Extending Redis to Support GDPR

Motivation

As technology has advanced to store and manage big data, consumers and governments have become increasingly concerned with regulations on the privacy, sharing, and ownership of data. In an effort to give individuals control over their personal data, the European Union (EU) adopted the General Data Protection Regulation (GDPR). Besides providing rights and protections to the European people concerning their data, it also assigns responsibilities to companies holding it.

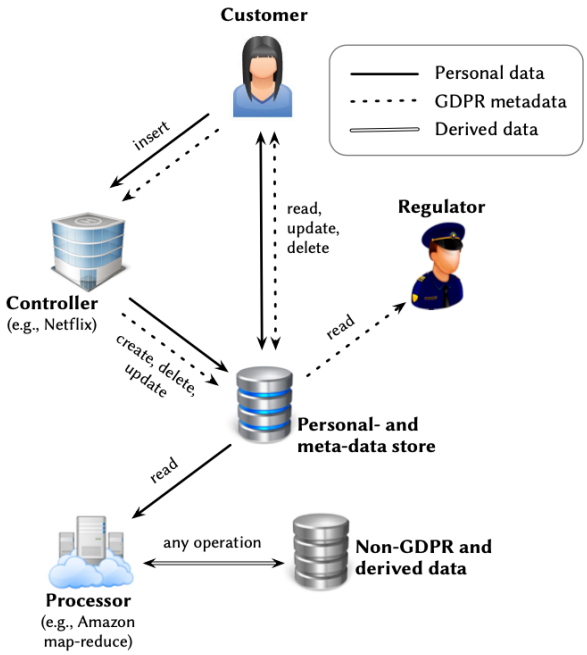
However, from a systems perspective, translating GDPR articles to capabilities of compliant systems can be a painstaking effort. Moreover, building a system that incorporates all GDPR requirements poses challenges in terms of performance, and storage costs. Therefore, extending a storage engine to support all (or at least most of) GDPR requirements while ensuring it is performant in terms of latency, throughput, etc is something all systems designers should consider.

The vagueness of the language in GDPR poses issues for both companies and users; companies may not understand what is necessary and users may not understand their rights. Further, compliance costs are high for businesses. They include system updates, increased storage, and beyond infrastructure changes, failure to comply could result in hefty fines (up to €20M or 4% of global revenue whichever is higher). Without any aid in transitioning toward compliance, smaller companies will be likely to fail quicker than larger ones with more resources at their disposal. Something to note is that of the $63 million in fines issued between the start of GDPR and June 2020, $57 million were issued to Google.

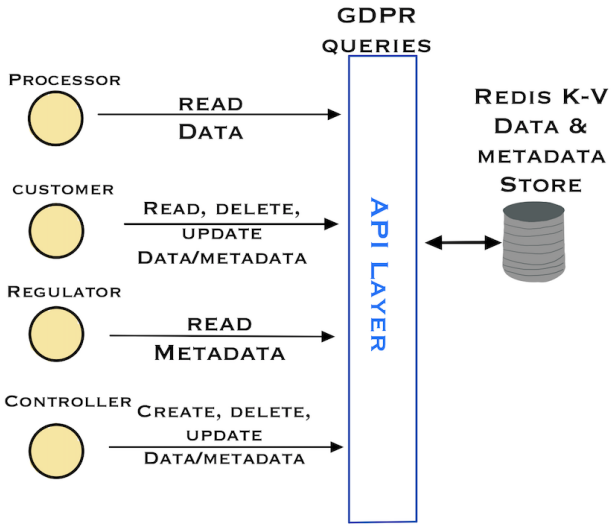
Approach

We wanted to create a simple, dynamic solution, so we built an API layer that can sit between a Redis K-V store and a flask web application, and can enforce GDPR compliance. To do so, we considered four different types of users, who would need to have interactions with the data store: Controllers, Processors, Regulators, and Customers.

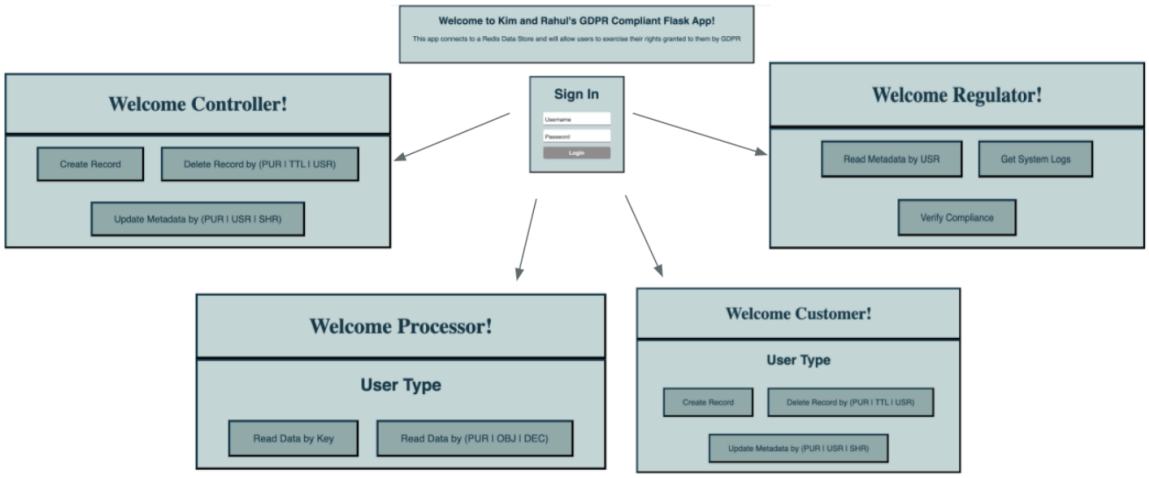
The figure below is an example scenario, showing all the 4 user types. In order to access the services of a company (the **Controller**), **Customers** share their personal data with the company. The Controller is responsible for management and administration of its Customers' data (creation, updates or deletion). Customers should be able to exercise their GDPR rights (read their data, know what purpose their data is being used for, object to their data being used for a specific purpose, delete their data, etc.). Controllers can share Customers' personal data with **Processors** who can process their personal data (train an ML model to generate recommendations, etc.). Finally **Regulators** also need access to the storage system in order to investigate and enforce GDPR laws.



Instead of the four user categories directly interacting with the data store, they would talk through our API layer as shown below:



On top of restricting queries to be GDPR compliant, users also must be restricted from certain APIs. This access control was handled by our web app. Below are screenshots from the simple app built to enforce access control and demonstrate our API layer.



Few simplifying assumptions/design choices we made include:

1. Only one user can be reading or writing any piece of data at any given point in time. Concurrent operations are not supported since it would make things complicated (a single update to a data item corresponds to potentially multiple writes in our storage system).
2. It is assumed if a Customer inputs their data they have consented to a subset of purposes. At any point, after creation, they may amend this subset. If Customers revoke their consent for all purposes, then their data record will be deleted.

Design/Implementation

In this section, we will first describe what capabilities the API layer has. Then we will discuss how we design our Redis data store to support these APIs. And finally we will explain our web application/frontend. Our implementation is available as a GitHub repo.

API Layer

Each piece of data needs to be treated as an active entity that has its own set of rules. Meaning, along with each personal data item (such as phone number, email, etc.), we need to store its metadata attributes. The figure below shows a sample datarecord. For each datarecord, we store the following:

```
ph-1x4b;123-456-7890;PUR=ads,2fa;TTL=7776000;
USR=neo;OBJ=∅;DEC=∅;SHR=∅;SRC=first-party;
```

1. A unique key identify the datarecord
2. The personal data item (in this case, a phone number).
3. Some metadata attributes:
    a. PUR: list of purposes for which this data will be used
    b. TTL: time in secs after which the data would be deleted

c. USR: user to whom the data item belongs to
d. OBJ: list of purposes for which the user objects their data being used for
e. DEC: list of automated decisions where this data was used
f. SHR: list of 3rd party companies with whom the Controller has shared this data item
g. SRC: how the data was obtained

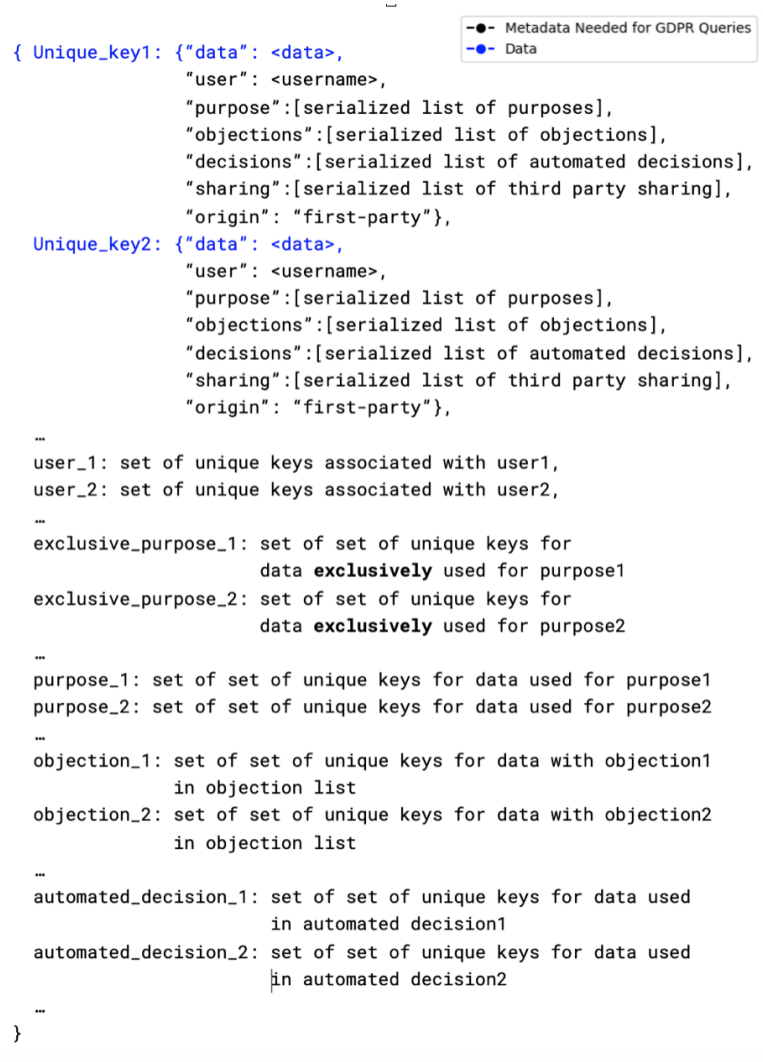
To keep our design simple, we combined both PUR and OBJ fields into one field. If there are purposes a Customer objects to, they will be removed from the PUR field instead of being stored separately in the OBJ field.

As described in the GDPRBench paper [6], these are the capabilities that the API layer should be able to support for different roles:

- Controller
  - Create data records to store Customers' data along with the metadata.
  - When a data record has served its purpose and no longer needed, the Controller should delete those records.
  - A Customer may request their data to be deleted, the Controller should delete all the data records belonging to that user.
  - In addition, the paper also recommends an explicit delete by TTL, however we used Redis' built-in lazy deletion strategy. Since this is a non critical operation, it would be better to process this asynchronously for performance reasons.
  - Further, the Controller should also be able to update metadata such as access lists, third party sharing, etc.
- Customers
  - They should be able to read all their personal data. It should also be ensured that a Customer is accessing only their data and not some other user's data.
  - In order to rectify any inaccuracies, Customers would want to update their data. They may also raise objections for their data being used for certain purposes.
  - We should also provide a deletion function, in case they decide to delete any data record by its unique key.
- Processors
  - Can request data items from Controllers. It is important that they receive only the data items and not the entire datarecord.
  - They should also be able to register their given data for automated decisions, 3rd party sharing, etc.
- Regulators
  - In order to enforce GDPR laws, Regulators would need access to system logs and its features (security features to ensure Customers' data is protected).

In our application, we implemented access control and logging in the web app instead of the API layer, this is discussed further in the Frontend section.

Data Layer



The figure shows the data layout of our Redis K-V store. In our design, we minimized serialization overhead and the memory footprint by using Redis built-in hash maps. Reducing the number of top level keys can reduce the overall memory used by the Redis server. See [8] [9] and [10].

Since GDPR requires reading data by purpose, user, etc, we created secondary indexes to store unique IDs associated with each metadata value. As an example, we may want to filter records that are being used for ads, or belonging to a given user. Secondary indexes enable faster retrieval, but they come at the cost of extra space and write overhead. For instance, to update the purpose list within a datarecord, we need to modify the data record as well as update the purpose and the exclusive purpose sets.

All the above metadata collections were implemented using sets instead of linked lists because we should be able to delete unique IDs from arbitrary locations within them in an efficient way. Further, we have kept the number of metadata sets minimal in our design since having a large number of top level keys would not be space efficient and does not scale very well.

GDPR also requires the Controller to delete the data record if it has served its purpose. To make deletion easier, we have an exclusive purpose set for each purpose separate from the purpose set.

Frontend/Web Application

The frontend was mainly used to demonstrate the capabilities of our API layer. We implemented a simple login scheme. Once logged in, the user was brought to a page that only gave them access to the API functions designated for their usertype. We created a python script to generate dummy data in order to simulate the GDPR queries.

To generate the random data, we used a library called names which would come up with random names for us, we assigned each user a username that was <first initial><last name><random number> and gave them a dummy email address and phone number. Then, for each piece of data we created a GDPR datarecord.

We had a static list of purposes, and third parties. We created a function that would generate a random number between 0 and the size of the list, shuffle the array randomly and then take that number of elements. We did this with our static purpose and third-party sharing lists in order to generate the PUR field and SHR field. Finally, for objections we considered every element within our static list of purposes, but not within the purpose list for that user. When starting our flask app, it loads a certain number of these data records into the redis database and that is the data the web app interacts with.

Results

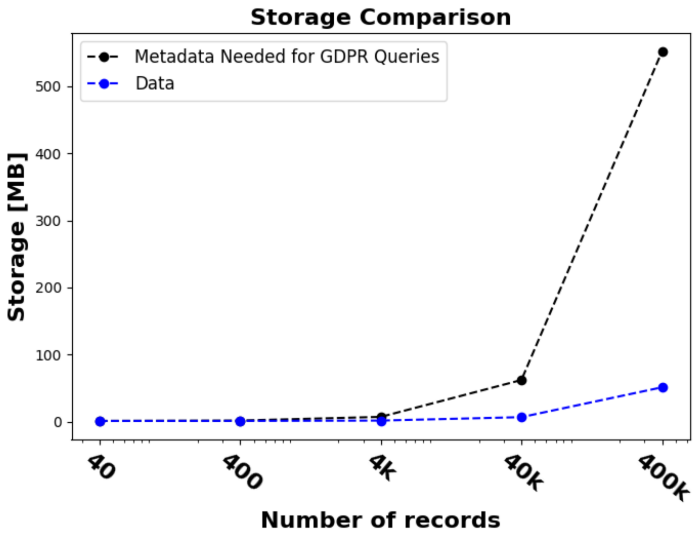
We performed a series of experiments using our API layer in order to test it for correctness, storage overhead and completion time. We discuss the details of these below.

Correctness

In order to test the correctness of our implementation, we ran a series of tests. Please see this document for more details on correctness.

Storage Overhead

We created variable size records and measured the size of the Redis database with both GDPR style storage and Non-GDPR style storage. A GDPR style data record can be found in the Design/Implementation section whereas a Non-GDPR compliant storage data record would simply look like: {Unique_ID_1: Data_1, Unique_ID_2: Data_2, ... }



As shown, GDPR style data grows much faster due to all of the metadata necessary to keep on each record. The curve grows much faster for GDPR records, but the log scale x-axis means that they grow linearly, the steeper curve just shows that it's a faster rate.

Completion Time

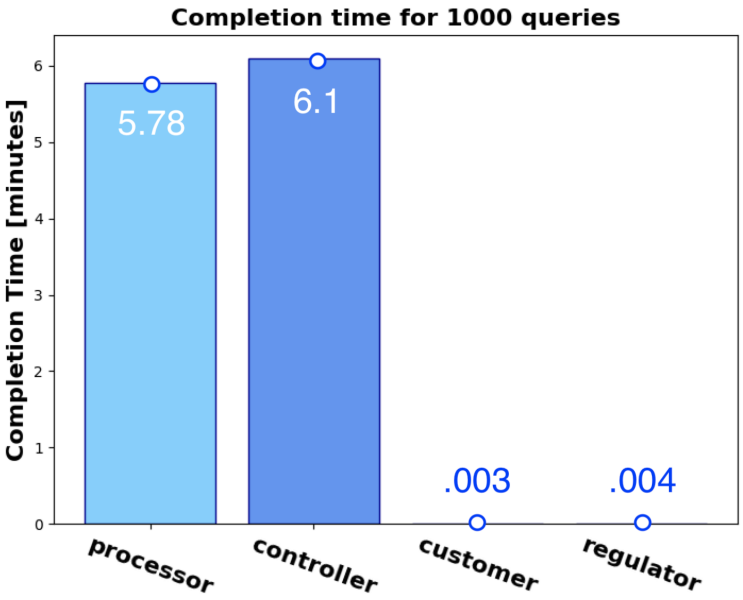
Due to the real-world implications of GDPR completion time is an obvious metric to test our system on.

Completion Time Calculation

In order to calculate the time a function from our API took, we needed to keep a fairly constant database. For more details on calculation, see this document.

Completion Time: By User type

To see how the different workload times compared, we created 100k data-records in Redis and ran 1000 GDPR queries which were evenly split amongst the 4 users. Below are the results.

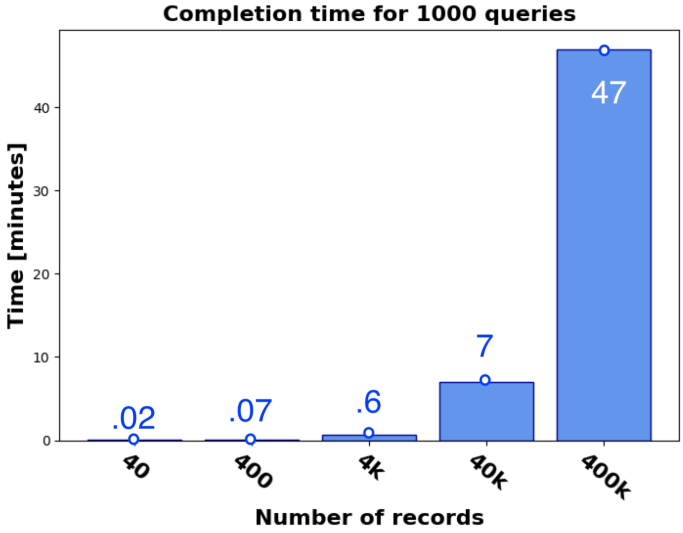


As shown, user types like the Customer and Regulator who are performing O(1) operations are not affected by the large size of the database, however, the Processor and Controller are performing operations dependent on database size, which causes high completion times for their workloads.

Completion Time: Scalability

Finally, we created variable size databases and ran 1000 queries for each size to understand the impact of scalability on our system.

The GDPR compliant queries that are dependent on size of the database scale poorly. The metadata explosion makes GDPR compliance very inefficient as shown by our results.



What we proposed vs. What we Achieved

Normal text is what we proposed, boldface items are what we achieved

1. Complete implementation of middleware/proxy. **We achieved this with the simplifying assumptions mentioned earlier. We had not anticipated until further researching GDPR.**
2. Complete evaluation using benchmarks such as GDPRBench. **We performed our own benchmarking/evaluation because GDPRBench assumes direct access to the database which is not how we set out to build our system. This was heavily based on GDPRBench's evaluation metrics.**
3. Decisions on various algorithms to use in order to both comply with GDPR and improve performance. **This was completed, See design/implementation section for more info on this.**
4. Comprehensive documentation of compliance and limitations of our service. **Assumptions and limitations are included in READMEs in git repo.**

What we learned

- GDPR language is vague. Even lawyers have trouble making sense of it in the context of tech companies' responsibilities to their clients.
- Leveling up to be GDPR compliant will be costly.
    - GDPR systems scale linearly but at a much higher rate than non GDPR systems
    - GDPR takes up over 3x more space than non-GDPR compliant style datakeeping. As we increase the number of users, we not only increase the number of data records, but also the number of metadata sets that hold the data records of each user.
- Avoiding serialization helps with time complexities.
- Minimizing top level keys and storing data records as hashmaps was crucial in our design as it is space optimal.

Conclusion

GDPR is expensive, both in terms of time and space. Small design decisions can be crucial in creating an efficient system. Metadata explosion issue persists regardless of how we design our GDPR compliant system. While companies are getting used to these additional performance and storage costs, it may be necessary to give them financial/technical aid in order to foster competition amongst large tech companies. Otherwise, laws like this may add to the issues, which enable big tech companies to have so much power and become oligopoly-like.

Future Work

1. Setup Redis Pipelines (lowers RTs) and Transactions (handles concurrency).
2. Make optimizations for space. Compare and contrast results.

3. How to make operations efficient as the database scales? Make non critical jobs async and batch those requests (processor registering the automated decision metadata, datarecord deletes).
4. Encrypt all of the data.
5. Heighten security measures on Flask App.

References

[1] EU GDPR: https://gdpr-info.eu/

[2] Comparing privacy laws: GDPR v. CCPA
https://fpf.org/wp-content/uploads/2018/11/GDPR_CCPA_Comparison-Guide.pdf

[3] Top five concerns with GDPR compliance
https://legal.thomsonreuters.com/en/insights/articles/top-five-concerns-gdpr-compliance

[4] EU Admits it has been hard to implement GDPR
https://www.ft.com/content/66668ba9-706a-483d-b24a-18cfbca142bf

[5] A Year in the Life of the GDPR: Must-Know Stats and Takeaways
https://www.varonis.com/blog/gdpr-effect-review/

[6] Understanding and Benchmarking the Impact of GDPR on Database Systems
Paper: https://arxiv.org/pdf/1910.00728.pdf
Slides:
https://04e19274-9945-4166-b1be-95d42dc718a3.filesusr.com/ugd/13b079_4887efca8131475a93b05048804fe0d4.pdf

[7] GDPRBench website: https://www.gdprbench.org/benchmark

[8] Performance comparison of using Redis hashes vs many keys
https://stackoverflow.com/a/62974526/

[9] https://redis.io/topics/memory-optimization

[10] https://matt.sh/thinking-in-redis-part-one