied to the database by a dedicated database manager. When changes to the database are made, the database manager also looks for changes in permission in the database before and after the transaction. If there are changes, these get forwarded to a separate controller - named `KafkaController`. Here, the changes arrive in the form of two parameters - a HashMap of Strings (user name) and List of Strings (permissions) as well as a boolean (whether constructive or destructive change). These are then transformed into ACLs and forwarded to Kafka.

To establish a connection with Kafka's administrative interface, capable of realizing changes on the Kafka server, specific configurations are mandated. First, authorization and authentication must be enabled. Note how this is not the default behavior of Apache Kafka. Additionally, access control lists are enabled in Zookeeper to enact the access control mechanisms within Kafka. There are different authentication techniques available including Kerberos, SSL, and Simple Authentication and Security Layer (SASL). We opted for SASL plaintext authentication for simplicity. Each Client and Server in the Kafka network then must provide a Java Authentication and Authorization Service (JAAS) configuration file to authenticate against the network. These are included in the run scripts of both Zookeeper and the Kafka Server as well as all Clients. The JAAS configuration files crucially include username and password for the network. Furthermore, the user specified in the JAAS configuration file of the RBAC system must be registered as a superuser in the server properties of Kafka to be allowed to administer ACL changes.

With proper authentication and authorization in place, role-based access control can effectively be facilitated seemingly straightforwardly by the Data Officer for Apache Kafka.

4.3 DASH - Data Anonymization Stream Handler

Integral to the system is the anonymization of the data stream. This led to the conceptualization of the Anonymization Stream Factory, as introduced in Section 3.3.1. Its main tasks are threefold: First, it must understand the anonymization granularity relayed in the requirements input by the user. Second, it must apply these requirements to separate anonymized streams. Third, it must provide an interface to monitor and manipulate the streams. Additionally, the success of this component is critically dependent on its reliability, adaptability, and performance. To fulfill these tasks, the Data Anonymization Stream Handler (DASH) was developed. The following subsection will provide a comprehensive analysis of DASH, addressing its functionalities, interactions, and thought processes that went into its development. DASH was developed with Java (Version 11), selected due to Kafka's implementation in Java, thus offering more comprehensive and up-to-date support. A UML Class Diagram was created and will assist in explaining the implementation. It can be found in full in Figure 25 in Appendix A. However, as it spans multiple pages, we will only include excerpts of it in the subsections.

4.3.1 Overview of Anonymization Techniques

In chapter 2 we have detailed various anonymization techniques. In the subsequent chapter 3 we have categorized them into more comprehensible groups. Based on that we have implemented the vast majority of anonymization techniques and made them available in DASH. They are elaborated in Table 10. The anonymization techniques are color-coded to show their corresponding anonymization category. Each anonymizer needs some parameters specified for configuration, these are detailed in the table as well. This includes their format or type, whether they are required for the configuration or optional, and a short description of the parameter. Among the value-based anonymization techniques, highlighted in blue, we implemented all but one in DASH, Tokenization. Tokenization requires a database as an input parameter and the development overhead for the creation, validation, and testing of such a database was not in the scope of this thesis.

The green highlighted tuple-based anonymization technique conditional substitution is included in DASH. This technique supports three distinct kinds of conditions: value matches, numerical ranges, and regular expressions, thereby enhancing its versatility in application.

The red highlighted implemented attribute-based anonymization techniques include Aggregation, its specialization Univariate Microaggregation, and Shuffling. As attribute-based anonymization techniques, they operate on a collection of tuples. DASH creates discrete sets of tuples by cutting up the data stream, a technique called windowing. Kafka supports this inherently. There are different kinds of windows available. DASH can be configured to work with different kinds of windowing techniques per anonymization stream. The available window types are tumbling and sliding. Both are fixed size, as specified by the required windowSize parameter. Sliding in comparison to tumbling allows the overlapping of windows. A tuple in a sliding window can therefore be included in more than one window, whereas in a tumbling window, it can only be included in one. By setting only the windowSize parameter, the user specifies the window size of a tumbling window. The specification of the optional advance time turns that into a sliding window that moves along the time axis according to that parameter, creating new windows for every advance time unit. Additionally, a grace period can optionally be set. As data streams operate in real-time tuples can be delayed in their transition to the system. A late-arriving tuple can be identified by its associated timestamp and stream position. If the grace period is specified a window waits that amount of time before forwarding the tuples to processing to allow for latecomers to be included. Grace periods are available for both tumbling and sliding windows. These window configurations apply to all attribute-based anonymization techniques. Aggregation and Univariate Microaggregation operate on numerical attributes and aggregate the attributes specified in the 'keys' parameters within a window. Aggregation offers various modes: sum, median, average max, min, count, and mode. It replaces the values of these attributes with the computed values of the specified modes (count refers to the number of tuples within the window; mode refers to the most frequent value within the window). The Univariate Microaggregation is implemented as detailed in Algorithm 1. Shuffling can be additionally configured with a seed, making it deterministic. In the case that there are multiple keys to shuffle it can both shuffle them together, ensuring that the content of these attributes remain together after the shuffle, or independently.

Finally, the table-based anonymization technique is highlighted in yellow. It includes the implementation of CASTLE's k-anonymization as specified in 3.2.4. Here, the tuples are processed one at a time instead of windowed as in the attribute-based anonymization techniques. Even though they are entering DASH one at a time, they are grouped in static clusters and are only released after expiration as determined by the delta parameter in CASTLE. The implementation requires the Data Officer to specify multiple parameters. Most are simply positive integers defining different aspects of the algorithm. Additionally, the attributes containing PII are included to be suppressed. Finally, the quasi-identifiers have to be specified. Remember, these are the attributes that the k-anonymization is applied to. Any set of k entries in the output stream must be indistinguishable concerning their quasi-identifiers. These are generalized and not suppressed to minimize information loss. The 'quasiIdentifiers' parameter is a List containing a String and GeneralizationHierarchy for each quasi-identifier. The String is the attribute name. The 'GeneralizationHierarchy' is a special data structure designed for the implementation of CASTLE. It can take on two forms for the two types of attributes - numerical and categorical. A numerical generalization is specified through a numerical range as well as the bucket size and in its generalization logic equals bucketizing. The categorical attribute requires a generalization tree. Each node has two attributes - the value, e.g. the generalization of that node, and its children, e.g. an array of nodes. DASH then applies the algorithms described in CASTLE. The extension to l-diversity was not implemented in DASH, but it already provides the bulk of the code necessary including various data structures and methods. However, it was not within the scope of this thesis. Similarly, further extensions to facilitate t-closeness were also excluded from the scope of this thesis. It was decided against implementing Multivariate Microaggregation as there is no optimal algorithm available at this time.

Table 10: List of Anonymizers available in the Data Anonymization Stream Handler (DASH). They are listed with their respective category color coded as well as their parameters including the parameter type and whether it is required or not. A description of the parameters is also added.

Category	**Anonymizer**	**Parameter**	**Type**	**Required**	**Description**
ValueBased	Blurring	keys	List<String>	✓	List of attribute names to blur
		nFields	positive integer	×	Number of fields to blur
	Bucketizing	keys	List<String>	✓	List of attribute names to bucketize
		bucketSize	positive integer	✓	Size of each bucket
	Generalization	keys	List<String>	✓	List of attributes to generalize
		generalization-Map	HashMap <String, String>	✓	Generalization for each value
	NoiseMethods	keys	List<String>	✓	List of attributes to apply noise
		noise	positive double	✓	Amount of Noise; typically between 0 and 1
	Substitution	keys	List<String>	✓	List of attributes to substitute
		substitutionList	List<String>	✓	List of substitutes
	Suppression	keys	List<String>	✓	List of attributes to suppress
TupleBased	Conditional Substitution	keys	List<String>	✓	List of attributes to substitute

Continued on next page

Table 10 – *Continued from previous page*

Category	Anonymizer	Parameters	Type	Required	Description
		conditionMap	HashMap <String, Object>	✓	Conditions mapped to the attribute they correspond to. Conditions can be of three different formats: value matches, numerical ranges, and regular expressions
AttributeBased	Aggregation	keys	List<String>	✓	List of attributes to aggregate
		aggregationMode	String	✓	Type of numerical aggregation performed on the values. The options are "sum", "median", "average", "max", "min", "count" and "mode"
		windowSize	positive integer	✓	Duration of a window in ms.
		advanceTime	positive integer	×	Duration of the window's advance time in ms.
		gracePeriod	positive integer	×	Duration of the window's grace period in ms.
	Shuffling	keys	List<String>	✓	List of attributes to shuffle
		seed	positive integer	×	Specify the randomizing seed.
		shuffleIndividually	boolean	×	Shuffle the specified attributes individually or collectively e.g. disrupt the correspondence of the attributes that are shuffled.
		windowSize	positive integer	✓	Duration of a window in ms.

Continued on next page

Table 10 – *Continued from previous page*

Category	**Anonymizer**	**Parameters**	**Type**	**Required**	**Description**
		advanceTime	positive integer	×	Duration of the window's advance time in ms.
		gracePeriod	positive integer	×	Duration of the window's grace period in ms.
	Univariate Micro-aggregation	keys	List<String>	✓	List of attributes to microaggregate. They will be aggregated individually as this is univariate
		k	positive integer	✓	Number of minimal observations per subgroup
		windowSize	positive integer	✓	Duration of a window in ms.
		advanceTime	positive integer	×	Duration of the window's advance time in ms.
		gracePeriod	positive integer	×	Duration of the window's grace period in ms.
TableBased	K-Anonymization	keys	List<String>	✓	List of attributes to suppress e.g. PII
		quasiIdentifiers	List<String, Generalization-Hierarchy>	✓	List of quasi-identifiers with their attribute and corresponding generalization hierarchy structured as a tree
		k	positive integer	✓	Number of tuples indistinguishable in regard to their quasi identifiers
		delta	positive integer	✓	Number of subsequent tuples until expiration

Continued on next page

Table 10 – *Continued from previous page*

Category	Anonymizer	Parameters	Type	Required	Description
		mu	positive integer	✓	Number of output clusters that define the average information loss parameter τ within CASTLE
		beta	positive integer	✓	Maximum amount of clusters

Internally, DASH categorizes the anonymizers with the usage of interfaces. There are four distinct interfaces, one for each of the four anonymization categories. Then there is the overarching Anonymizer interface, which dictates all anonymizers. This interface mandates the implementation of four methods. The primary method, `anonymize(lines : List<Struct>)`, feeds input data streams to the anonymizers and yields the anonymized output. The `Struct` data structure is what DASH uses internally. It is Kafka native and is similar to a tuple in that it splits the tuple into its attributes, which can be accessed and modified in place through its indices as well as the attribute name. The latter is what DASH uses primarily and which is the reason it requires the List of attributes as the anonymizer parameters. Alternatively, indices could have been used synonymously. Note, value, and tuple-based anonymizers receive and return only one tuple at a time, attribute-based receive and return collections of tuples, and table-based receive one tuple at a time but return empty lists or collections of tuples courtesy of CASTLE. Owing to these semantic variations, DASH necessitates that anonymizers within the same stream belong to the same category. The anonymizers are connected in series so the output of one anonymizer will be the input of the next. If their semantics were different, this would not work. To ensure category compliance the anonymizer interface requires the implementation of the `getAnonymizationCategory()`. This method is not implemented by any anonymizer - but defaulted by their category interface. This validation process is handled by the stream configuration builder, a component that will be discussed in greater detail later in this section. For further validation, the anonymizer interface includes the method `getParameterExpectations()`. Each anonymizer relies on different specific parameters that need or can be configured before runtime. These are specified in their implementations together with validation logic syntactically and semantically validating the input configuration. How this works will also be addressed in-depth in the subsequent subsections. The anonymizers are made available through a static class `AnonymizerRegistry`. It includes a HashMap of all anonymizer names and their classes. From here, the stream configuration builder can get access to the anonymizers' methods, validate them, and ultimately with their last method `initialize(parameters : List<Parameter>)` configure them. This structure was designed to facilitate further adaptations. To include a new anonymizer the developer has to decide on an anonymization category and create a class implementing the corresponding interface. The inclusion of the anonymizer's name and class in the registry completes this integration process. From then on it will be available in DASH.

4.3.2 Parsing and Analysis of Anonymization Requirements

This subsection begins with a dissection of the anonymization requirements, made up of two distinct components: the underlying data stream and the anonymization granularity built upon it. Together they contain all information necessary for the anonymization process. The requirements are encapsulated in a JSON file format. We chose JSON because of its widespread use and familiarity among technical users. JSON's format, with its ability to handle attribute-value pairs and arrays, allows efficient parsing and representation of the requirements. Code Fragment 1 shows an exemplary configuration setup.

```
{
       "globalConfig": {
          "bootstrapServer": "STRING",
          "topic": "STRING",
          "dataSchema": "AVRO" or  "KAFKA_STRUCT"
       },
        "streamProperties" : [
          {
             "applicationId": "STRING",
             "category": "ENUM (VALUE_BASED, TUPLE_BASED,
                ATTRIBUTE_BASED, TABLE_BASED)",
             "anonymizers": [
                   {
                      "anonymizer": "STRING (suppression,
                         substitution, etc.)",
                      "parameters": [
                         {
                               "keys": [
                                  "key": "STRING"
                                  // ... other keys
                               ],
                               // ... other parameters
                         }
                      ]
                   },
                   // ... other anonymizers
                ]
            },
            // ... other stream configs
        ]
}
```

Code Fragment 1: Example JSON Configuration for DASH. The *globalConfig* specifies the underlying data stream. The *Schema* refers to either an *AVRO* or *KAFKA_STRUCT* data schema. Each entry in the *streamProperties* defines an anonymized version of the original data consisting of a list of *anonymizers*.

We named the first component global config as it includes valuable information for all streams. The bootstrap server, defined as the IP address of the Kafka server, the topic is the unique name of the stream that is to be anonymized - both are strings. The data schema, highlighted in green, defines the attributes and types of the data within the stream. Here, DASH supports two data schema types: Kafka Struct and Avro. They are among the most popular schemas in use for Kafka. For the use case example of Section 3.1 the schemas would be defined as shown in Figure 20.

```
{
  "type": "AVRO",
  "name": "Patient",
  "fields": [
    {"name": "pid", "type": "int"},
    {"name": "name", "type": "string"},
    {"name": "zip", "type": int},
    {"name": "sex", "type": string},
    {"name": "age", "type": int},
    {"name": "ins. co.", "type": string},
    {"name": "ins. no.", "type": string},
    {"name": "diag.", "type": string},
    {"name": "gluc.", "type": double},
    {"name": "hba1c", "type": double},
    {"name": "med.", "type": string}
  ]
}
```

```
{
  "type":  "KAFKA_STRUCT" ,
  "fields": [
    {"field": "pid", "type": "int32"},
    {"field": "name", "type": "string"},
    {"field": "zip", "type": int32},
    {"field": "sex", "type": string},
    {"field": "age", "type": int16},
    {"field": "ins. co.", "type": string},
    {"field": "ins. no.", "type": string},
    {"field": "diag.", "type": string},
    {"field": "gluc.", "type": double32},
    {"field": "hba1c", "type": double32},
    {"field": "med.", "type": string}
  ]
}
```

Figure 20: Example Avro (left) and Kafka Struct (right) schemas for the patient use case.

The second component of the requirements file details the different anonymized streams. The array can include any number of elements but must not be empty. Each stream is given an 'applicationId' equalling a name. This is important - because the combination of the original topic name with applicationId is the topic name of the resulting data stream. Next, the stream is associated with a category. Remember, in Section 3.2 all anonymization techniques were categorized. An individual anonymized stream will only be allowed to contain anonymizers falling into that category. This addition to the requirements is mainly there to remind the user, the Data Officer, to remain in one category. It is optional and can be inferred by the system based on the rest of the stream property. Then come the anonymizers. Each is specified with a name and equipped with parameters. The list of available anonymizers and their respective parameters can be found in Table 10. Extensive documentation, as well as exemplary requirements, can be found alongside all other artifacts, the codebase, and the thesis itself under `https://github.com/TheRealHenri/master_thesis`.

Next, let us look at how DASH parses and transforms the requirements. Figure 21 shows the part of DASH's Class Diagram responsible for this task.

We have included similar color coding as in the Configuration in Code Fragment 1 to assist in its comprehensibility. Classes with the same color signify close relations. At the very bottom of the Diagram is the connection to the rest of DASH with the `JSONLoader`. DASH will always put the requirements in the same file location, which is identical to the path in the `JSONLoader`, which is why the `JSONLoader` is static and does not need to be instantiated. In this process, Jackson is employed to parse the JSON. Jackson is a high-performance JSON processor for Java. It is specifically potent at creating Java Objects with the same hierarchy and types as the JSON file. This is the reasoning behind the structure of the `SystemConfiguration`. Additional logic was implemented to check for a change in the JSON file. Here, the checksum of the file is calculated and compared with the one found in the cache. If it is the same, the old `SystemConfiguration`, also cached, can be returned as no changes have occurred. Should there be any changes, or in the absence of a cached SystemConfiguration, Jackson tries to parse the JSON. It consists of two components, the `GlobalConfig` (highlighted in red in the Diagram) and the `StreamProperties` (highlighted in orange). The `GlobalConfig` includes the `DataSchema` (highlighted in teal) alongside the bootstrap server and topic identifier. The Kafka Server information together with the topic identifier uniquely specifies the underlying anonymized stream. Collectively, these elements uniquely identify the stream to be anonymized and delineate its attributes and (de-)serialization process. The `DataSchema` specifies the attribute type and attribute name pairs that make up the schema of a data stream. There are various data schema formats available and did not want to limit DASH to a single one. Instead, DASH abstracts data schemas into an internally used `SchemaCommon` simply containing the data fields as a HashMap. The available types are maintained in the `FieldType` Enum and include String, Int, Long, Float, Double, Boolean, and Optional values. This abstraction is achieved by implementing the `DataSchema` interface. It forces a new data schema to implement the conversion of its schema into a `SchemaCommon`. Further, it must provide the structure to be deserialized by Jackson. DASH already provides two implementations of the `DataSchema` interface, Avro (highlighted in blue) and Kafka Struct (highlighted in purple). This should already cover most of the use cases and should further serve as examples if in the future another data schema is to be added. The deserialization of both schemas is straightforward for Jackson with the provided helper classes breaking down their hierarchies. The only difficulty lies in the types available for Avro as these include single as well as multiple types for one attribute. An additional deserializer (highlighted in green) was implemented to accomplish this task for Jackson.

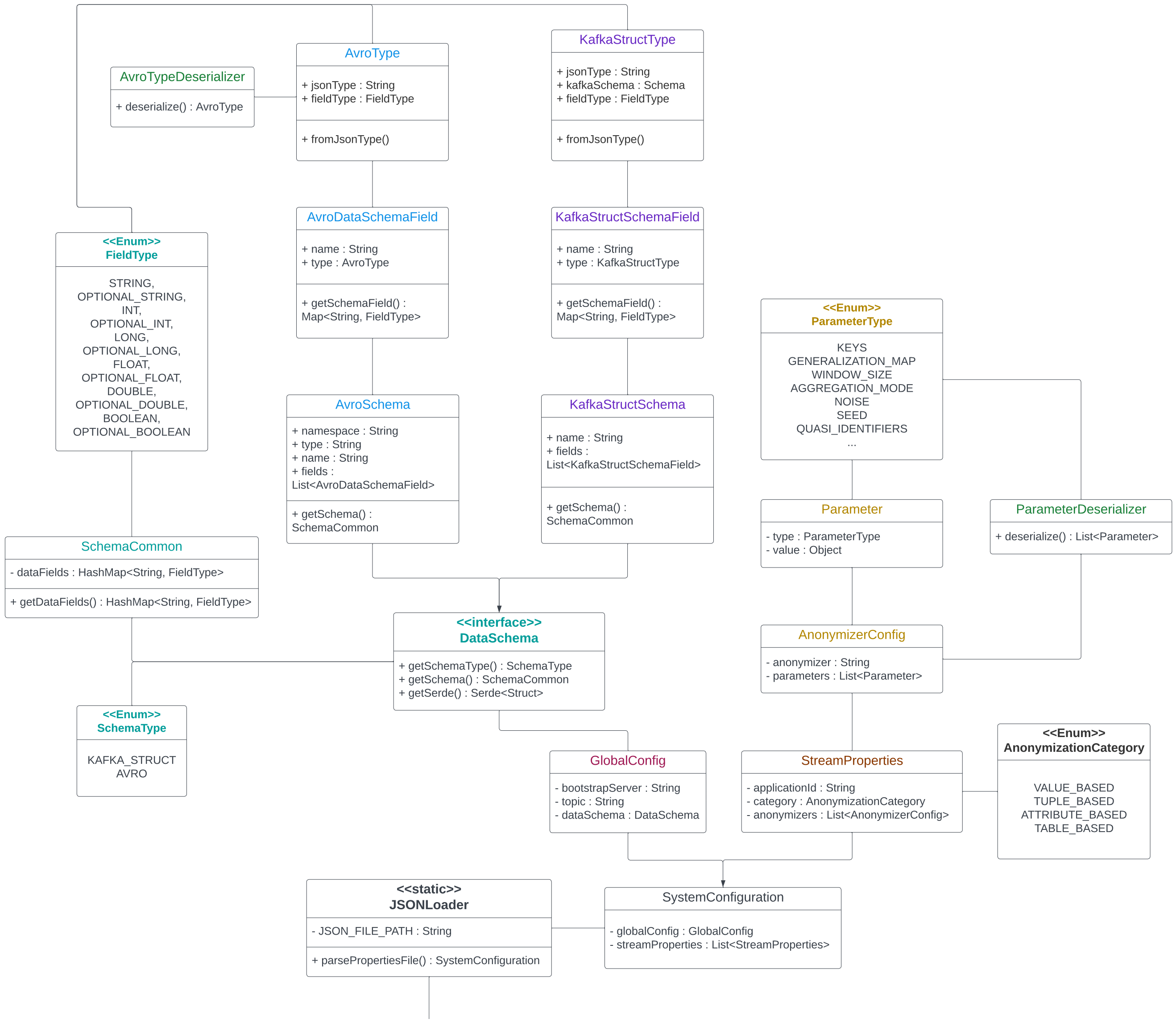


Figure 21: Class Diagram of the JSON Loader responsible for parser the Data Officer's requirements.

Next to the GlobalConfig are the StreamProperties. Each stream includes at least one anonymizer. An anonymizer (highlighted in yellow) is specified with a name and its parameters. All available parameters of all anonymizers are represented as a constant in the ParameterType enum. As one anonymizer can include many parameters of different types a special ParameterDeserializer was implemented. This deserializer processes each parameter in the list individually, converting it into an object as defined by the Parameter class corresponding to that ParameterType. These include simple types like double or int, but can simultaneously be more complex as hierarchies, enums, hashmaps, and lists. Again, DASH has been designed to facilitate change. Adding new Parameters is as simple as adding a new value to the ParameterType enum and implementing its deserialization in the ParameterDeserializer.

The deserialization relies heavily on the correct format of the requirements and thus provides extensive error messages to the user, the Data Officer, upon failures. Additionally, a recipe with examples as well as documentation - is included in the artifacts found in the thesis repository (https://github.com/TheRealHenri/master_thesis). Consequently, the JSONLoader accomplishes another crucial task alongside the processing of the configuration, it checks the syntax of the requirements JSON. The deserialization accepts only the expected types for all classes and parameters. A wrong input syntactically will be caught here. The semantics of the requirements, specifically of the parameters, are validated in another component of DASH responsible, which we will turn our attention to next.

4.3.3 Stream Config Validation and Construction

The Stream Config Validation and Construction aspect of DASH serves a fundamental role in preparing stream data for Kafka. It starts by taking StreamProperties and methodically converting them into a Kafka-compatible configuration. This step involves not just creating configurations but also ensuring that each anonymizer's parameters undergo a strict validation process before they are used. The accompanying Class Diagram, shown in Figure 22, outlines this essential process, with the functionality responsible for validation marked in teal and the functionality responsible for construction marked in red.

The central component is the AnonymizationStreamConfigBuilder, shown on the left in the figure. Algorithm 2 shows pseudocode for its one method - build(streamProperties).

The config builder first validates the input StreamProperties by checking whether an application ID for the stream is set and that it does not contain characters that Kafka does not allow i.e. spaces and other special characters. It

Algorithm 2: Pseudocode for Building Stream Configurations

```
Parameter: StreamProperties
Output   : Configuration for Kafka Streams
Validate StreamProperties
anonymizationCategory ← null
anonymizers ← Instantiate Anonymizers
foreach anonymizer do
    if anonymizationCategory == null then
        anonymizationCategory ← anonymizer's anonymization category
    else
        Validate anonymizationCategory
    end
    parameterExpectations ← anonymizer's parameter expectations
    foreach parameterExpectation do
        parameterValidators ← parameterExpectation's validators
        foreach validator do
            Validate Parameter
        end
    end
end
Initialize Anonymizers with their validated parameters
if anonymizationCategory == ATTRIBUTE_BASED then
    Validate WindowConfig
end
return new AnonymizationStreamConfig with Application ID,
  anonymizers list, and anonymizationCategory
```

also checks whether the specified anonymizers are available by verifying against the `AnonymizerRegistry`. The `AnonymizerRegistry` contains a map of all anonymizer names and corresponding anonymizer classes that are currently available in DASH.

Next, the appropriate anonymizer class is instantiated. This is necessary to access their individual `getParameterExpectations`. For every parameter that an anonymizer offers, a `getParameterExpectation` is specified. It contains the parameter name, the validation logic and whether is it required for this anonymizer's functionality. The validation logic is encapsulated in a list of validators implementing the `ParameterValidator` interface. There are currently five validators for all the parameters used in DASH, `KeyValidator`, `PositiveNumberValidator`, `QIKeysValidator`, `ConditionMapValidator` and an `EnumValidator`.

To facilitate the validation the `AnonymizationStreamConfigBuilder` is instantiated with the `SchemaCommon` containing the data fields of the data schema of the underlying stream. For example, every single anonymizer requires a 'keys' parameter, which specifies the scope of the anonymizer's anonymize function. All attributes for a tuple with a key within the 'keys' parameter will get anonymized. A 'keys' parameter, therefore, is valid if there is an attribute in the data stream for every key. After the anonymizers are instantiated and their parameters are validated, they are initialized with their parameters.

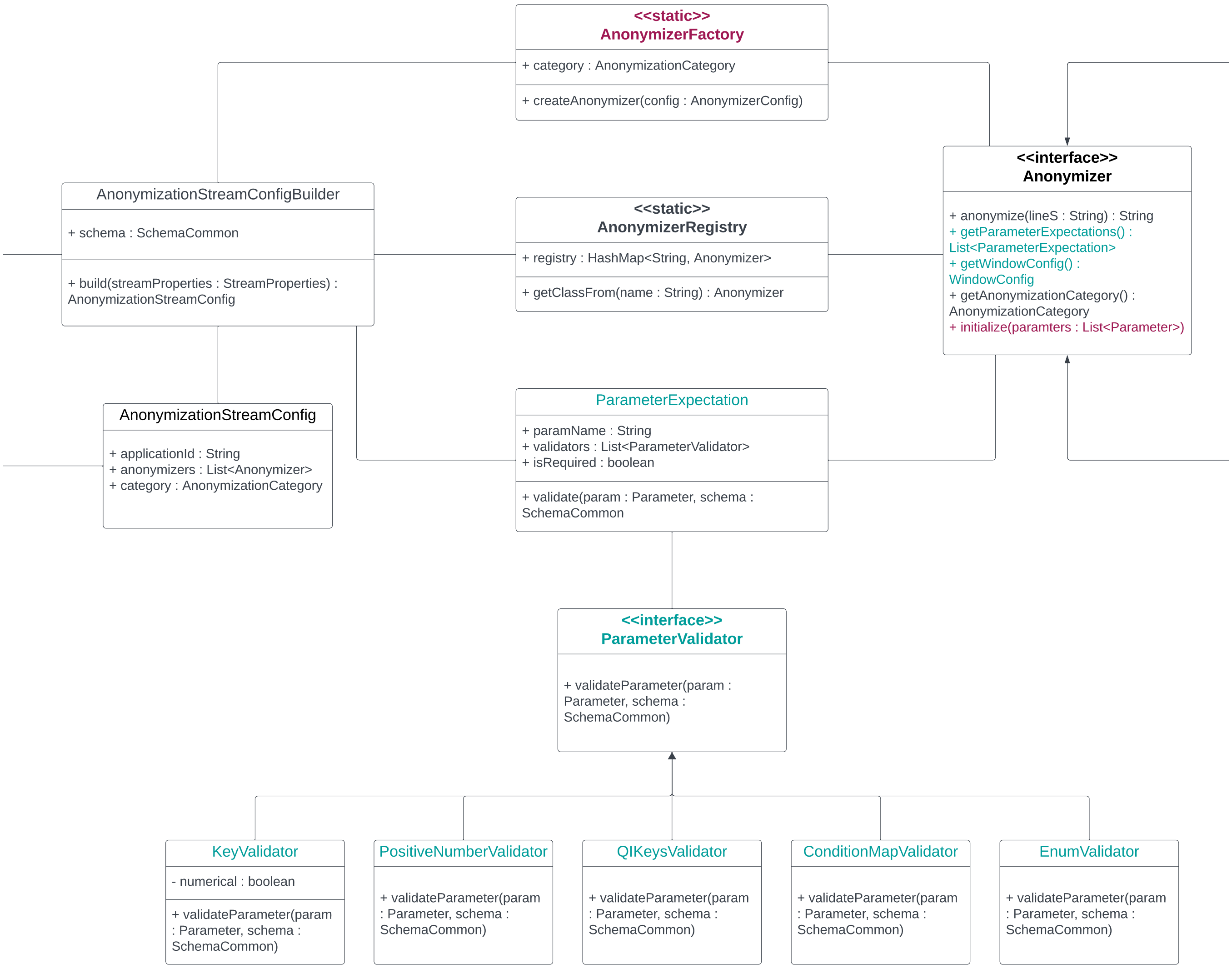

<<static>>
AnonymizerFactory
+ category : AnonymizationCategory
+ createAnonymizer(config : AnonymizerConfig)
<<interface>>
Anonymizer
+ anonymize(lineS : String) : String
+ getParameterExpectations() : List<ParameterExpectation>
+ getWindowConfig() : WindowConfig
+ getAnonymizationCategory() : AnonymizationCategory
+ initialize(paramters : List<Parameter>)
AnonymizationStreamConfigBuilder
+ schema : SchemaCommon
+ build(streamProperties : StreamProperties) : AnonymizationStreamConfig
<<static>>
AnonymizerRegistry
+ registry : HashMap<String, Anonymizer>
+ getClassFrom(name : String) : Anonymizer
AnonymizationStreamConfig
+ applicationId : String
+ anonymizers : List<Anonymizer>
+ category : AnonymizationCategory
ParameterExpectation
+ paramName : String
+ validators : List<ParameterValidator>
+ isRequired : boolean
+ validate(param : Parameter, schema : SchemaCommon
<<interface>>
ParameterValidator
+ validateParameter(param : Parameter, schema : SchemaCommon)
KeyValidator
- numerical : boolean
+ validateParameter(param : Parameter, schema : SchemaCommon)
PositiveNumberValidator
+ validateParameter(param : Parameter, schema : SchemaCommon)
QIKeysValidator
+ validateParameter(param : Parameter, schema : SchemaCommon)
ConditionMapValidator
+ validateParameter(param : Parameter, schema : SchemaCommon)
EnumValidator
+ validateParameter(param : Parameter, schema : SchemaCommon)

Figure 22: Class Diagram of the Stream Config Builder component and its corroborators responsible for validating (highlighted in teal) and instantiating (highlighted in red) individual anonymization stream configurations.

Following the initialization, for an anonymization stream with attribute-based anonymizers their windowing strategy is validated. As many anonymizers can be included in one stream, but only one windowing technique is applied per stream, it must be ensured that all anonymizers have the same window configuration.

Finally, a new `AnonymizationStreamConfig` is created and returned, which includes the stream's name, its anonymizers, and the anonymization category.

If any of these validations fail DASH will not proceed to running any streams, instead the setup phase will fail and an extensive log is output to the Data Officer to aid in fixing the setup mistakes.

4.3.4 Centralizing Stream Management

The `StreamManager` functions as the operational extension of the Data Officer within DASH. This entity centralizes all operations and handles both internal and external communications within DASH. Take a look at Figure 23 for the Class Diagram. In the diagram, all green highlighted functionalities are directly administrable by the Data Officer.

In the center is the `StreamManager`. The constraint notation beneath the class name in Figure 23 denotes the Singleton pattern, ensuring a single instance is instantiated and globally accessible within the system. The Data Officer invokes system actions through the `stream_manager.sh` shell script, located at the system's root directory. This opens up a command line, which internally creates (in true Kafkaesque fashion) a Kafka Producer. Commands issued during the session are produced to a designated 'commands' topic. The CommandConsumer within DASH subscribes to this topic and forwards the commands to the StreamManager. The commands encompass initialization, starting, pausing, and stopping of all or individual streams, and terminating DASH. Additionally, there is a help command, which is also invoked on wrong input, to aid the Data Officer in administering and a list command to monitor the running system. The `JSONLoader` as well as the `AnonymizationStreamConfigBuilder` and the `AnonymizationStreamConfig` are previously introduced components, serving specific functions in stream configuration and setup. The `AnonymizationStreamFactory` handles the creation and configuration of the anonymization streams on the Kafka side. The global config defines the source Kafka Stream. The tuples then are forwarded to the stream specified in the `AnonymizationStreamConfig`. Subsequently, tuples are anonymized according to their category, as determined by the configuration. The anonymization itself is executed by the function within the anonymizer, which is specified by the configuration. In the case of an attribute-based stream, the stream is first windowed according to the specification, aggregated, and then forwarded

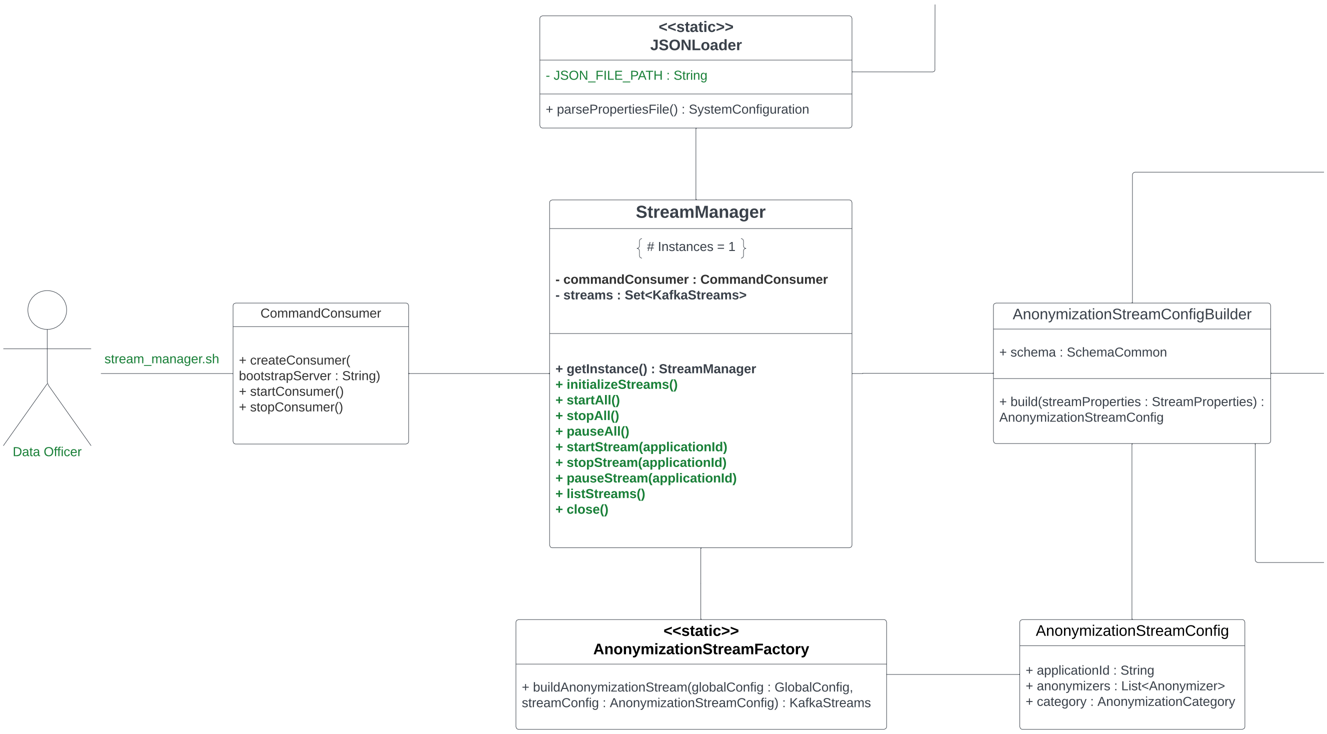

<<static>>
JSONLoader
- JSON_FILE_PATH : String
+ parsePropertiesFile() : SystemConfiguration
StreamManager
{ # Instances = 1 }
- commandConsumer : CommandConsumer
- streams : Set<KafkaStreams>
+ getInstance() : StreamManager
+ initializeStreams()
+ startAll()
+ stopAll()
+ pauseAll()
+ startStream(applicationId)
+ stopStream(applicationId)
+ pauseStream(applicationId)
+ listStreams()
+ close()
Data Officer
stream_manager.sh
CommandConsumer
+ createConsumer(
bootstrapServer : String)
+ startConsumer()
+ stopConsumer()
AnonymizationStreamConfigBuilder
+ schema : SchemaCommon
+ build(streamProperties : StreamProperties) :
AnonymizationStreamConfig
<<static>>
AnonymizationStreamFactory
+ buildAnonymizationStream(globalConfig : GlobalConfig,
streamConfig : AnonymizationStreamConfig) : KafkaStreams
AnonymizationStreamConfig
+ applicationId : String
+ anonymizers : List<Anonymizer>
+ category : AnonymizationCategory

Figure 23: The central component of DASH illustrated in a UML Class Diagram. All green highlighted functionalities are directly administered by the Data Officer.

as a collection of tuples to the anonymizers. All streams can contain multiple anonymizers, which process the data sequentially. This means that the output of the first anonymizer will be the input of the second anonymizer and so on.

Ultimately, the system publishes the anonymized tuples to a newly created topic within Kafka. Its naming convention is '{source_stream_name}-{application_id}'. From here they exist in the Kafka server and can be consumed from outside DASH. In combination with the aforementioned RBAC system, these newly produced topics can be restricted to be only consumable by users who are assigned to a certain role.

4.4 Data Pipeline

To facilitate the development and evaluation of DASH and RBAC mechanisms within Kafka, we implemented a data pipeline. It encompasses a data generator feeding data into a custom Kafka Connector. Within Kafka, the data is anoynmized and distributed into multiple role-restricted output data streams. These are read by Kafka Consumers. Figure 24 depicts the flow of data from the source to the destination as indicated by the blue arrows. Each component is associated with its respective detailed explanation in the indicated sections.

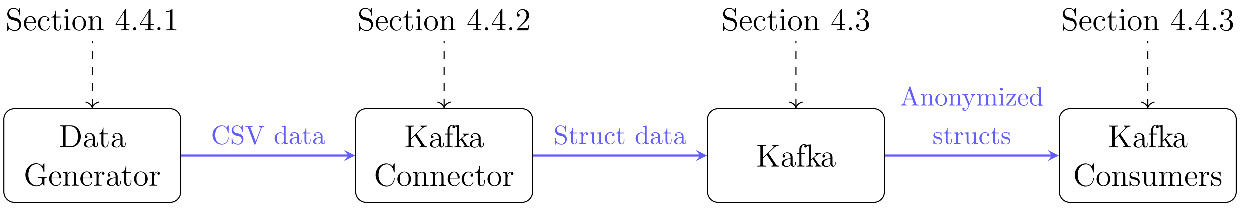


Figure 24: Data Pipeline of the Implementation: The diagram shows the various components and their data flow, indicated by blue arrows, with references to the corresponding sections.

In this section, we will take a closer look at the components of the data pipeline outside of Kafka. It follows the flow of data beginning with the generator.

4.4.1 Data Generator

The data generator, developed in Python and accessible via a Jupyter Notebook [28], is designed to simulate patient data for a German hospital specializing in diabetes care, as detailed in the Use Case Example (Section 3.1). Operating on a Python3 kernel, this generator utilizes modules such as Pandas [47], Numpy [14], and Faker [11] to create and manipulate CSV data.

The number of lines generated can be specified at the top of Data Generation section of the `dataGenerator.ipynb` notebook by adjusting the `nRows` value. A sample collection of data generated in this way with $nRows = 40$ is shown in the Appendix in Table 11. The script employs the Faker module for generating realistic yet fictitious personal data, such as names, addresses, and phone numbers. Additionally, a seed can be specified to easily reproduce the same data.

The data generator also ensures diversity in data by assigning gender, diagnosis, insurance companies, and other patient attributes based on realistic distributions and correlations. For instance, the genders Male and Female are assigned 48 percent of the time respectively for each entry, and 2 percent it assigned to Non-Binary. Similarly, it generates the diagnosis of Type 1 in nine percent of the cases, Type 2 in 89 percent, and Type 1.5 in the remaining two. An insurance company is randomly assigned to each entry according to their distribution in Germany.

Height, weight, age, glucose, and HbA1C are randomly assigned from within a provided reasonable range. The medication is correlated with the diagnosis.

The generator's design allows for straightforward adjustments of data randomness, structure, and volume, which can be configured in the initial cell of the notebook.

4.4.2 Kafka Connector

Since CSV files cannot be directly published to the Kafka network, it requires the use of a middleman, a Kafka Producer. This requirement is frequently encountered by Kafka users. This is where Kafka Connectors come into play. They alleviate this task of developing a Producer manually and transform the data for the user. There are many such Connectors available, but most are restricted by licenses that restrict their usage. Therefore, we developed our own Kafka Connector for CSV files.

Implementing a custom connector involves utilizing two key interfaces of the Kafka Connect framework: `SourceConnector` and `SourceTask`. The source connector class sets up the connection to Kafka. This includes setting the properties for the server, the security protocols, and the topic name that the connector should produce to. It also specifies the `SourceTask` class and how many instances of that class should run simultaneously.

The implemented task reads the CSV file located in a fixed location. Data is read into a buffer employing Java's `BufferedReader`. This buffer, containing a batch of lines, is then processed. Lines are extracted by detecting end-of-line characters '\n' and '\r'. The lines are converted into Kafka Structs with a predefined Schema that fits the data generated by the aforementioned Python script. Ultimately, the structs are produced to the specified topic in batches.

The connector is integrated with the Kafka server by compiling the Java code into a JAR file and transferring this file to Kafka's distribution data folder. Subsequently, it can be executed with the specified properties at runtime.

4.4.3 Kafka Consumers

In general, the terminal consumers included in the Apache Kafka distribution suffice for numerous implementations. It simply consumes the tuples from a specified topic. However, when implementing access control and simultaneously monitoring multiple data streams, the setup becomes laborious.

Consequently, a script was developed to instantiate consumers with predefined roles, including authentication, in a unified setup. It is set up with the specifications of the default DASH configuration and writes the consumed tuples to the terminal. Unlike default terminal consumers that output data as plain text, custom consumers recognize the data structure and display it in a more aesthetically pleasing format.

The implementation details are available in the 'consumers' folder within the Kafka pipeline directory.

4.5 Docker-based System Integration

The systems developed, including Kafka, ZooKeeper, the RBAC system, DASH, the data generator, the source connector, and consumers, are designed to operate in conjunction. This ensemble forms a complete data pipeline, simulating data streams being anonymized in real-time. However, the simultaneous management of seven or more systems can be a fault-prone and tedious process, requiring the handling of numerous terminals and configurations. This is why we have opted early in the development process to unify building, deploying, and running all systems in containers using Docker [25].

Docker facilitates the containerization of applications into standardized, executable components. This is achieved by allocating resources from the underlying hardware for use by the Docker engine. Atop this engine, within the container, developers can freely choose the operating system to build the application. The application within the container will then be runnable on any environment simplifying the deployment.

The integration of multiple containers is efficiently achieved through a 'Docker Compose' file. Each Docker container is specified as a service in the compose file. This is how our systems are unified and become executable in any environment with the simple command `Docker compose up` in the code folder of the repository (which can be found alongside all other artifacts, the thesis itself, and relevant literature in `https://github.com/TheRealHenri/master_thesis`).

The compose file is written in yaml and lists services e.g. containers followed by internal structures such as networks and Docker volumes. The latter can be used for persisting storage generated by and used by Docker containers. There are two types of volume storage: bind mounts point to the storage of the host system, Docker volumes on the other hand are completely managed by Docker. Each service represents a Docker container and consists of its specifications as shown in Code Fragment 2.

```
service_name:
    build: ./path/to/project
    # alternatively:
    image: 'dockerhub_link'
    networks:
        - network_name
    ports:
        - internal_port:exposed_port
    depends_on:
        - other_service_name
    environment:
        - variable_name: variable_value
    volumes:
        - ./storage/path/locally:/docker/path1
        - docker—volume:/docker/path2
    command: executed_on_startup

other_service_name:
    ...

networks:
    network_name:
        driver: driver_type

volumes:
    docker—volume:
```

Code Fragment 2: Docker compose structure

For each service within Docker, the specification of either a build or an image is mandatory. The build refers to a project with its own Dockerfile. Images may be selected from Docker Hub [26]. We use the `bitnami/zookeeper:3.8.2` image for zookeeper and the `bitnami/kafka:3.5.1` image for kafka. The rest of our systems were developed from scratch and included a Dockerfile with the necessary build information.

In the docker compose, we specify a network for all containers to use called simply 'kafka_network'. This allows containers within the network to communicate with each other by service name instead of IP address. In addition, communication necessitates exposed ports; without these, inter-container is not possible. Ports exposed by the application running inside a container need to be mapped to ports accessible from the outside. For instance, Kafka exposes the communication port

9092 to its clients and on port 8083 with its connectors. The communication between Kafka and Zookeeper occurs on port 2021. We have set up the data generator to run on a web socket on port 8888, so if someone using the system wants to change the way data is generated, the person can do it during runtime with a user interface in the browser at `localhost:8888`.

Dependencies among containers are also managed through the Docker Compose file. For instance RBAC, DASH, and the Kafka Connector are dependent on a running Kafka server. Kafka in turn is dependent on a running ZooKeeper instance. These dependencies can be specified by simply adding the name in the 'depends_on' section of the compose. Furthermore, since version 3 of compose, the status of the dependent container can be specified as well. We require the data pipeline build process to be finished successfully before starting the Kafka connector. This better allocates resources at runtime by not spending it on processes that would have to wait for others later on.

Environment variables correspond to configuration parameters within the system. Commonly, there is a set of parameters available for images from Dockerhub. In the case of the Kafka image, several parameters have to be set. For one the listeners dictate the protocols used for internal and external communication with the server. Additionally, the zookeeper communication is specified as well. In our docker compose we emulate a distributed event store. For this, we run the Kafka server on three different brokers. Each broker is assigned its container, they differentiate in their port mappings as well as, crucially, their broker ID environment variable. All clients and connectors specify all three brokers as their bootstrap server e.g. `bootstrapServer=kafka1:9092,kafka2:9093,kafka3:9094`.

We utilize Docker volumes for data storage. As intended by Docker we use bind mounts for data we specify on the host system like the Data Officer's anonymization requirements, Kafka Connector's configuration, and the RBAC database. For data shared between containers, we use specific Docker volumes managed by Docker. For example, the data generator produces a CSV file used by the Kafka Connector. Produced logs are also stored in Docker.

Finally, a command can be specified to be executed upon completion of the build process. Calling `docker compose up` starts up all containers and executes the specified command. The RBAC system opens a shell for the CLI, while the Kafka Connector is started up with a shell script.

Ultimately, Docker is configured to initially launch ZooKeeper, followed by the Kafka server with three operational brokers for load balancing. In the meantime, data is generated. Upon completion, the Kafka Connector, the RBAC system, and DASH. The generated data is fed through the Connector into DASH and produced to the anonymized streams. This entire process is executable through a single command in one terminal, provided Docker is installed on the host system.

5 Testing and Evaluation

In the thesis

5.1 Experimental Setup

... should include the following:

- define experimental data and workload(s),
- discussion about the selection and interpretation of the evaluation metrics,
- discussion about the computing environment, including hardware, software, and tools.

5.1.1 Experimental Design

5.2 Parameter Optimization

5.2.1 Interpretation of the Results

This sub-section should include

- description and an interpretation of the experimental results.
- explanation for any anomalies or any unexpected behavior.

6 Related Work

Organizations worldwide are increasingly committed to complying with data privacy and protection regulations. This trend is also evident among users of Apache Kafka, who are adapting their systems to comply with these international standards.

Out-of-the-box Apache Kafka supports authentication with SASL, SSL, and Kerberos as well as authorization with ACLs, but only when specifically enabled. As highlighted throughout this thesis the default configuration of Kafka contains no inherent data privacy measures. Kafka's stance is clear: they leave enforcing data privacy regulations to their users.

In addressing this need, users of Kafka, such as Amazon AWS, have expanded upon the platform's basic authentication and authorization capabilities. Amazon AWS, for example, integrates multi-factor authentification and RBAC on their clusters that run Kafka. Additionally, they encrypt all data flowing through Kafka [3]. Thus, Amazon understands the need for more privacy - but enacts it on its product instead of Kafka itself. This approach diverges from the core objective of this thesis, which is to embed such privacy features directly into distributed event stores.

There is one organization, however, that turns Apache Kafka into an enterprise data streaming platform. Founded by the creators of Apache Kafka, Confluent [22] is built with Apache Kafka at heart. Utilized by prominent companies such as BMW, Bosch, the Deutsche Bahn, and Domino's [23], Confluent brings a wide variety of additional features to Kafka. This includes a user-friendly graphical interface for configuring and monitoring Kafka, a cloud service, a wide array of custom Kafka Connectors, and what they call 'Enterprise-grade Security'. A closer look shows that that encompasses secret protection (at-rest encryption of passwords, tokens, and configuration files), structured audit logs, as well as RBAC. While providing robust security features, it does not address the need for anonymized access control within Kafka.

Confluent's Privitar Connector [12] showcases that Confluent too recognizes the need for more anonymization granularity within Kafka. The Privitar Connector provides basic anonymization techniques, i.e. Blurring and Suppression, for specified PII. Their presentation at the Kafka Summit London [24] shows great potential for further anonymization granularity with more advanced anonymization techniques. However, as of the writing of this thesis, the development seems to still be in progress as indicated by their Connector Hub. Furthermore, its availability will be under a costly subscription, which contrasts with the open-source and cost-effective nature of the solution proposed in this thesis.

In the context of Kafka, our work in granular anonymization and access control is a novelty as far as we know. By developing an open-source solution, we aim to foster access to privacy-enhancing tools without a paywall, potentially encouraging companies to invest more in user data privacy.

7 Conclusion

7.1 Future Work

... should include the following:

- problem restated and a brief summary of the methodology,
- student contributions (e.g., survey, open-source software, journal publication),
- a brief summary of the findings and results,
- limitations and generalizability of the findings and results.
- lessons learned,
- recommendations for future research.

Bibliography

[1] Amazon: Amazon kinesis (2023), https://aws.amazon.com/kinesis/, accessed: 2023-10-01

[2] Amazon: Amazon redshift (2023), https://aws.amazon.com/redshift, accessed: 2023-10-01

[3] Amazon: Data protection in amazon managed streaming for apache kafka (2023), https://docs.aws.amazon.com/msk/latest/developerguide/data-protection.html, accessed: 2023-11-01

[4] Apache Software Foundation: Apache kafka (2023), https://kafka.apache.org, accessed: 2023-10-01

[5] Beust, C.: Jcommander (2023), https://jcommander.org/, accessed: 2023-11-01

[6] Byun, J.W., Li, N.: Purpose based access control for privacy protection in relational database systems. The VLDB Journal 17, 603–619 (7 2008)

[7] Cao, J., Carminati, B., Ferrari, E., Tan, K.L.: Castle: A delay-constrained scheme for k-anonymizing data streams. pp. 1376–1378. IEEE (4 2008)

[8] Chaudhuri, S., Kaushik, R., Ramamurthy, R.: Database access control and privacy: Is there a common ground? In: CIDR. pp. 96–103. Citeseer (2011)

[9] Colombo, P., Ferrari, E.: Privacy aware access control for big data: A research roadmap. Big Data Research 2, 145–154 (12 2015)

[10] Commission, E.: General data protection regulation (gdpr) (2023), https://gdpr.eu/, accessed: 2023-06-01

[11] Community, F.: Faker (2023), https://faker.readthedocs.io/en/master/, accessed: 2023-11-01

[12] Confluent: Privitar kafka connector (2023), https://www.confluent.io/hub/privitar/privitar-kafka-connector/, accessed: 2023-06-01

[13] Coporation, O.: Oracle databases (2023), https://www.oracle.com/database/, accessed: 2023-10-01

[14] Developers, N.: Numpy (2023), https://numpy.org/, accessed: 2023-11-01

[15] Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs, pp. 287–290. Association for Computing Machinery, New York, NY, USA, 1 edn. (2022), https://doi.org/10.1145/3544585.3544600

[16] Federation, I.D.: Diabetes facts & figures (2023), https://idf.org/about-diabetes/diabetes-facts-figures/, accessed: 2023-10-01

[17] Ferraiolo, D.F., Kuhn, D.R.: Role-based access controls. In: Proceedings of the 15th National Computer Security Conference. pp. 554–563. National Computer Security Conference, Baltimore, Maryland, United States (October 1992), published October 13-16, 1992

[18] Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed nist standard for role-based access control. ACM Transactions on Information and System Security 4, 224–274 (8 2001)

[19] Foundation, A.S.: Apacha cassandra (2023), https://cassandra.apache.org/index.html, accessed: 2023-10-01

[20] Foundation, A.S.: Apache pulsar (2023), https://pulsar.apache.org/, accessed: 2023-10-01

[21] Hansen, S.L., Mukherjee, S.: A polynomial algorithm for optimal univariate microaggregation. IEEE Transactions on Knowledge and Data Engineering 15, 1043–1044 (7 2003)

[22] Inc, C.: Confluent (2023), https://www.confluent.io/, accessed: 2023-11-01

[23] Inc, C.: Confluent customers (2023), https://www.confluent.io/customers/, accessed: 2023-11-01

[24] Inc, C.: Privacy engineering for the world of kafka (2023), https://www.slideshare.net/ConfluentInc/privacy-engineering-for-the-world-of-kafka-alexander-cook-privitar-kafka-summi accessed: 2023-11-01

[25] Inc, D.: Docker (2023), https://www.docker.com/, accessed: 2023-11-01

[26] Inc, D.: Docker hub (2023), https://hub.docker.com/, accessed: 2023-11-01

[27] Inc, M.: Mongodb (2023), https://www.mongodb.com/, accessed: 2023-10-01

[28] Jupyter, P.: Jupyter notebook (2023), `https://jupyter.org/`, accessed: 2023-11-01

[29] of California Department of Justice, S.: California consumer privacy act (ccpa) (2023), `https://oag.ca.gov/privacy/ccpa`, accessed: 2023-06-01

[30] LeFevre, K., DeWitt, D.J., Ramakrishnan, R.: Incognito: Efficient full-domain k-anonymity. In: Proceedings of the 2005 ACM SIGMOD international conference on Management of data. pp. 49–60. ACM (2005)

[31] LeFevre, K., DeWitt, D.J., Ramakrishnan, R.: Mondrian multidimensional k-anonymity. In: 22nd International Conference on Data Engineering (ICDE'06). pp. 25–25. IEEE (2006)

[32] Li, N., Li, T., Venkatasubramanian, S.: T-closeness: Privacy beyond k-anonymity and l-diversity. In: 2007 IEEE 23rd International Conference on Data Engineering. pp. 106–115. IEEE (2007)

[33] LLC, G.: Cloud bigtable (2023), `https://cloud.google.com/bigtable`, accessed: 2023-10-01

[34] Ltd, R.T.: Rabbitmq (2023), `https://www.rabbitmq.com/`, accessed: 2023-10-01

[35] Machanavajjhala, A., Kifer, D., Gehrke, J., Venkitasubramaniam, M.: l-diversity: Privacy beyond k-anonymity. ACM Transactions on Knowledge Discovery from Data (TKDD) 1(1), 3 (2007)

[36] MySQL: Mysql (2023), `https://www.mysql.com/`, accessed: 2023-10-01

[37] Oganian, A., Domingo-Ferrer, J.: On the complexity of optimal microaggregation for statistical disclosure control. Statistical Journal of the United Nations Economic Commission for Europe 18(4), 345–353 (12 2001)

[38] Organization, W.H.: Diabetes fact sheet (2023), `ttps://www.who.int/news-room/fact-sheets/detail/diabetes`, accessed: 2023-10-01

[39] Otgonbayar, A., Pervez, Z., Dahal, K.: Toward anonymizing iot data streams via partitioning. In: 2016 IEEE 13th International Conference on Mobile Ad Hoc and Sensor Systems (MASS). pp. 331–336 (2016)

[40] Sandhu, R.S.: Role-based access control. In: Advances in computers, vol. 46, pp. 237–286. Elsevier (1998)

[41] Sandhu, R., Coyne, E., Feinstein, H., Youman, C.: Role-based access control: a multi-dimensional view. pp. 54–62. IEEE Comput. Soc. Press (1994)

[42] Sandhu, R., Samarati, P.: Access control: principle and practice. IEEE Communications Magazine 32, 40–48 (9 1994)

[43] Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Proceedings of the 1979 ACM SIGMOD international conference on Management of data. pp. 23–34 (1979)

[44] Statista: Ranking of the most popular database management systems worldwide, as of september 2023 (2023), `https://www.statista.com/statistics/809750/worldwide-popularity-ranking-database-management-systems/`, accessed: 2023-11-01

[45] Sweeney, L.: Weaving technology and policy together to maintain confidentiality. In: Journal of Law, Medicine & Ethics. pp. 184–195. Wiley Online Library (1997)

[46] Sweeney, L.: K-anonymity: A model for protecting privacy. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 10(05), 557–570 (2002)

[47] Team, P.D.: Pandas (2023), `https://pandas.pydata.org/`, accessed: 2023-11-01

[48] Team, S.: Sqlite (2023), `https://www.sqlite.org/`, accessed: 2023-11-01

[49] Wallwork, A.: English for writing research papers, chap. Abstracts, pp. 177–245. Springer International Publishing Switzerland (2011)

[50] Wallwork, A.: English for writing research papers, p. 306. Springer International Publishing Switzerland (2011)

[51] Wang, Q., Yu, T., Li, N., Lobo, J., Bertino, E., Irwin, K., Byun, J.W.: On the correctness criteria of fine-grained access control in relational databases. In: Proceedings of the 33rd international conference on Very large data bases. pp. 555–566 (2007)

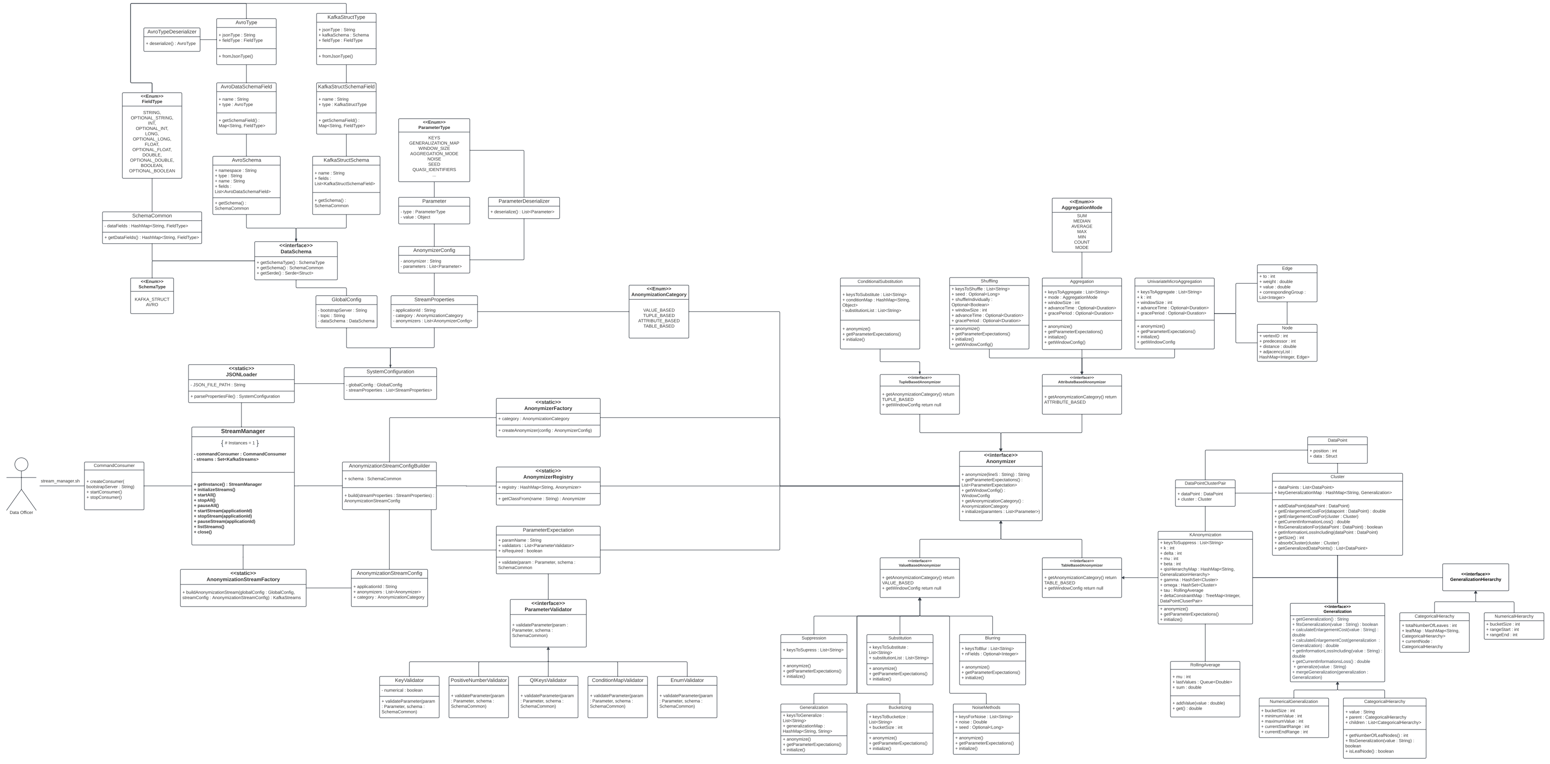
[52] Young, G.: Eventstoredb (2023), `https://www.eventstore.com/`, accessed: 2023-10-01

[53] Zhang, J., Yang, J., Zhang, J., Yuan, Y.: Kids:k-anonymization data stream base on sliding window. In: 2010 2nd International Conference on Future Computer and Communication. vol. 2, pp. V2–311–V2–316 (2010)

Appendix A. Further Details on the Solution Approach

Table 11: Extended table of patient data used. The table shows a part of the data used when testing the system. The attribute abbreviations correspond as follows: hgt - height; wgt - weight; ins._company - insurance_company; ins._number - insurance_number, diag. - diagnosis; gluc. - glucose.

id	name	address	zip	phone	gender	hgt	wgt	age	ins._company	ins._number	diag.	gluc.	hba1c	medication
1	Sarah Kreusel	Benthinring 6	34597	0711 98 81331	Female	169	62	95	108312586	L660487647	E11	65	9.36	Metformin
2	Ronny Kohl	Sauerplatz 42	93801	089 14407487	Male	180	73	68	103508742	R938242194	E11	157	7.77	Metformin
3	Lilly Geißler	Mühlestr. 565	66369	03525914827	Female	170	51	75	108928697	B659387784	E11	98	9.79	Metformin
4	Marian Bauer	Bruderweg 6/2	27618	034 56379318	Male	189	69	23	103306961	N988901370	E10	112	8.67	Insulin
5	Sandra Jähn	Södinggasse 82	87339	02437 40551	Female	166	64	61	101002659	R453561328	E10	382	8.88	Insulin
6	Steffen Henk	Junkenallee 03	11517	03939818024	Male	169	82	57	103508742	Z988986270	E11	96	5.76	Metformin
7	Dennis Dobes	Trübplatz 465	00839	08180041538	Male	178	94	86	109500490	T577049432	E11	437	7.8	Metformin
8	Arif Geisler	Bienstraße 792	34510	09659 95631	Diverse	169	61	68	109500490	Z210900364	E10	182	5.6	Insulin
9	Udo Bürger	Kunst Allee 3	44149	023 1977653	Male	182	92	61	109519176	J339204213	E11	179	7.7	Metformin
10	Max Mustermann	Wegstr. 42	69840	0309 049397	Male	175	80	40	109500398	E822232308	E11	120	7.0	Metformin
11	Erika Mustermann	Hauptstr. 5	52820	04839768544	Female	165	65	36	108817930	K759451642	E10	140	6.8	Insulin
12	Julia Schmidt	Bergweg 13	33888	0612347192	Female	168	60	28	109000051	L858268436	E11	150	7.5	Metformin
13	Tobias Müller	Seeallee 21	78151	07899 544559	Male	180	85	55	108928697	I998139981	E11	130	6.9	Metformin
14	Christina Klein	Talweg 8	37111	0603328566	Female	160	55	33	108888888	K926100157	E10	135	7.2	Insulin
15	Uwe Lehmann	Stadtweg 14	27417	04896 92775	Male	175	78	45	109500398	P619653870	E10	145	7.1	Insulin
16	Anna Becker	Waldstr. 3	10044	08645 546756	Female	170	68	31	108312586	D769681473	E11	155	7.6	Metformin
17	Frank Schubert	Blumenweg 7	19364	0087010395	Male	178	82	48	109500787	G425120298	E11	125	7.4	Metformin
18	Kathrin Neumann	Feldstr. 11	56066	0759017707	Female	162	58	29	108814099	L006897178	E10	138	7.3	Insulin
19	Lars Hoffmann	Wiesenweg 18	75261	06671 81305	Male	182	90	52	109000051	A510337443	E11	128	6.7	Metformin
20	Sabine Fuchs	Bachstr. 22	68656	06366 46753	Female	167	63	34	109500398	H758511496	E10	132	7.0	Insulin
21	Dirk Sommer	Forstweg 5	73277	09304 75332	Male	170	76	43	109500787	M562343754	E10	142	6.9	Insulin
22	Marie Lange	Hügelstr. 9	72773	0353876928	Female	165	61	38	108313123	D564564174	E11	160	7.8	Metformin
23	Oliver Krause	Grabenstr. 2	35141	01214338986	Male	180	84	50	101002659	U921658735	E11	118	6.8	Metformin
24	Susanne Winter	Brunnenweg 19	01870	05033 36428	Female	159	54	32	108918320	Q667879936	E10	137	7.1	Insulin
25	Markus Winkler	Kanalweg 4	90111	06309 336421	Male	176	77	46	103306961	X360906143	E10	148	7.0	Insulin
26	Stefanie Berger	Bahnhofstr. 31	83716	0371472504	Female	169	67	39	108811215	I955572981	E11	153	7.7	Metformin
27	Peter Klein	Rosenweg 12	30962	01556 031663	Male	179	83	51	109500398	H607086736	E11	121	6.6	Metformin
28	Claudia Schmitt	Kirchweg 29	57024	09251 652740	Female	164	59	30	101575519	T477237457	E10	136	7.2	Insulin
29	Heiko Schulz	Sonnenallee 23	67294	07686863295	Male	183	88	53	108928697	K281348102	E11	127	6.5	Metformin
30	Birgit Maier	Deichstr. 1	28024	01098 38370	Female	166	64	35	109500044	G059276367	E10	134	6.9	Insulin
31	Jan Fischer	Windweg 6	60379	04196 93544	Diverse	171	79	44	109519176	R260583528	E10	140	7.0	Insulin
32	Silke Wagner	Steinweg 10	09392	05613 34703	Female	172	70	37	108334056	O996297939	E11	158	7.9	Metformin
33	Daniel Meier	Schlossallee 15	23078	02210254705	Male	177	81	49	108817930	B037958300	E11	124	7.5	Metformin
34	Heike Schmidt	Lindenweg 20	85957	01958909813	Female	161	57	27	109500398	V237931864	E10	139	7.4	Insulin
35	Andreas Schneider	Parkstr. 16	67797	0118630356	Male	184	89	54	103306961	O573258576	E11	129	6.6	Metformin
36	Petra Fischer	Mühlenweg 17	92660	00420619289	Female	168	62	36	108918428	W571231267	E10	131	7.1	Insulin
37	Christian Weber	Querstr. 8	40430	0025856617	Male	172	75	42	108815718	M968302874	E10	143	6.8	Insulin
38	Laura Hoffmann	Auenweg 33	45953	0778320272	Female	173	69	40	108815718	T881197036	E11	157	7.6	Metformin
39	Stefan Bauer	Kreuzweg 3	24996	01188195858	Male	178	80	47	108815718	L541039098	E11	123	7.3	Metformin
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
n	Henri Allgöwer	Einsteinufer 17	10587	01765 123456	Male	178	68	27	101575519	T460187489	E10	453	10.13	Insulin



Data Officer
stream_manager.sh
CommandConsumer
+ createConsumer(bootstrapServer : String)
+ startConsumer()
+ stopConsumer()
StreamManager
{ # Instances = 1 }
- commandConsumer : CommandConsumer
- streams : Set<KafkaStreams>
+ getInstance() : StreamManager
+ initializeStreams()
+ startAll()
+ stopAll()
+ pauseAll()
+ startStream(applicationId)
+ stopStream(applicationId)
+ pauseStream(applicationId)
+ listStreams()
+ close()
<<static>>
JSONLoader
- JSON_FILE_PATH : String
+ parsePropertiesFile() : SystemConfiguration
<<static>>
AnonymizationStreamFactory
+ buildAnonymizationStream(globalConfig : GlobalConfig, streamConfig : AnonymizationStreamConfig) : KafkaStreams
SystemConfiguration
- globalConfig : GlobalConfig
- streamProperties : List<StreamProperties>
AvroTypeDeserializer
+ deserialize() : AvroType
AvroType
+ jsonType : String
+ fieldType : FieldType
+ fromJsonType()
KafkaStructType
+ jsonType : String
+ kafkaSchema : Schema
+ fieldType : FieldType
+ fromJsonType()
<<Enum>>
FieldType
STRING,
OPTIONAL_STRING,
INT,
OPTIONAL_INT,
LONG,
OPTIONAL_LONG,
FLOAT,
OPTIONAL_FLOAT,
DOUBLE,
OPTIONAL_DOUBLE,
BOOLEAN,
OPTIONAL_BOOLEAN
AvroDataSchemaField
+ name : String
+ type : AvroType
+ getSchemaField() : Map<String, FieldType>
KafkaStructSchemaField
+ name : String
+ type : KafkaStructType
+ getSchemaField() : Map<String, FieldType>
AvroSchema
+ namespace : String
+ type : String
+ name : String
+ fields : List<AvroDataSchemaField>
+ getSchema() : SchemaCommon
KafkaStructSchema
+ name : String
+ fields : List<KafkaStructSchemaField>
+ getSchema() : SchemaCommon
SchemaCommon
- dataFields : HashMap<String, FieldType>
+ getDataFields() : HashMap<String, FieldType>
<<interface>>
DataSchema
+ getSchemaType() : SchemaType
+ getSchema() : SchemaCommon
+ getSerde() : Serde<Struct>
<<Enum>>
SchemaType
KAFKA_STRUCT
AVRO
GlobalConfig
- bootstrapServer : String
- topic : String
- dataSchema : DataSchema
<<Enum>>
ParameterType
KEYS
GENERALIZATION_MAP
WINDOW_SIZE
AGGREGATION_MODE
NOISE
SEED
QUASI_IDENTIFIERS
...
Parameter
- type : ParameterType
- value : Object
ParameterDeserializer
+ deserialize() : List<Parameter>
AnonymizerConfig
- anonymizer : String
- parameters : List<Parameter>
StreamProperties
- applicationId : String
- category : AnonymizationCategory
- anonymizers : List<AnonymizerConfig>
<<Enum>>
AnonymizationCategory
VALUE_BASED
TUPLE_BASED
ATTRIBUTE_BASED
TABLE_BASED
<<static>>
AnonymizerFactory
+ category : AnonymizationCategory
+ createAnonymizer(config : AnonymizerConfig)
AnonymizationStreamConfigBuilder
+ schema : SchemaCommon
+ build(streamProperties : StreamProperties) : AnonymizationStreamConfig
<<static>>
AnonymizerRegistry
+ registry : HashMap<String, Anonymizer>
+ getClassFrom(name : String) : Anonymizer
AnonymizationStreamConfig
+ applicationId : String
+ anonymizers : List<Anonymizer>
+ category : AnonymizationCategory
ParameterExpectation
+ paramName : String
+ validators : List<ParameterValidator>
+ isRequired : boolean
+ validate(param : Parameter, schema : SchemaCommon
<<interface>>
ParameterValidator
+ validateParameter(param : Parameter, schema : SchemaCommon)
KeyValidator
- numerical : boolean
+ validateParameter(param : Parameter, schema : SchemaCommon)
PositiveNumberValidator
+ validateParameter(param : Parameter, schema : SchemaCommon)
QIKeysValidator
+ validateParameter(param : Parameter, schema : SchemaCommon)
ConditionMapValidator
+ validateParameter(param : Parameter, schema : SchemaCommon)
EnumValidator
+ validateParameter(param : Parameter, schema : SchemaCommon)
<<Enum>>
AggregationMode
SUM
MEDIAN
AVERAGE
MAX
MIN
COUNT
MODE
ConditionalSubstitution
+ keysToSubstitute : List<String>
+ conditionMap : HashMap<String, Object>
- substitutionList : List<String>
+ anonymize()
+ getParameterExpectations()
+ initialize()
Shuffling
+ keysToShuffle : List<String>
+ seed : Optional<Long>
+ shuffleIndividually : Optional<Boolean>
+ windowSize : int
+ advanceTime : Optional<Duration>
+ gracePeriod : Optional<Duration>
+ anonymize()
+ getParameterExpectations()
+ initialize()
+ getWindowConfig()
Aggregation
+ keysToAggregate : List<String>
+ mode : AggregationMode
+ windowSize : int
+ advanceTime : Optional<Duration>
+ gracePeriod : Optional<Duration>
+ anonymize()
+ getParameterExpectations()
+ initialize()
+ getWindowConfig()
UnivariateMicroAggregation
+ keysToAggregate : List<String>
+ k : int
+ windowSize : int
+ advanceTime : Optional<Duration>
+ gracePeriod : Optional<Duration>
+ anonymize()
+ getParameterExpectations()
+ initialize()
+ getWindowConfig
Edge
+ to : int
+ weight : double
+ value : double
+ correspondingGroup : List<Integer>
Node
+ vertexID : int
+ predecessor : int
+ distance : double
+ adjacencyList : HashMap<Integer, Edge>
<<interface>>
TupleBasedAnonymizer
+ getAnonymizationCategory() return TUPLE_BASED
+ getWindowConfig return null
<<interface>>
AttributeBasedAnonymizer
+ getAnonymizationCategory() return ATTRIBUTE_BASED
<<interface>>
Anonymizer
+ anonymize(lineS : String) : String
+ getParameterExpectations() : List<ParameterExpectation>
+ getWindowConfig() : WindowConfig
+ getAnonymizationCategory() : AnonymizationCategory
+ initialize(paramters : List<Parameter>)
<<interface>>
ValueBasedAnonymizer
+ getAnonymizationCategory() return VALUE_BASED
+ getWindowConfig return null
<<interface>>
TableBasedAnonymizer
+ getAnonymizationCategory() return TABLE_BASED
+ getWindowConfig return null
Suppression
+ keysToSupress : List<String>
+ anonymize()
+ getParameterExpectations()
+ initialize()
Substitution
+ keysToSubstitute : List<String>
+ substitutionList : List<String>
+ anonymize()
+ getParameterExpectations()
+ initialize()
Blurring
+ keysToBlur : List<String>
+ nFields : Optional<Integer>
+ anonymize()
+ getParameterExpectations()
+ initialize()
Generalization
+ keysToGeneralize : List<String>
+ generalizationMap : HashMap<String, String>
+ anonymize()
+ getParameterExpectations()
+ initialize()
Bucketizing
+ keysToBucketize : List<String>
+ bucketSize : int
+ anonymize()
+ getParameterExpectations()
+ initialize()
NoiseMethods
+ keysForNoise : List<String>
+ noise : Double
+ seed : Optional<Long>
+ anonymize()
+ getParameterExpectations()
+ initialize()
DataPoint
+ position : int
+ data : Struct
DataPointClusterPair
+ dataPoint : DataPoint
+ cluster : Cluster
Cluster
+ dataPoints : List<DataPoint>
+ keyGeneralizationMap : HashMap<String, Generalization>
+ addDataPoint(dataPoint : DataPoint)
+ getEnlargementCostFor(datapoint : DataPoint) : double
+ getEnlargementCostFor(cluster : Cluster)
+ getCurrentInformationLoss() : double
+ fitsGeneralizationFor(dataPoint : DataPoint) : boolean
+ getInformationLossIncluding(dataPoint : DataPoint)
+ getSize() : int
+ absorbCluster(cluster : Cluster)
+ getGeneralizedDataPoints() : List<DataPoint>
KAnonymization
+ keysToSuppress : List<String>
+ k : int
+ delta : int
+ mu : int
+ beta : int
+ qisHierarchyMap : HashMap<String, GeneralizationHierarchy>
+ gamma : HashSet<Cluster>
+ omega : HashSet<Cluster>
+ tau : RollingAverage
+ deltaConstraintMap : TreeMap<Integer, DataPointCluserPair>
+ anonymize()
+ getParameterExpectations()
+ initialize()
<<interface>>
GeneralizationHierarchy
<<interface>>
Generalization
+ getGeneralization() : String
+ fitsGeneralization(value : String) : boolean
+ calculateEnlargementCost(value : String) : double
+ calculateEnlargementCost(generalization : Generalization) : double
+ getInformationLossIncluding(value : String) : double
+ getCurrentInformationsLoss() : double
+ generalize(value : String)
+ mergeGeneralization(generalization : Generalization)
CategoricalHierachy
+ totalNumberOfLeaves : int
+ leafMap : MashMap<String, CategoricalHierarchy>
+ currentNode : CategoricalHierarchy
NumericalHierarchy
+ bucketSize : int
+ rangeStart : int
+ rangeEnd : int
RollingAverage
+ mu : int
+ lastValues : Queue<Double>
+ sum : double
+ addValue(value : double)
+ get() : double
NumericalGeneralization
+ bucketSize : int
+ minimumValue : int
+ maximumValue : int
+ currentStartRange : int
+ currentEndRange : int
CategoricalHierarchy
+ value : String
+ parent : CategoricalHierarchy
+ children : List<CategoricalHierarchy>
+ getNumberOfLeafNodes() : int
+ fitsGeneralization(value : String) : boolean
+ isLeafNode() : boolean

Figure 25: Full UML Class Diagram of DASH.

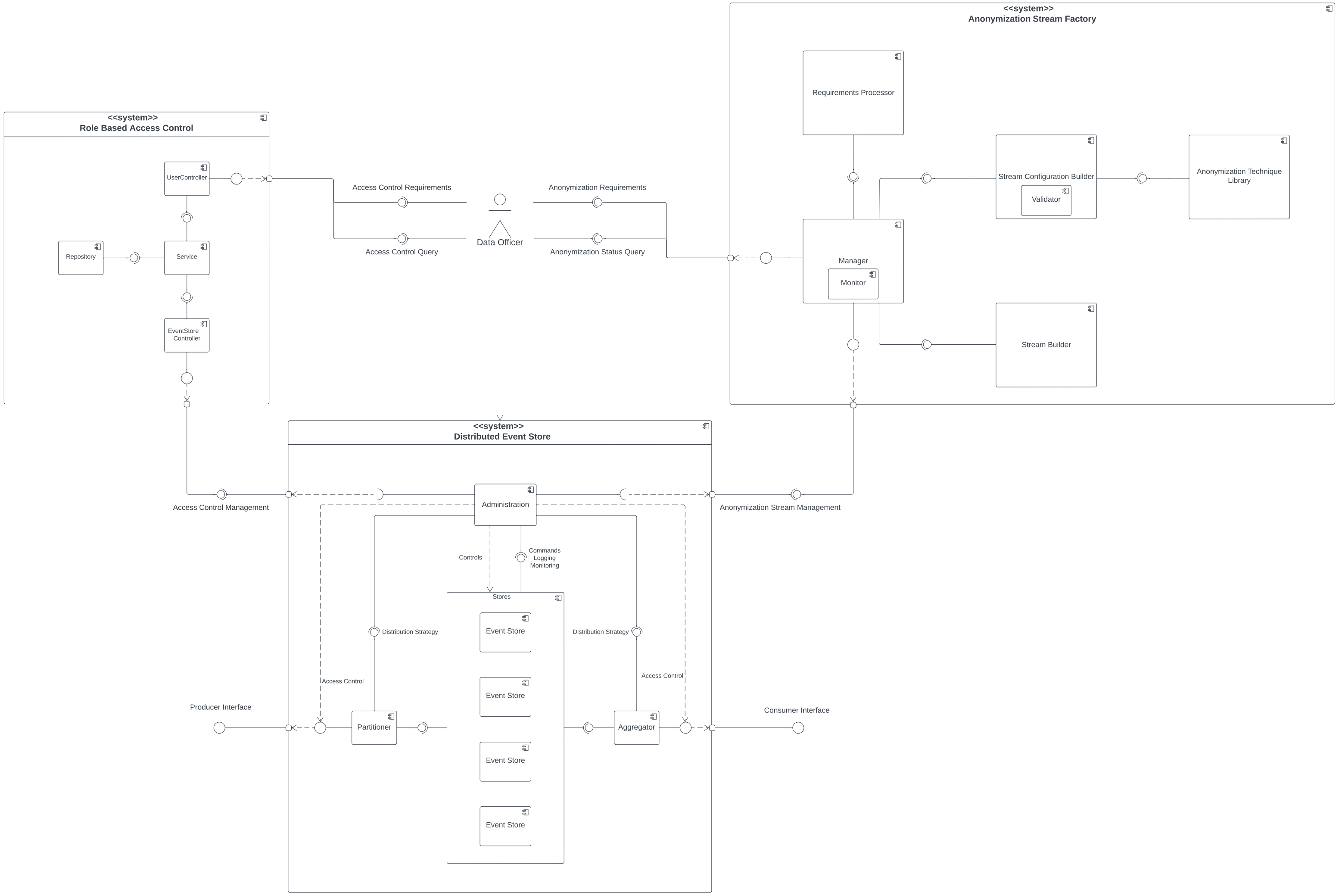


Figure 26: Full UML Component Diagram for the system as a whole.

Appendix B. Extended Version of the Experimental Results