NATIONAL RESEARCH UNIVERSITY
HIGHER SCHOOL OF ECONOMICS

Faculty of Computer Science
Bachelor's Programme "Applied Mathematics and Informatics"

BACHELOR'S THESIS

Software Project on the Topic:

Reactions Storage Service

Submitted by the Student:

group #БПМИ205, 4th year of study Brusenin Dmitrii Aleksandrovich

Approved by the Supervisor:

Parshina Anastasiia Alekseevna
Senior Lecturer
Faculty of Computer Science, HSE University

Co-supervisor:

Meshkov Aleksandr Nikolaevich
Software Developer
Tinkoff

Moscow 2024

Reactions Storage Service

Dmitriy Brusenin
Faculty of Computer Science
National Research University Higher School of Economics
Moscow, Russia
brusenin.dmitriy@gmail.com

***Abstract*—Various services are built on user interactions, such as social media and content platforms. In an effort to enhance user experience, many of these services have recently implemented reactions, a form of engagement allowing users to express their reaction to the posted content. This paper introduces a world-first, open-source, general-purpose reactions storage system. We developed an efficient and flexible service designed to encompass a wide range of potential use cases.**

The efficiency of the system is defined by its capacity to concurrently handle a substantial number of requests, while its flexibility is gauged by its ability to address diverse use cases and its management simplicity. Following the usual practices in distributed systems, each decision involves trade-offs, and this work endeavors to highlight various trade-offs encountered throughout the implementation process, ultimately presenting a production-ready system.

This paper holds relevance for both those interested in leveraging the service and developers seeking an efficient approach for real-time data aggregation.

***Index Terms*—reactions; storage; data aggregation; optimizations; Golang; PostgreSQL**

I. Introduction

In the past decade, reactions have emerged as one of the most popular tools for users to express their emotions, serving as a primary engagement metric within the business domain. Virtually every IT company with a substantial user base has already incorporated their own version of reactions. Notable examples include Telegram with message reactions, Github with comment reactions, Linkedin with post reactions, and even YouTube with its like, dislike reactions. Market research has revealed significant variations in reaction implementations, as they can be applied to a wide range of content entities, each necessitating specific business conditions to be met.

While reactions storage may not be as substantial as, for example, general-purpose databases, storing reactions introduces significant complexity. Therefore, any system can benefit from encapsulating all implementation details under an efficient yet simple abstraction that is guaranteed to cover all necessary use cases and seamlessly fulfilling non-functional requirements.

Our goal is to develop an efficient, flexible and robust service that embodies this level of abstraction. Throughout the process, we will experiment with diverse strategies for storing reactions, comparing and selecting the most suitable one for the service. The implementation, developed in Golang, will include a user-friendly API, mask the implementation complexity, and serve thousands of concurrent requests. Therefore, one of the primary efforts is dedicated to identifying the most suitable approach to store and process data using popular databases familiar to developers.

Our study fills an important gap in existing literature, as to the best of our knowledge, no prior public work has directly addressed the challenges posed by storing user reactions. We rely mostly on existing information about similar proprietary reactions storage implementations, database researches, and established engineering best practices.

The ultimate objective of our work is to conduct a comprehensive exploration of diverse strategies for data aggregation and to develop and publish an open-source system that will become the world-first off-the-shelf solution for storing user reactions.

The rest of the paper is organized as follows. Section 2 presents a review of similar services. In section 3, we describe Reactions Storage's design and key implementation details. We introduce different strategies for storing and processing reactions in Section 4. In section 5 we present the load simulation and evaluate different implementation strategies experimentally. Section 6 describes the final service. We share lessons learned in Section 7; future work in Section 8; and conclude in Section 9.

II. Related Work

All other existing solutions are proprietary, and the majority of information regarding them has been sourced from publicly available API documentations. During the design phase of our system, we used the available information about implementations from platforms such as Telegram, Discord, Facebook, Github, Instagram, Linkedin, and Gitlab. We will list some of them, provide concise overviews and clarify their relevance to our work.

Telegram Reactions

Telegram [13] is a messenger that features a reaction functionality for user messages. According to what we could infer from the Telegram API documentation [14], Telegram implements two distinct flows for storing reactions: the primary flow addresses the majority of reactions, while the local cache shared between devices is used for recent reactions. The primary flow periodically polls for reactions "for visible messages (that were not sent by the user) once every 15-30 seconds [14]". The second flow is used only in specific cases, utilizing the local cache that is shared among other logged-in sessions.

Telegram reactions offer a simple API for creating, retrieving, and deleting reactions. To create or delete a reaction, users send a message ID and a vector of reaction IDs. Retrieving reactions involves providing message IDs, with each ID returning a list of $reactionCount$ objects, essentially pairs of reaction ID and count. Similarly to other implementations, Telegram supports two types of reactions: normal (Unicode symbols) and custom. Custom reactions can also include a Lottie [15] animation.

Telegram reactions include several settings, with two particularly noteworthy. The first involves a limitation on the number of unique reactions per message, managed by assigning an index to each new reaction within a message. The second pertains to the maximum unique reactions per user within a single message, typically limited to 1; sending a second reaction for the same message will replace the first.

The primary criticism from users of Telegram reactions is the lack of settings to customize the reactions-related UI and reactions inconsistencies that sometimes appear across different platforms.

Similarly to Telegram reactions, our work anticipates clients polling for reactions, support for both Unicode and custom reactions, and implementing a configuration with similar settings. For instance, users will be able to restrict the maximum number of unique reactions per message. While there are numerous similarities, unlike Telegram reactions, we will not implement the second flow for recent reactions as it requires integration with client-side code. Instead, our focus is to implement a general-purpose system.

Discord Reactions

Discord [16] is a social platform that also integrates reactions for messages. As outlined in the documentation [17], it provides a concise API for creating, retrieving, and deleting reactions. To retrieve reactions, it requires a message ID and a reaction ID, returning an array of users who applied this reaction. This operation supports pagination through parameters such as "after" and "limit."

Similar to Telegram's implementation, Discord supports both Unicode and custom emojis for reactions. Discord utilizes client-side cache for storing reactions data and imposes a limit on the maximum number of unique reactions within a single message.

Criticism from users of this implementation focuses on the absence of an API option to add multiple reactions at once and on so-called "ghost reactions" that appear, when there are more reactions than users in the chat. Some inconsistencies are also reported within individual interfaces, such as discrepancies between the displayed reaction counter and the displayed number of users who applied this reaction.

Our work will also support custom reactions. Retrieving the whole list of users who applied given reaction with pagination will be an additional feature for our service that will be implemented in future work. Each operation will be either idempotent according to its implementation or support an idempotency key in order to prevent ghost reactions.

LinkedIn Reactions

LinkedIn [18] is a social media platform featuring reactions for every post, comment and repost on the platform. Unlike Telegram or Discord, LinkedIn has a strictly limited selection of reactions, currently offering only six custom reactions. LinkedIn displays only the total count of reactions and does not reveal counts for each reaction separately unless a user clicks on the button.

The API documentation [19] indicates that the GET operation takes an entity ID and a sort parameter, returning a list of reaction events without aggregation into $(reaction, count)$ pairs. The sort parameter offers different options such as chronological, reverse chronological, and relevance.

Public criticism of LinkedIn reactions consists of the users' desire for a more varied selection of reactions, including negative reactions.

In contrast to LinkedIn's implementation, our work will not be limited by any selection of reactions and will return aggregated reactions data. Supporting sort parameters will be an additional feature.

III. Design

In this section, we first review the requirements for the general-purpose Reactions Storage, then evaluate the expected workload. Finally, we present an architecture and delineate key implementation details.

Gathering Requirements

Functional Requirements. The primary functional requirements include support for adding, removing, and retrieving reactions for individual content entities such as posts, videos, messages, or any other objects with user reactions (hereinafter referred to simply as 'entities').

Another crucial aspect is the ability to impose restrictions on adding new reactions. One common restriction is setting a $max\_uniq\_reactions$ parameter, which limits the number of different reactions users can add to a single entity. This parameter is business-driven, aiming to maintain a simple user interface by avoiding the display of an excessive number of reactions under a single piece of content.

Additionally, there is often a scenario involving conflicting reactions, such as like and dislike, or a 1 to 5-star rating system. In such cases, reactions are termed as $mutually\_exclusive\_reactions$. To address this, the system must support a $mutually\_exclusive\_reactions$ parameter, defining groups of conflicting reactions. This ensures that users are prevented from adding conflicting reactions to a single entity according to business requirements.

There may also be situations where users need to be restricted from adding more than one reaction to an entity. This can be achieved by creating a mutually exclusive reactions group that includes all available reactions.

Given that we are developing a general-purpose Reactions Storage system, it is essential to support reactions configuration. Clients should have the ability to update all the parameters mentioned above and modify the set of reactions

TABLE I: Design Summary

Functional Requirements	Non-Functional Requirements	Load Estimation
Add, remove and retrieve reactions	Availability, Reliability and Responsiveness	Read RPS: 3500
Configure $max\_uniq\_reactions$	Configuration Flexibility and Simplicity	Write RPS: 70
Configure $mutually\_exclusive\_reactions$	Efficiency	Storage Size $\leq$ 1TB
Update reactions configurations	Durability	Row Count per Table $\leq$ 1T
Multitenancy support		

available for users (referred to simply as a $reaction\_set$). Additionally, we have considered more intricate scenarios, such as the multitenancy of different customers or situations where a single application requires different rules for different content. To accommodate these scenarios, Reactions Storage should be able to handle multiple configurations simultaneously.

Non-Functional Requirements. Among the non-functional requirements, we emphasize the following. Firstly, availability, reliability, and responsiveness. Although in our case, availability mostly depends on the deployment configuration (number of replicas, etc.), our architecture still influences it. The ability of our API to handle as many concurrent requests as possible directly affects the reliability and responsiveness of the entire system.

Secondly, efficiency. Our system is expected to process requests as quickly as possible since end-users anticipate instantaneous functionality. While client-side code can mask backend slowness by employing optimistic strategies and displaying information as if the backend instantly processed the request and returned a successful status code, this approach may lead to a higher rate of inconsistencies and a poorer user experience.

Thirdly, configuration flexibility and simplicity. Clients should be able to express any intent regarding reactions configuration without encountering unnecessary complexities.

Lastly, durability. Formally, Reactions Storage is expected to securely store user reactions regardless of caches and other in-memory storages. Data should always be replicated on a cold storage such as SSD or HDD.

Load Estimation.

In order to design and test the system properly, we require load estimation. Unfortunately, we do not possess exact numbers, but we have made our best effort. Since Reactions Storage service will be utilized within the Tinkoff [20] app for social media content reactions, our evaluation will be based on their load expectations, publicly known numbers about social media platforms, and our own assumptions.

RPS. The Tinkoff app expects up to 30M DAU (Daily Active Users). According to research [4, 5] and publicly available information about engagement rates on social media platforms, engagement decreases with the number of followers, typically ranging from 1-2% when the number of followers approaches one million. The only exception may be Twitter, with an average engagement rate of 4.75% [6]. We will assume that each active user views 10 posts daily, has a 1% engagement rate (probability to react to a post), and puts no more than 2 reactions on a single post. With these assumptions, we can calculate that there are 60M new reactions daily ($30M \times 10 \times 1\% \times 2$), resulting in approximately 70 new reactions each second, which is the write RPS (Requests Per Second). The read RPS will be equal to 3500 ($30M \times 10$ / $(24 \times 60 \times 60)$).

Storage. Estimating storage parameters such as size and row count is difficult as it heavily depends on the implementation. However, we provide upper bound estimations: the storage size is not expected to exceed 1TB of data, including replicas, and is more likely to be in the range of hundreds of gigabytes. The row count for each table is not expected to surpass 1 trillion, as this is an upper bound expectation for the total number of reactions in the Tinkoff app.

Network. The network load is negligible since reactions data is relatively small, and thousands of RPS do not pose any challenge for modern computer networks.

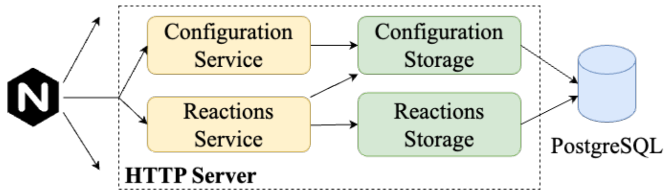

Fig. 1: Reactions Storage architecture

Architecture

Figure 1 shows the architecture of the Reactions Storage system. While it does not inclued all features and integrations, it highlights the three most crucial components: Nginx [22], an HTTP [11] server, and a PostgreSQL [21] server. The Nginx server distributes all incoming requests among the HTTP servers, which in turn process requests by making queries to the PostgreSQL server. Each HTTP server comprises two parts: the configuration part, responsible for storing and managing configuration settings, and the reactions part, responsible for handling user reactions in a similar manner.

Configuration. Before clients can begin using Reactions Storage, they need to provide a reactions configuration. As shown in Figure 2, this configuration comprises three files: reactions.yaml, reaction_sets.yaml, and namespaces.yaml. The

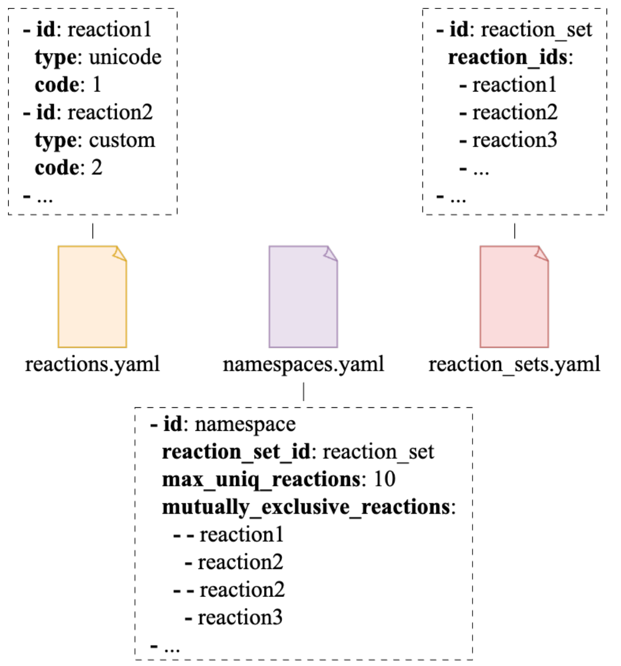

Fig. 2: Reactions Storage configuration

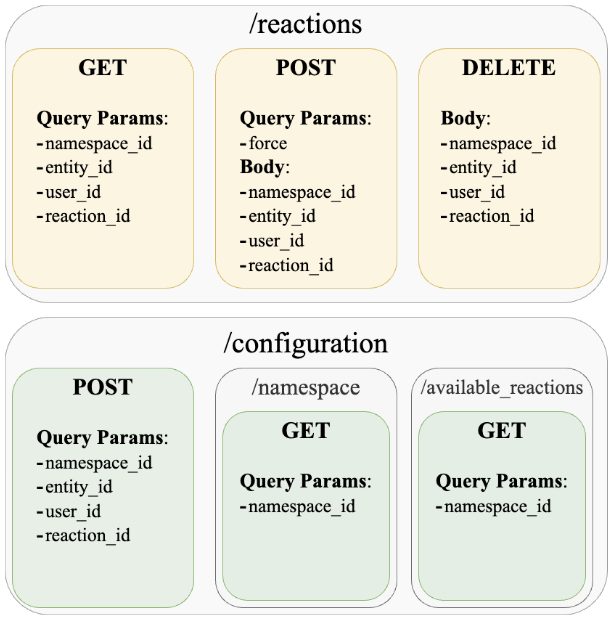

Fig. 3: Reactions Storage API

primary objective of configuring Reactions Storage is to create appropriate namespace(s).

To streamline this process, there are two additional files. The first file, reactions.yaml, outlines all the reactions available for system use. Each reaction is defined by two main fields: id and $type$. The $type$ field must be either ”unicode” or ”custom”. A ”unicode” reaction refers to a Unicode symbol, with its code provided in the $code$ field. On the other hand, a ”custom” reaction indicates a reaction with a custom image, and the image URL should be specified in the url field. Once all the necessary reactions are specified in reactions.yaml, the client proceeds to create one or more reaction sets in reaction_sets.yaml, which group reactions defined in the previous step.

Finally, to address the multitenancy problem, the Namespace abstraction was introduced. Essentially, a namespace configuration consists of a set of rules for a given reaction set, and each namespace should be registered in the namespaces.yaml file. It includes fields such as id, $reaction\_set\_id$, $max\_uniq\_reactions$, and $mutually\_exclusive\_reactions$. The Namespaces system enables multitenancy and provides additional configuration flexibility, allowing for the implementation of different sets of rules within a single application.

API. The Reactions Storage API was carefully designed to offer granularity and simplicity, ensuring it can meet diverse client requirements while maintaining minimal performance overheads. As shown in Figure 3, it comprises three primary handlers for managing user reactions: GET, POST, and DELETE methods for the $/reactions$ resource.

The GET method allows to retrieve reactions, taking parameters such as $namespace\_id$, $entity\_id$, and $user\_id$. It returns a union of two objects: an array of counters, indicating the frequency of each reaction associated with the specified entity, and user reactions, represented as an array of reaction IDs that the user has applied to the entity.

For the POST method, it handles scenarios where users click to add a reaction to an entity. Alongside parameters similar to those in the GET method, it also requires $reaction\_id$. Additionally, it supports a query parameter, $force$, enabling the override of any conflicting reactions. For instance, if a user initially likes a post and subsequently dislikes it, the like reaction will be replaced with the dislike reaction upon the second click, using the $force$ flag.

Conversely, the DELETE method addresses cases where users remove previously added reactions. It shares the same input parameters as the POST method.

The API plays a significant role in the performance of the overall system. Enhancing the efficiency of the API with more comprehensive handlers is a possibility. For instance, we could include the current state of reactions in the response of the POST method, as it already needs this information to verify configuration constraints such as$max_uniq_reactions$ and $mutually\_exclusive\_reactions$. While this adjustment may slightly reduce latency, it could also introduce complexity to the system and potentially result in inconsistencies. For instance, optimizations in the GET method might cause it to return stale data different from what the POST method provided moments earlier. Therefore, we have decided not to proceed with such enhancements unless it becomes absolutely necessary.

Additionally, there are three more technical handlers for managing configuration.

The GET $/namespace$ endpoint returns a namespace configuration for a given ID, while GET

$/namespace/available\_reactions$ provides an array of reaction IDs for a given $namespace\_id$.

The POST $/configuration$ endpoint allows updating the current configuration. This method overrides the existing configuration with a new one. It involves no additional steps, except for verifying the correctness of the new configuration itself. Thus, there is no check for compatibility with the current storage state. This means it is possible to add a new constraint to the current namespace that is incompatible with existing reactions, or to accidentally remove an existing namespace by misspelling it in a new version. While the first case might be acceptable and no good solution is found, the second case is dangerous, and future work is required to make the API safer.

Technology Stack

Reactions Storage comprises two main components: a database and an HTTP server.

Database The database plays the most crucial role in determining the overall latency of the system. While it might seem appealing to use in-memory storage to minimize latency as much as possible, we consider this approach too expensive for a general-purpose solution. Based on our estimations, we anticipated storing hundreds of gigabytes of data, which would be costly in terms of RAM, especially when considering replicas.

Furthermore, we expected our datastore to support aggregation, as the final implementation algorithm had not yet been chosen at that point We may also require additional handlers in the future, such as retrieving users who have applied a given reaction to a specific entity. Therefore, a SQL database is the most suitable option for Reactions Storage. SQL databases are widely used and well-suited for general-purpose systems.

The initial decision to choose PostgreSQL was based on several factors. Firstly, it is an industry-proven technology that can be easily integrated into the existing infrastructure of most companies. Additionally, its API is highly expressive, allowing for numerous optimizations.

HTTP Server For the HTTP server, there are numerous options available. We chose the Go language [23] for two main reasons: firstly, it meets our requirements, and secondly, it is familiar to Tinkoff developers who will be involved in future work on Reactions Storage. While languages like C++ may offer higher speed, we anticipated that the database would be a primary bottleneck. Therefore, Golang was chosen for its simplicity, convenient dependency management, extensive set of built-in tools, and suitability for our web server needs.

The service will implement an API similar to REST [9] due to its popularity and widespread usage. However, we may consider support for an RPC [10] API in future work.

We selected Gin [24] as the web framework for Reactions Storage. Firstly, we acknowledge that ORM (Object-Relational Mapping) gives a false impression of solving all problems [7], and modern systems require more fine-grained access to the database, which direct SQL provides. Therefore, we decided against using frameworks like Beego [26] or GoBuffalo [27]. Secondly, Gin demonstrates decent performance according to TechEmpower Benchmarks [8]. Thirdly, Gin is popular, with comprehensive documentation and one of the largest communities, which is crucial for the long-term support of Reactions Storage. Additionally, Gin has a better learning curve and offers a wealth of additional functionality [30], such as caching, which can be installed with contributed middleware [25].

Among other important libraries, we decided to use pgx [28] as our PostgreSQL driver and zap [29] for logging. Both are efficient and have stood the test of time.

IV. Strategies

This section describes different strategies for storing reactions, their performance bottlenecks, potential improvements, and and their suitability for serving as a general-purpose Reactions storage.

Strategy 0: MVP

Following iterative development [2, 3], we initially implemented an MVP. The Reactions Storage MVP meets all functional requirements outlined in the Gathering Requirements subsection and features the simplest implementation. The rationale behind prioritizing the development of the MVP was to establish end-to-end tests, which could then be used to verify the correctness of more complex strategies.

Implementation Algorithm In this subsection, we describe the database model and an algorithm for the three primary handlers: GET, POST, and DELETE methods on the $/reactions$ resource. Inside the database, we maintain three tables for configuration, structured similarly to how they are organized in files, and a single table named $user\_reaction$ to store user reactions, functioning essentially as an event log.

To handle GET requests, Reactions Storage sends two SQL queries to the database. The first query aggregates all events for the given entity and extracts counters for each reaction, while the second query retrieves all reactions added by the specified user. Next, the service combines the results of these two queries into an HTTP response and returns it to the client.

When adding a new user reaction, Reactions Storage initially obtains an advisory lock to prevent another reaction from being added to the entity concurrently. Then, it sends one query to the database to retrieve a namespace configuration and the same two queries as in the GET request. It then checks for two constraints specified in the namespace: $max\_uniq\_reactions$ and $mutually\_exclusive\_reactions$. If these constraints are met, it simply inserts a new row into the $user\_reaction$ table; otherwise, it returns a 403 HTTP status code.

Handling remove requests is the simplest process: Reactions Storage sends a single DELETE query to the database and finishes the operation.

It is worth noting that serving GET requests is accurate even without using Repeatable Read isolation. This is because inconsistency between the two query results would only occur if the same user updated reactions to the given entity. In such cases, the client-side code should not trust the response of concurrent GET requests, as they may not contain up-to-date information.

Potential Improvements. The implementation is kept as simple as possible, without PostgreSQL indexes and with an inefficient database configuration. Therefore, there are many potential improvements. Since Strategy 1 was built on top of the MVP, most of these improvements will be described in the next section, Strategy 1: Runtime JOIN.

Evaluation. The MVP itself is not suitable to serve as a general-purpose Reactions Storage, as it lacks the efficiency required to handle thousands of concurrent requests.

Strategy 1: Runtime JOIN

The first strategy, named "Runtime JOIN" due to its reactions retrieving algorithm, operates similarly to the MVP. It sends three requests to serve GET requests, incorporating a runtime join to extract counters for each reaction.

Optimizations Unlike the MVP, the current strategy incorporates numerous optimizations.

Firstly, it utilizes indexes for each table, ensuring that each service query to the database uses Index Scan instead of Sequential Scan. For the $user\_reaction$ table, we employed the default B-Tree index, as we anticipate numerous updates in a real-world workload. Nonetheless, we acknowledge the importance of selecting the appropriate index type and plan to evaluate other options in our future work on Reactions Storage.

Secondly, it was observed that obtaining a PostgreSQL connection from pgxpool results in a significant delay, accounting for up to 90 percent of request latency. Consequently, the maximum number of connections to PostgreSQL was increased to 500. Through experimentation, it was determined that there was a notable improvement after increasing the maximum number of pool connections to 100, further improvement after increasing to 250, and still a noticeable speed up after reaching 490 connections, with no visible improvement beyond that. As a result, pgxpool was configured to maintain a minimum of 40 active connections and a maximum of 490 active connections. In addition, we increased $shared\_buffers$ to 2GB to accommodate the increased number of concurrent connections and the additional memory required to cache disk read operations. Furthermore, we set the $kernel.shmmax$ Linux parameter to 2684354560, which is 2.5GB, to allow the kernel to allocate 2GB shared buffers for the database. Additionally, we configured the $MaxConnLifetimeJitter$ pgxpool parameter to 1 second, which helps prevent all connections from being closed simultaneously, thereby avoiding starvation of the connection pool.

Thirdly, we aimed to further reduce the contribution of connection acquisition to the overall latency by minimizing the number of times it is called. By default, each call to the pgxpool method acquires a connection to execute an SQL query. To prevent multiple acquisitions of connections within a single request, we redesigned each method implementation to acquire a connection once at the beginning and then reuse it for subsequent queries within the same HTTP request.

Forth, we addressed the high latency in acquiring connections and observed CPU usage metrics indicating that goroutines were starved for CPU time when handling thousands of concurrent requests. Through experiments involving a gradual increase in the number of CPU cores, we determined that 16 cores is optimal for handling the workload of Reactions Storage and for conducting fair evaluations of different strategies.

Fifth, we attempted to tune the PostgreSQL WAL (Write-Ahead Log) size parameters. We conducted experiments with various pairs of (min_wal_size, max_wal_size) parameters: (80MB, 1GB) (default), (160MB, 2GB), and (320MB, 4GB). While we anticipated that increasing the max_wal_size parameter would enhance overall performance by reducing the frequency of WAL checkpoints, especially following the increase in $shared\_buffers$, experiments revealed that altering the WAL size parameters either had no effect on latencies or slightly worsened them. Therefore, we decided to keep the default values for the WAL size parameters.

Sixth, we aimed to improve query plans, which can be examined using the EXPLAIN command preceding the query. We observed that the default value of the $effective\_cache\_size$ PostgreSQL parameter, responsible for the planner's assumptions regarding available memory, is set to 4GB. This exceeds our requirements, as the total size of table indexes in our workload, detailed in the Experiments section, is slightly over 2GB. Consequently, the analysis of each query revealed that the planner consistently prioritized Index Scan over Sequential Scan.

Potential Improvements. Among potential improvements, we highlight the following.

As mentioned earlier, we have not thoroughly explored different indexes yet and cannot be certain that B-Tree is the optimal choice. Additionally, we have not experimented with grouping queries into batches where possible. This could potentially improve overall latency by reducing the Requests Per Second (RPS) to the database by approximately 20%, which is significant when handling thousands of concurrent requests.

Furthermore, while examining a Flame Graph [31], we observed that a considerable amount of CPU time (about 29%) is spent on network system calls within the net/http Golang library. It may be possible to bypass the kernel or reduce the number of system calls to save CPU time.

Evaluation. With all the optimizations implemented, the service demonstrates decent performance while maintaining a simple and flexible implementation. Therefore, the strategy can be considered for the final Reactions Storage solution.

Strategy 2: No JOIN

The second strategy, named "No JOIN," avoids using JOIN operations.

The main difference in implementation from the previous strategy is the use of two tables for storing user reactions. The first table, *reactions_count*, stores one row for each entity. Each row contains a JSON object representing a map from $reaction_id$ to the number of times that reaction was added to the corresponding entity. The second table, *user_reactions*, stores an array of *reaction_id* for each (user, entity) pair, representing the reactions added by the given user to the given entity.

For serving GET requests, this strategy still requires two SQL queries: one to retrieve counters for each reaction from the *reactions_count* table, and another to find all reactions added by the given user from the *user_reactions* table.

To serve POST requests, it follows a similar approach to the previous strategy by acquiring the advisory lock and sending three SQL queries to check namespace constraints. It then executes two SQL queries in batch: the first query inserts a row into the *reactions_count* table and increments the corresponding counter in case of a conflict (if it is not the first reaction to the entity and the row already exists), while the second query inserts a row into the *reactions_count* table and appends a new *reaction_id* in case of a conflict (if it is not the first reaction by the given user to the entity).

Handling DELETE requests operates similarly to POST requests, but instead of incrementing the counter, it decrements it in case of a conflict. And, instead of appending a new *reaction_id* to the array, it removes the reaction from it.

Optimizations. The primary optimization involves setting a $fillfactor$ for the database tables. This parameter instructs PostgreSQL to initially reserve some additional disk space on data pages. ”This gives UPDATE a chance to place the updated copy of a row on the same page as the original, which is more efficient than placing it on a different page, and makes heap-only tuple updates more likely.” [21]. This indeed improves performance because, unlike Strategy 1, this strategy updates rows on most POST and DELETE requests. Through experimentation, we determined that a $fillfactor$ of 93 is optimal for *reactions_count* and 97 for *user_reactions*.

Additionally, this strategy inherits all optimizations from Strategy 1 without modification because the total size of table indexes remains slightly over 2GB, and all other optimization grounds are still applicable.

Evaluation. In comparison with Strategy 1, this implementation is more sophisticated and offers less flexibility in its usage (for example, it may not be efficient to retrieve all users who added the given reaction to the given entity if needed in the future). However, this strategy fulfills all requirements and achieves the best performance as no JOIN operation is required. Therefore, it is well-suited for a general-purpose Reactions Storage.

Strategy 3: Async JOIN

The final strategy aims to avoid the runtime JOIN operation by executing it asynchronously and sacrificing data freshness. It maintains the same database layout as Strategy 1. However, rather than executing a JOIN operation each time reactions need to be retrieved, it constructs a materialized view called *entity_reactions* with the following definition:

```sql
CREATE MATERIALIZED VIEW IF NOT EXISTS
    "entity_reactions"
AS
    WITH tmp AS (
        SELECT
            "namespace_id",
            "entity_id",
            "reaction_id",
            COUNT(*) AS "reaction_count"
        FROM "user_reaction"
        GROUP BY namespace_id, entity_id,
            reaction_id
    )
    SELECT
        "namespace_id" || '__' || "entity_id"
            AS "namespace_id__entity_id",
        json_object_agg(
            "reaction_id", "reaction_count"
        ) AS "reactions_count"
    FROM tmp
    GROUP BY namespace_id, entity_id
WITH NO DATA;

CREATE UNIQUE INDEX entity_reactions_pkey ON
    entity_reactions(namespace_id__entity_id
    text_ops);

CREATE INDEX entity_reactions_pkey_hash ON
    entity_reactions USING HASH
    (namespace_id__entity_id text_ops);
```

Optimizations. In order to continue using SELECT queries on *entity_reactions*, it is necessary to refresh the view concurrently. Unfortunately, this process takes several minutes (9 minutes on average), which is not suitable for the workload of the Tinkoff app. In this workload, the load is mostly distributed among a relatively small number of entities (hundreds to thousands). Consequently, such a slow refresh rate would result in noticeably stale data and inconsistencies across users.

In an effort to improve the situation, we began exploring possible optimizations. We observed that disk load, primarily consisting of I/O operations and the volume of read/write data, was fluctuating significantly. We hypothesized that these fluctuations were caused by flushes data to disk due to a lack of memory for performing aggregation.

Figure 4 shows the dependence of concurrent refresh on *work_mem* PostgreSQL parameter. By increasing the *work_mem* parameter, we managed to reduce the refresh time to 3.43 minutes with *work_mem* set to 8GB. However, further increases in memory did not yield any additional improvements.

Evaluation. Due to the significant refresh time, we concluded that this strategy is not suitable to be a general-purpose solution for storing reactions. While it might be a suitable option for certain types of workloads, such as when the load is distributed among a large number of entities, similar to

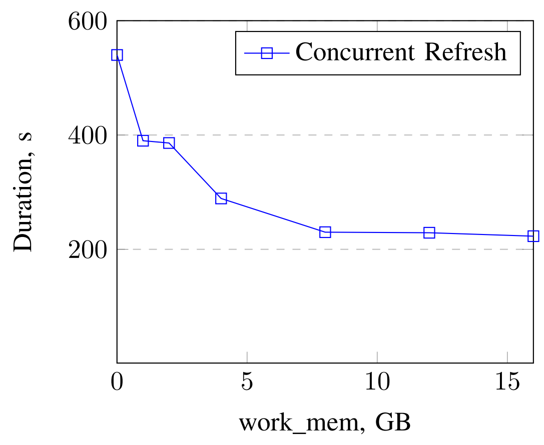

Fig. 4: CONCURRENT REFRESH duration dependence of work_mem parameter (10.2M reactions)

the case of GitHub with millions of discussion messages and relatively low engagement for a particular one, this strategy cannot properly meet the requirements of a workload that necessitates a higher refresh rate.

V. Experiments

In this section, we describe the load simulation used to evaluate Reactions Storage and compare different strategies.

Simulation

To assess the efficiency of Reactions Storage, it is essential to establish a universal benchmark or a method to create customized benchmarks tailored to client requirements. Unfortunately, simulating a real-world workload with static request samples poses a challenge, and existing benchmarking solutions do not offer sufficient flexibility in scripting to simulate user behavior effectively.

As a result, we decided to develop our own simulation tool to reproduce real-world loads as accurately as possible. This approach allows clients to configure their workload and benchmark Reactions Storage accordingly.

Implementation Details In essence, the simulation emulates real users and their interactions with entities. Each entity belongs to a single group called a *topic*, which could represent a chat in a messenger or a feed in social media. The simulation allows topics to be configured by specifying their number, their size (number of entities within each topic), and whether the topics should be shuffled for each user individually.

The simulation begins with users selecting a random topic and initiating their actions on topic entities. There are five actions available to users in the simulation: switch topic, scroll, add reaction, remove reaction, and quit. The probability of each action can be configured to reproduce the desired workload.

Moreover, the simulation is organized into multiple turns, each representing a unit of time. During a single turn, a client performs exactly one action and then waits for the end of the current turn if it is not yet complete. Each turn has a minimum duration specified in milliseconds and ends when both the user completed their action and the minimum duration has passed. Therefore, the client must configure the number of users, the number of turns, and the minimum turn duration for the simulation.

Additionally, the configuration includes parameters such as the number of entities visible on a single screen (which can also simulate pagination in entity loading), the delay between automatic refreshments of visible entities (to handle cases where users do not scroll and reactions data for visible entities should be updated), and others. Figure 5 illustrates a complete configuration example.

```
rules:
  turns:
    count: 180
    min_duration_ms: 1000

  users:
    count: 10
    turn_start_skew_ms: 900

    id_template: user_%06d

    screen:
      visible_entities_count: 3

    app:
      background:
        refresh_reactions:
          timer_in_turns: 3

    action_probs:
      switch_topic: 5
      scroll: 75
      add_reaction: 10
      remove_reaction: 5
      quit: 5

  topics:
  - count: 10
    namespace_id: namespace
    size: 1000
    shuffle_per_user: true
```

Fig. 5: Simulation configuration example

Among the important details is that users take turns asynchronously. This means that one user can start turn 3 while another is still working on turn 1. This can occur if the server is still processing the request of the second user, and this user cannot finish their turn until their action (request) is completed, and the HTTP response is received. This asynchronous behavior mirrors real-world workload dynamics, and we observed that the lag between users does not exceed two turns.

Additionally, background refreshment of visible entities will not be applied to entities that were updated by the current user. Essentially, if a user adds or removes a reaction from an entity while its refreshment is in progress, the refreshment process will be canceled to avoid overwriting the last user's action.

Furthermore, there is a rule for action probabilities. Each user action has a configurable probability, and when a user selects the next action, it randomly chooses from the available actions using a weighted random selection algorithm. If the scroll action is not available (which occurs when the end of the current topic is reached, as only forward scrolling is possible), its weight is added to the switch topic action, which is always available.

It is also worth noting that the switch topic action can select the current topic, simulating scrolling to the top of the current page. The simulation attempts to share objects wherever possible (for example, namespace configurations are shared across users and entities because they are considered immutable during the simulation), ensuring that the simulation does not consume a large amount of RAM.

Observability To observe the system during the load simulation, we integrated the Reactions Storage and Simulation systems with a monitoring setup, which includes Prometheus [33], Grafana [34], Pushgateway [35], and Node Exporters [36].

Reactions Storage calculates primary request metrics, such as request duration histograms and total request counts. It also tracks the number of executed SQL queries and their latencies. The HTTP server exposes a $/metrics$ handler to allow Prometheus to scrape the collected metrics.

Since requests can be aborted before reaching Reactions Storage's middleware and may not be recorded, the simulation system also records request metrics and sends them to Pushgateway. This ensures that Prometheus can pick them up later and helps measure any significant networking overhead between the Simulation and the Reactions Storage.

Additionally, to observe the general state of the VMs and identify hardware bottlenecks, each VM has a running instance of Node Exporter, which serves as a target for Prometheus scrape jobs.

All scraped data is pulled by Grafana, which displays various dashboards for comprehensive monitoring. These dashboards are carefully backed up into the code repository and can be loaded into any running Grafana instance with just a single command.

Setup

Figure 6 illustrates the experimental setup and the interaction among its four main components: Reactions Storage, Simulation, Database, and Monitoring. Each component is hosted on a virtual machine on the Yandex Cloud [12] platform, with multiple Docker [32] containers running within it.

Hardware. Finding a hardware setup that equally evaluates different strategies is challenging, but we found one that does not bottleneck any of the tested strategies under our experimental workload.

The Reactions Storage component had 1 instance with 16 Intel Ice Lake vCPUs, 16GB RAM, and a 50GB SSD primarily used for networking and logging.

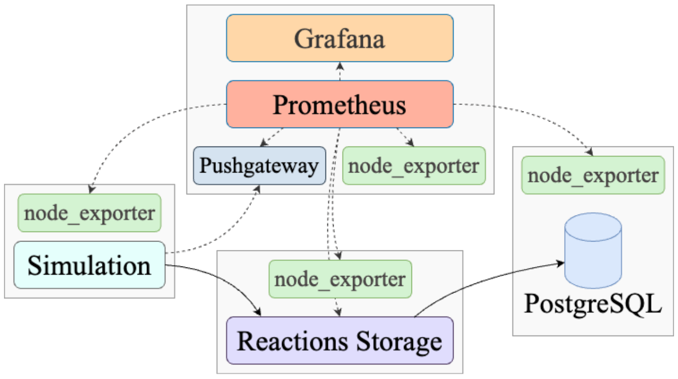

Fig. 6: Simulation setup

The Database component had a similar configuration but with 8 vCPUs, 8GB RAM, and a 100GB SSD with a block size of 4KB, 4000 max IOPS (read/write), and 60 MB/s max bandwidth (read/write). Notably, 8GB RAM was sufficient for the database to prefer Index Scan over Sequential Scan whenever possible, and the SQL queries of the tested strategies did not require other in-memory operations. However, additional memory may be beneficial for maintenance operations.

The Simulation component was set up with 6 Intel Ice Lake vCPUs, 12GB RAM, and a 20GB disk VM.

Lastly, the Monitoring subsystem had 6 of the same vCPUs, 30GB RAM, and a 150GB SSD.

Configuration. Prometheus was configured to scrape metrics from Node Exporters, Pushgateway and Reactions Storage's $/metrics$ handler. It had a $scrape\_interval$ of 30 seconds and a $scrape\_timeout$ of 15 seconds. The database settings, as described in the Strategies section, included parameters such as $max\_connections$, $shared\_buffers$, and $kernel.shmmax$.

For Reactions Storage, Gin was set to release mode, pgxpool had a minimum of 4 active connections and a maximum of 490 connections, with $MaxConnLifetimeJitter$ set to 1 second.

Workload. In our experiments, we aimed to simulate the real-world workload of the Tinkoff app. Among the most important parameters, we considered the probability distribution of user actions, which was as follows: 10% for switching topics, 70% for scrolling, 9% for adding reactions, 1% for removing reactions, and 10% for quitting the app.

Given that the Tinkoff app is similar to a social media platform but with a relatively small number of topics (groups of content entities), and our workload was designed to span a relatively short duration of 240 turns, we configured the simulation to have 10 topics, each with a size of 30 entities, and all entities within topics were shuffled individually for each user.

Additionally, we set the number of visible entities on the screen to 10 to simulate entities loading with pagination, and refreshed the visible entities every three turns. The workload ran for 240 turns, with a minimum turn duration of 1 second.

This configuration would result in a simulation duration of 240 seconds if the server efficiently processed all requests before the minimum turn duration elapsed.

We observed that an RPS, achieved by setting the number of users to 500, closely resembled real-world figures. Therefore, we chose to evaluate our workload with 100, 500, and 1000 users.

Process Before each simulation launch, the database was populated with 10 million random reactions to bloat a table to a size closer to its real-world counterpart. These 10 million reactions did not interfere with the simulation reactions and only impacted the performance of the database.

Additionally, we populated the database with 200,000 reactions that were applied to the entities appearing in the simulation, thereby affecting the simulation itself. To accomplish this, we executed a "setup" simulation with users distinct from those in the experimental simulation. This step was necessary to ensure that entities had already accumulated a certain number of reactions when the experimental simulation started.

Therefore, before each experimental simulation launch, we reset the reactions stored in the database to 1,200,000 and vacuumed all tables to eliminate the impact of previous simulation launches. Upon completion of the simulation, we queried Prometheus to retrieve the target metrics.

Results

Table II shows the most important metrics captured after the simulation launches. The data shows that the "NO JOIN" strategy exhibits the best overall performance, as evidenced by its request latency.

The MVP strategy does not include results for 1000 users because it proved incapable of handling such a load, as reflected in its request latency during the load of 500 users.

The observed increase in error RPS during the workload with 1000 users using the "NO JOIN" strategy can be attributed to the occurrence of too many HTTP requests for a single VM simultaneously. This resulted in the resetting of some active connections and subsequently led to a higher number of client-side errors.

It is important to clarify that the variance in total RPS across different strategies is due to the fact that simulated users cannot initiate a new action until the server has processed the previous one and the minimum turn duration has elapsed. If the server takes longer to handle a request than the minimum turn duration, then the RPS is constrained by the server's efficiency. This also implies that a total RPS of two thousand is close to the limit for the "RT JOIN" strategy.

VI. Final Service

The Reactions Storage has evolved into a fully-fledged service that fulfills all the requirements outlined in the Design section, offering a diverse set of features, integrations, and tools. Besides its capability to handle thousands of concurrent requests having latency within 100 milliseconds, it also contains a flexible reactions configuration with namespaces and settings such as $mutually\_exclusive\_reactions$, enabling it to meet various business demands.

Additionaly, the service offers fine-grained server configuration, allowing clients to tune middleware preferences, choose metrics to collect, and adjust pgx settings. Load testing with user-defined workloads is seamlessly supported by writing a corresponding configuration file and launching the simulation with a single command. Both the simulation and the Reactions Storage are integrated with essential observability tools like Prometheus and Grafana. Furthermore, the Reactions Storage includes comprehensive Grafana dashboards out-of-the-box, offering dozens of panels to observe the current service state.

Moreover, it provides a suite of tools for automating deployment routines across three different deployment types: local, docker-compose, and remote VMs. Developer tools are also included, offering various functionalities such as automated Flame Graph generation and management of Grafana dashboard backups. The service is also thoroughly covered by end-to-end and unit tests.

Finally, the Reactions Storage comes with comprehensive documentation featuring step-by-step tutorials and an API definition in the OpenAPI 3.0 [37] format.

VII. Lessons Learned

In this section we share useful lessons we learned from our work on Reactions Storage.

Fair Benchmarking Comes at a Cost

As we described in the Experiments section, we developed a simulation system to benchmark the Reactions Storage. While this approach was essential for us to fairly evaluate different strategies, the development of the system and all necessary automation tools (e.g., capturing metrics and aggregating results into a single JSON file) required a significant investment of time. Tasks such as writing client-side code, addressing race conditions, testing, and other essential activities consumed more time than the development of the Reactions Storage strategies altogether.

While we recognize the importance of thorough testing and benchmarking, we also understand that creating a benchmarking system with automation and necessary observability is more challenging than it may seem. In cases where applicable, choosing off-the-shelf solutions is often the best choice.

Load Simulation: A First-Class Bug Hunter

Writing tests is typically the initial and primary task for developers to ensure there are no bugs or mistakes in the code. However, identifying bugs only through tests can be challenging because tests only cover scenarios that developers anticipate. As the codebase grows larger than what a developer can hold in their memory, it is almost certain that tests will not cover all possible scenarios comprehensively. On the other hand, simulating real-world load not only ensures that the program can handle it but also helps detect bugs that are likely to occur after release.

Strategy	Latency (p95)			Avg RPS	
	GET	POST	DELETE	Total	Error
MVP	21ms	25ms	8ms	1089	0
RT JOIN	5ms	18ms	20ms	1246	0
NO JOIN	5ms	19ms	9ms	1251	0

(a) 100 users, 240 turns

Strategy	Latency (p95)			Avg RPS	
	GET	POST	DELETE	Total	Error
MVP	2156ms	2119ms	832ms	1208	0.6
RT JOIN	477ms	463ms	223ms	2246	0
NO JOIN	16ms	38ms	26ms	3062	0

(b) 500 users, 240 turns

Strategy	Latency (p95)			Avg RPS	
	GET	POST	DELETE	Total	Error
MVP	—-	—-	—-	—-	—-
RT JOIN	1154ms	1006ms	634ms	2025	0.03
NO JOIN	78ms	146ms	67ms	3731	0.28

(c) 1000 users, 240 turns

TABLE II: Experiment Results

For example, our simulation helped uncover a scenario where database locks were not released automatically after a transaction had been rolled back, eventually leading to a deadlock. As a solution, we had to utilize $pg\_advisory\_xact\_lock$ instead of $pg\_advisory\_lock$. It was not initially obvious to us to write tests involving aborted transactions and then inspect the database for dangling locks. Additionally, we discovered bugs related to namespace constraint checks that we had previously believed were thoroughly tested with end-to-end tests.

However, while simulation proves beneficial in bug hunting, it is not typically designed for this purpose, and investigating occurred errors can be challenging without a proper logging system.

Comprehensive Monitoring is Essential

Performance optimization is an ongoing process, and you never know which part of the system will become the next bottleneck until you address the current one [1]. Therefore, if any component of the system is not appropriately monitored, it may eventually become the bottleneck without any means to discover it. Moreover, modern systems comprise numerous components that influence overall performance, including hardware, the operating system and its configuration, external processes, and more. Thus, attempting to guess the current bottleneck is usually a waste of time.

VIII. Future Work

While the Reactions Storage Service has already met all the initial requirements, there are still areas for potential improvement.

Firstly, we are going to generalize constraints and allow clients to define their own. Currently, only two constraints are implemented: $max\_uniq\_reactions$ and $mutually\_exclusive\_reactions$. These constraints validate whether an ”add reaction” operation is permitted. If any constraint is violated, the operation is aborted and the reaction is not added. In the future, constraints will be generalized to any validators that accept information regarding the ”add reaction” operation (such as user, reaction counters, and the new reaction being added) and determine whether the constraint is met. This will enable clients to implement their own user-defined constraints by writing corresponding Golang code. There are numerous applications for user-defined constraints. For instance, clients could implement user-specific rules via constraints, such as allowing only particular users to use specific reactions. We are currently discussing the best approach to implement this feature and are considering using Golang plugins to isolate clients' code.

Additionally, the Reactions Storage service currently lacks support for migrations. If user reactions are already stored in another database, there should be a method to efficiently migrate these reactions to our storage. This includes the ability to add reactions in batches. Moreover, constraints should be ignored during migration to ensure no data, even if invalid, is lost. Therefore, new tools are necessary to help clients with migration challenges.

We are still considering sharding the PostgreSQL storage. Since a NO JOIN strategy was chosen for storing reactions, we can further improve performance by splitting the data across multiple database instances. However, the current workloads we observe do not necessitate additional performance enhancements. Therefore, we will implement sharding when it becomes necessary.

We are also considering adding RPC API support. The Reactions Storage service is intended to be used by other backend services, and the RPC approach is currently popular for microservice architecture. Unlike REST, which may struggle to describe all possible use cases via resources and methods (GET, POST, etc.), RPC can provide a more flexible and comprehensive solution.

Finally, there are numerous minor improvements that the Reactions Storage service can benefit from. We are going to add new HTTP handlers to extend its possible use cases. For example, there will be a handler to retrieve a list of users who applied a given reaction to a specific entity. Additionally, the configuration update process is currently risky because incompatible configurations can be applied. To address this, we will make the process safer by adding handlers for validating new configurations. We also noticed that reaction counters lack timestamps, which are useful for deterministically sorting reactions with the same counter by using the timestamp as a secondary sorting key.

IX. Conclusion

The Reactions Storage is a world-first, open-source, general-purpose system for storing and managing user reactions. The Reactions Storage embodies an efficient server with a user-friendly API capable of scaling to thousands of concurrent requests. And its simple yet flexible reactions configuration allows it to meet various business demands.

Additionally, this work fills an important gap in existing literature by providing a performance overview of different strategies for data aggregation. We showed that runtime aggregation performance can be a feasible option. However, avoiding data aggregation by storing necessary aggregation results (such as counters for reactions) offers the best performance yet limits the service's flexibility. Lastly, while asynchronous aggregation may be a suitable option for certain workloads, it is not ideal for every scenario, as such calculations often take significant time and result in stale data being returned to users.

The Reactions Storage has proven its efficiency and has evolved into a fully-fledged service, offering a diverse set of features, integrations, and tools. We will continue our efforts to further enhance the Reactions Storage and make it the best off-the-shelf solution for storing reactions.

Reactions Storage is open source and available at https://github.com/Deimvis/ReactionsStorage.

References

[1] E. Pirozzi, I. Ahmed, and G. Smith, "PostgreSQL 10 High Performance: Expert Techniques for Query Optimization, High Availability, and Efficient Database Maintenance," 3rd ed. Packt Publishing, Limited, 2018, pp. 508. ISBN: 978-1788474481.

[2] C. Adams, L. Alonso, B. Atkin, J. Banning, S. Bhola, R. Buskens, M. Chen, et al. 2020. Monarch: Google's planet-scale in-memory time series database. Proceedings of the VLDB Endowment 13, 12 (2020), 3181–3194

[3] D. E. Knuth, "Structured programming with GOTO statements," Computing Surveys, vol. 6, pp. 261-301, Dec. 1974

[4] A. A. Arman and A. Pahrul Sidik, "Measurement of Engagement Rate in Instagram (Case Study: Instagram Indonesian Government Ministry and Institutions)," 2019 International Conference on ICT for Smart Society (ICISS), Bandung, Indonesia, 2019

[5] T. W. Beng, L. T. Ming, "A Critical Review on Impression Rate and Pattern on Social Media Sites," International Conference on Digital Transformation and Applications (ICDXA) 2020

[6] V. Wadhwa, E. Latimer, K. Chatterjee, J. McCarty, d R.T. Fitzgerald, "Maximizing the tweet engagement rate in academia: analysis of the AJNR Twitter feed," in American Journal of Neuroradiology, AJNR Oct. 2017. [Online]. Available: https://www.ajnr.org/content/38/10/1866

[7] A. Torres, R. Galante, M. S. Pimenta, A. J. B. Martins, "Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design," in Information and Software Technology, vol. 82, pp. 1-18, February 2017.

[8] E. Sochopoulos, "Distributed computing in Go: comparative analysis of web application frameworks," University of Macedonia, Table 3, p. 78, Available: http://dspace.lib.uom.gr/handle/2159/24536.

[9] REpresentational State Transfer (REST). https://standards.rest/. Accessed: 2024-05-12

[10] Internet Engineering Task Force (IETF), "RPC: Remote Procedure Call Protocol Specification Version 2," RFC 5531, Apr. 2009. https://datatracker.ietf.org/doc/html/rfc5531. Accessed: 2024-05-12

[11] Internet Engineering Task Force (IETF), "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616, Jun. 1999. https://datatracker.ietf.org/doc/html/rfc2616. Accessed: 2024-05-12

[12] Yandex Cloud. https://yandex.cloud/en. Accessed: 2024-05-12

[13] Telegram. https://telegram.org/. Accessed: 2024-05-12

[14] Telegram API Documentation. https://core.telegram.org/api. Accessed: 2024-05-12

[15] LottieFiles. https://lottiefiles.com/. Accessed: 2024-05-12

[16] Discord. https://discord.com/. Accessed: 2024-05-12

[17] Discord API Documentation. https://discord.com/developers/docs/intro. Accessed: 2024-05-12

[18] LinkedIn. https://www.linkedin.com/. Accessed: 2024-05-12

[19] LinkedIn API Documentation. https://learn.microsoft.com/en-us/linkedin/. Accessed: 2024-05-12

[20] Tinkoff. https://tinkoff-group.com/. Accessed: 2024-05-12

[21] PostgreSQL. https://www.postgresql.org/. Accessed: 2024-05-12

[22] Nginx. https://www.nginx.com/. Accessed: 2024-05-12

[23] Golang. https://go.dev/. Accessed: 2024-05-12

[24] Gin. https://gin-gonic.com/. Accessed: 2024-05-12

[25] Cache gin's middleware. https://pkg.go.dev/github.com/gin-contrib/cache. Accessed: 2024-05-12

[26] Beego. https://pkg.go.dev/github.com/beego/beego. Accessed: 2024-05-12

[27] GoBuffalo. https://gobuffalo.io/. Accessed: 2024-05-12

[28] pgx. https://pkg.go.dev/github.com/jackc/pgx/v5. Accessed: 2024-05-12

[29] Zap. https://pkg.go.dev/go.uber.org/zap. Accessed: 2024-05-12

[30] TechEmpower. https://www.techempower.com/benchmarks/. Accessed: 2024-05-12

[31] Flame Graphs. https://www.brendangregg.com/flamegraphs.html. Accessed: 2024-05-12

[32] Docker. https://www.docker.com/. Accessed: 2024-05-12

[33] Prometheus. https://prometheus.io/. Accessed: 2024-05-12

[34] Grafana. https://grafana.com/. Accessed: 2024-05-12

[35] Pushgateway. https://github.com/prometheus/pushgateway. Accessed: 2024-05-12

[36] Node Exporter. https://github.com/prometheus/node_exporter. Accessed: 2024-05-12

[37] OpenAPI. https://www.openapis.org/. Accessed: 2024-05-12