Project Report: GDPR Compliant Key-Value Stores

Archita Agarwal
aa12

Marilyn George
mgeorge5

Abstract

We design a GDPR compliant key-value store for social network applications. We focus on the Right of Access by the Data Subject. The Right of Access is particularly interesting in the social network setting because of the collaborative nature of data generation in a social network. We propose a Owner-Viewer paradigm to resolve dependencies when accessing the data. Our solution was implemented with Redis and tested with a synthetic dataset for upto 10000 users. Data access for a user with very high activity was completed in less than a minute, while also supporting efficient functionality. We believe this design is promising and could scale to realistic social networks.

1 Introduction

The European General Data Protection Regulation (GDPR) [6] was implemented in May 2018 [8]. It specified that any entity operating in the EU or handling data of EU citizens must enforce certain data protection policies or be subject to penalties. Under these policies, a citizen is guaranteed certain rights such as the Right of Data Access, the Right to Transparency, the Right to Object and the Right to be Forgotten. After the GDPR went into effect, there have been several cases of organizations (and even individuals) being held to account for violating these rights [2]. The penalties for violating the GDPR can be as high as 4% of the annual turnover of an enterprise [8]. Due to this high possible cost, international organizations have been scrambling to become GDPR compliant over the past few years.

However, several of the systems that are in use today were not designed to handle the requirements imposed by the GDPR. As such, becoming GDPR-compliant often necessitates large changes to existing systems and workflows which potentially introduce inefficiency. Given the engineering costs of retrofitting existing systems to support entirely new functionality that the systems were not originally designed for – it is worth considering the alternative, compliance by construction [11]. We should explore the possibilities of building new systems that incorporate GDPR compliant design while also supporting the functionality of the original system efficiently.

In this paper, we design a key-value store to back a social network application. Social networks are used by hundreds of millions of people worldwide, and data is being processed at the rate of petabytes a day for Facebook alone [1]. The data generated is highly collaborative in nature – multiple users interact on posts by adding comments and likes. In this setting the right of access by the data subject (Art. 15) [6] is interesting because there are questions of data ownership before a user can access his/her data. All the data generated by a user belongs to them, but some data might be meaningless without contributions from other users.

We propose an Owner-Viewer paradigm to resolve ownership of data and use a sharded design [11] for the key-value store. The data we store to back a social media application will be along the lines of Facebook's TAO [10]. For our paper we consider the following (limited) types of data and functionality: user data (profile, friend lists, posts) and post data (comments, likes). We generate synthetic datasets with upto 10000 users and test the performance of the data access for a user. We note here that the design looks promising in our small-scale implementation, the data access for a very active user takes less than a minute to complete. However, the data for 10000 users can be stored on one machine – there might be several challenges to scale up to hundreds of millions of users and petabytes of data.

2 Related Work

GDPR compliance requirements have led to substantial work in terms of the effects on modern-day systems [12, 13]. The influence of the GDPR requirements on storage systems in particular has been studied previously by Shah et al [12]. The authors study Redis, a popular key-value store, and attempt to use inbuilt functions to implement GDPR compliant Redis. They show that their approach leads to inefficient operations on Redis and conclude that retrofitting existing storage systems naively to be GDPR compliant might come at a large cost.

In this paper, we explore GDPR compliance by construction for key-value stores. This design paradigm was introduced by Schwarzkopf et al. [11] along with an illustrative design for relational databases. They introduce the concept of *sharding* the data store into user-specific instances – either physically or logically. This implies that a user's data is always stored in one *shard* and is therefore easy to access, delete and manipulate in order to ensure GDPR compliance with respect to that user. The paper also discusses how to handle *materialized views* that need to access the data of multiple users. We use these principles to guide our design as described in the following section.

3 Design

We design a key-value store to be used as the data store for a social network application. We offer APIs for the social network application to interact with our key-value store. The APIs implement (limited) functionality such as the creation of users, friendships, posts, comments and likes. We also allow the social network application to invoke the right to access for any user and receive all of the users associated data in return.

Our key-value store incorporates sharded design [11]. Sharding enables us to break up the data store into logical units that can be accessed efficiently. We use both user-level and post-level sharding in our design. Each user has a user-specific logical shard of the key-value store that stores all the keys to access the data they generated. This enables us to support efficient data access when a user wishes to invoke their right of access. However, it is not sufficient to merely support the right to access efficiently. We must ensure that we continue to support the functionality of the application efficiently. If a user wants to use the view post functionality (accessing collaborative data generated by multiple users) our design would be inefficient if we only supported user-level sharding. For instance, a popular post could have several hundred comments from different users. The social network application would have to access hundreds of user shards to render the post for any one of those users. In order to support viewing a post efficiently we create post-specific logical shards. Each post shard now contains all the data required to render a post. This makes post rendering very efficient. However, this design choice introduces an overhead on the data access – a user's data access must now access each post they own separately. We consider this an acceptable trade-off under the assumption that post rendering will be more common than the invocation of the right of access. Additionally, the text of Art. 15 [11] does not specify a timeframe for the execution of the right of access however we aim to allow data access in a reasonable timeframe.

Once we have the sharded key-value store, we can support the right of access by a user. However, we still have to decide what keys are 'associated' to a user and what a user can rightfully claim ownership over. It is clear that a user's profile data and their post data belong to them. The case for shared data is slightly less clear. To resolve this we propose a Owner-Viewer paradigm for the shared data generated by a social network. We say that for each piece of shared data created, the users creating it can play the following roles:

- Owner-Owner: All users own the shared data for this association. For instance, a friendship or group memberships. When the right to access is invoked by either of them they should receive the data. This is similar to TAO's bidirectional edges.

- Owner-Viewer: An Owner owns the data, for instance a post. All Viewers can interact with this data, like or comment on it. When the Owner invokes the right to access it receives the data as well as all the interactions associated with the data. When the Viewer invokes the right to access it receives only the information associated with the interaction, but not the data itself - this belongs to the Owner. This is similar to the unidirectional edges in TAO.

We ignore Viewer-Viewer associations currently as being out-of-scope of the right to access. Then our right of access functionality resolves data ownership using these rules and returns all data associated with a user in this paradigm.

4 Implementation

We implemented the design using Redis [5] and the Python3 redis-py client [4]. The implementation is single server using a single instance of the Redis server running on localhost. At the time of writing this paper, the Github repository [3] contains 1000LOC for the APIs and experiments. We encountered one major challenge while using Redis as the key-value store. Redis only stores values as strings so we had to serialize our dictionary data-structures or flatten them for easy access. We made the choice to flatten the fields that would be accessed repeatedly but keep the remaining fields in dictionaries. Our design had to be modified to account for this. The other major challenge that we faced was during testing: the generation of the synthetic dataset and testing for correctness. We provide a detailed explanation in the next section.

5 Evaluation

In order to evaluate our design we generated synthetic datasets for 10, 100, 1000, 10000 users. The synthetic data was generated to model real-world social network patterns. We generated static key-value stores for each setting and evaluated the efficiency of the data access for high-activity users (worst case) and average users. We also tested the efficiency of rendering[1] posts of various popularity levels. For simplicity, we did not interleave data accesses with other operations on the key-value store. The data generation posed some interesting questions as detailed below.

Data Generation. We generate the synthetic datasets using a Zipf distribution. This distribution is a power law which explains several types of data on the Internet in practice [9]. In our context, this means that the frequency of a data item in our dataset is inversely proportional to its length. This data item could be a user's friend list, number of posts (or) the number of comments or likes on a post. Additionally, in the real world there exist super-users who are very active on the social network. The super-users have large numbers of friends, posts, comments and likes. Similarly, there exist super-posts that are very popular and get large numbers of comments and likes. To achieve this distribution in our synthetic dataset we used the following methods:

1. For a given number of users, we sampled a list of friend counts from a Zipf distribution. However, these friend counts had to be *consistent* i.e. realizable in the real world. For example, for three users - (5, 3, 1) is not a consistent set of friend counts. In order to check consistency, we view the social network as a graph with a vertex for each node and an edge for each friendship. Then if the friend counts are a valid degree sequence for the graph, they are also consistent. We used the Erdos-Gallai theorem [7] to check this consistency condition. The time required to generate a consistent set of friend counts was fairly large and hence we used static friend counts once generated. We ordered these counts in order to keep track of the users with largest numbers of friends for later steps. These users are then our super-users.

2. We generate posts according to a Zipf distribution for each user, making sure the super-users have the most posts. On each post we sample a Zipf distribution for the number of comments, nested[2] comments and likes. When we select a user to own a comment or like, we weight by the number of friends a user has. This leads to the creation of super-users – if they received more friends in the generation they will be more likely to comment and like posts.

3. We note here that the Zipf distribution has a parameter that controls how skewed the distribution is – larger values of the parameter generate more skewed distributions i.e. less number of longer data items. We adjust this parameter to enable us to generate data sets fast, but still attempt to keep the dataset statistics as realistic as possible. In Figure 5 we report the size of each dataset and the total time taken (in seconds) to generate the contents starting from pre-calculated friend counts. We denote our datasets as D10, D100, D1000 and D10000 denoting the number of users.

[1]Accessing and rendering in our context only involves plain printing of the relevant data. A social network application might have more complicated functionality and hence higher latency. Our numbers should be considered indicative of the time to retrieve relevant data from the key-value store.

[2]We allow up to three levels of comment nesting.

Dataset	Total time(s)	Users	Friendships	Posts	Comments	Likes
D10	0.113	10	11	12	135	222
D100	1.89	100	398	179	2250	3851
D1000	15.31	1000	4724	2098	16993	25494
D10000	182.46	10000	51070	18501	134141	192717

Figure 1: Data generation statistics

Performance. In this section we report the performance of the data accesses for users as well as the post rendering. We consider the first 10 user ids to be super-users. Their posts are considered the super-posts[3]. In Figure 5 we show the average running time of the right to access for a super-user compared to the average running time over all users. Similarly,

Dataset	Total superusers	Avg for SU(s)	Avg for all(s)
D10	10	0.013	0.013
D100	10	0.102	0.016
D1000	10	0.402	0.011
D10000	10	2.6	0.01

Figure 2: Time taken for right of access

in Figure 5 we also show the average render time for a super-post compared to the average render time over all posts. We

Dataset	Posts by superusers	Avg for SU(s)	Posts by all	Avg for all(s)
D10	12	0.0027	12	0.0027
D100	88	0.0038	197	0.0021
D1000	301	0.0053	1892	0.0015
D10000	2410	0.0051	20010	0.0014

Figure 3: Time taken for post rendering

notice that the average access time over all users remains more or less constant, or even decreases as total number of users increases. However, the average over the super-users increases as we increase the number of users. This is due to the effect of the Zipf distribution. The top 10 users' activity corresponds to the 10 largest numbers in the Zipf sample. Hence the top 10 activity and therefore access time increases as the number of Zipf samples increases. The posts by the super-users have a similar property and so we can see the same trend in the post render time. Overall, we see that the

[3]The fact that the super-users have lower user ids is an artifact of our data generation.

post rendering remains under half a second for all users and the data access is a couple of seconds at most even with 10000 users.

Conclusions. We have showed that the prototype sharded design works with reasonable performance. The design is worth studying for further extension, both in application functionality as well as GDPR compliant functionality. Due to time constraints we could not implement groups of users for the social network but we believe that will be a simple extension. Another possible extension will be the implementation of user-specific policies when viewing, commenting and liking a post. Our current datastore makes no distinction and pushes the responsibility for privacy policy enforcement to the social network application. On the GDPR compliance front we could add encryption for data at rest to comply with the security of data processing (Art. 32). We could also incorporate the right to data portability (Art. 20) and the right to erasure (Art. 17) [6] by natural extensions of our current design. Additionally we could work on scaling up the application to millions of users by distributing data over several Redis instances. This will give rise to additional questions about the efficiency of data access and social network functionality.

References

[1] Brandwatch: Facebook. https://www.brandwatch.com/blog/facebook-statistics/. Accessed: 2019-12-09.

[2] Gdpr enforcement tracker. http://www.enforcementtracker.com/. Accessed: 2019-08-18.

[3] Github: Gdpr-kv. https://github.com/architaagarwal/gdpr_kv. Accessed: 2019-12-09.

[4] Redis-py for python3. https://pypi.org/project/redis/. Accessed: 2019-12-09.

[5] Redis: Remote dictionary server. https://redis.io/. Accessed: 2019-12-09.

[6] Regulation (eu) 2016/679 of the european parliament and of the council (general data protection regulation). https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679#d1e3722-1-1. Accessed: 2019-10-10.

[7] Wikipedia: Erdos-gallai. https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93Gallai_theorem. Accessed: 2019-12-09.

[8] Wikipedia: General data protection regulation. https://en.wikipedia.org/wiki/General_Data_Protection_Regulation. Accessed: 2019-12-09.

[9] Lada A Adamic and Bernardo A Huberman. Zipf's law and the internet. *Glottometrics*, 3(1):143–150.

[10] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. {TAO}: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 49–60, 2013.

[11] Malte Schwarzkopf, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. Position: Gdpr compliance by construction. In *To be part of Poly'19: Towards Polystores that manage multiple Databases, Privacy, Security andor Policy Issues for Heterogenous Data*, 2019.

[12] Aashaka Shah, Vinay Banakar, Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. Analyzing the impact of {GDPR} on storage systems. In *11th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[13] Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. Gdpr anti-patterns: How design and operation of modern cloud-scale systems conflict with gdpr, 2019.