Implementación de medallas sociales en jueces en línea

Trabajo de fin de grado
Doble Grado en Ingeniería Informática y Matemáticas
Facultad de informática
Universidad Complutense de Madrid
Curso 2016-2017

Realizado por
Eric García de Ceca Elejoste

Proyecto dirigido por
Marco Antonio Gómez Martín
Pedro Pablo Gómez Martín

Agradecimientos

A mi familia y amigos, por el apoyo que me han dado durante toda la carrera.

A mis compañeros de curso, con los que he compartido tantos momentos en estos años.

A los profesores de la facultad de informática, por el esfuerzo y dedicación que han puesto en su enseñanza y por todo lo que hemos aprendido de ellos.

A mis directores de proyecto, sin cuya paciencia, comprensión y consejos no habría sido posible este trabajo de fin de grado.

A todos ellos, gracias.

Eric.

Resumen

"Implementación de medallas sociales en jueces en línea" es un proyecto que tiene como objetivo la inclusión de diversas funcionalidades en el juez en línea de problemas de programación '¡Acepta el Reto!' para aumentar la interacción entre los usuarios, y fomentar la competitividad entre ellos y la autosuperación. La funcionalidad principal que queremos implementar es la inclusión de nuevos rankings. En concreto, queremos implementar rankings que ofrezcan datos actualizados en tiempo real, y que además puedan filtrarse por lenguaje. De manera secundaria, queremos extender la funcionalidad social con botones de 'Me gusta' y de 'Seguir'.

Durante el diseño, se ha prestado especial atención a la eficiencia tanto en tiempo como en memoria de las funcionalidades implementadas, de manera que sigan siendo válidas y rápidas a medida que la plataforma de '¡Acepta el Reto!' siga creciendo con nuevos usuarios y problemas. Para garantizar su efectividad, se han analizado los costes de las diversas implementaciones propuestas y se han realizado algunas pruebas de carga a posteriori usando datos de usuarios, problemas y envíos reales de la web.

Palabras clave

- Juez en línea
- Ejercicios de programación
- Desarrollo de framework
- Me gusta
- Seguidor
- Clasificación

Abstract

"Implementation of social badges for online judges" is a project that aims to include various functionalities into '¡Acepta el reto!', an online judge of programming exercises, in order to increase user competitiveness and self-improvement. The main functionality we want to implement is the inclusion of new rankings. In particular, we want to implement rankings that offer real-time data, which will be able to be filtered by language. Secondly, we want to extend social functionality with 'Like' and 'Follow' buttons.

During the design phase, special attention was paid to achieving a both time and memory-efficient implementations, so they remain valid and fast while '¡Acepta el reto!' platform grows, both in users and exercises. In order to guarantee its efficiency, the efficiency of the suggested implementations was analyzed, and some load tests have been made afterward using real user, problem and submission data from the platform.

Keywords

- Online judge
- Programming exercises
- Framework development
- Like
- Follower
- Ranking

Contenido

Agradecimientos ii

Resumen..... iii

Palabras clave..... iii

Abstract iv

Keywords..... iv

1. Introducción 1
   1.1. Motivación 1
   1.2. Objetivos 1
   1.3. Estructura de la memoria..... 2
1. Introduction 4
   1.1. Motivation..... 4
   1.2. Goals..... 4
   1.3. Memory structure 5
2. Estado del arte 7
   2.1. Jueces en línea 7
      2.1.1. UVa Online Judge 8
      2.1.2. Sphere Online Judge..... 9
      2.1.3. Jutge 10
   2.2. Rankings 11
   2.3. Botones 'Me gusta' 14
   2.4. Seguidores..... 15
3. Punto de partida..... 16
   3.1. Estructura de '¡Acepta el Reto!' 18
4. Primera aproximación de la plataforma a las redes sociales..... 21
   4.1. Implementación del sistema de 'Me gusta' 21
      4.1.1. Validación..... 22
   4.2. Implementación del sistema de seguidores..... 23
      4.2.1. Validación..... 23
5. Obtención de la posición en el ranking en tiempo real 25
   5.1. Árboles de Fenwick 26
   5.2. Aplicación en el proyecto..... 27
   5.3. Ordenación de envíos con el mismo tiempo de ejecución 28
6. Optimización de la clasificación 29

6.1. Diseño ... 30
6.1.1. Opción 1: Tablas Hash ... 30
6.1.2. Opción 2: Montículos ... 30
6.1.3. Opción 3: Vector ordenado por inserción ... 31
6.1.4. Opción 4: Vector + tabla Hash ... 32
6.1.5. Opción 5: Optimización de la base de datos ... 32
6.2. Implementación ... 33
7. Rankings por lenguaje ... 35
7.1. Clasificación por lenguaje ... 35
7.2. Puesto en tiempo real (y en el momento del envío) por lenguaje ... 36
8. Reevaluaciones ... 38
8.1. Ampliación del sistema de eventos ... 38
8.2. Ampliación en los árboles de Fenwick ... 38
8.3. Ampliación en las clasificaciones ... 39
8.4. Modificaciones en la corrección de envíos ... 39
9. Estudio de la eficiencia de los sistemas implementados ... 41
9.1. Estudio de la eficiencia de los árboles de Fenwick ... 41
9.1.1. Inserción de nuevos envíos ... 41
9.1.2. Corrección de envíos ... 42
9.1.3. Consulta de posiciones ... 42
9.1.4. Conclusiones ... 43
9.2. Estudio de la eficiencia del sistema de rankings ... 44
9.2.1. Inserción de nuevos envíos ... 44
9.2.2. Corrección de envíos ... 45
9.2.3. Consulta de posiciones ... 46
9.2.4. Conclusiones ... 46
10. Conclusiones ... 47
10.1. Valoración personal ... 47
10.2. Trabajo futuro ... 47
10. Conclusions ... 49
10.1. Personal assessment ... 49
10.2. Future work ... 49
Bibliografía ... 51

1. Introducción

'¡Acepta el reto!' es un juez online, es decir, un repositorio de ejercicios de programación con capacidad de compilar y ejecutar la solución propuesta por los usuarios a los ejercicios. Actualmente, los 20 usuarios con los envíos más rápidos se muestran en una clasificación, y los usuarios conocen la posición que obtuvo su solución en el momento del envío. Sin embargo, esta información es escasa y, en el segundo caso, no se actualiza con el tiempo, por lo que se queda obsoleta. Además, la plataforma no cuenta con interacción social entre los usuarios, y el aumento de interacción entre ellos ha sido demandado por los propios usuarios para conseguir un ambiente más social en la plataforma. Sin embargo, esto último actualmente es una carencia menos urgente que la de los rankings.

De entre las opciones para aumentar las posibilidades de interacción de los usuarios, se desea dar soporte a botones 'Me gusta', en principio para problemas, aunque ampliable en el futuro a otras entidades de la plataforma. También se desea dar la posibilidad a los usuarios de seguir a otros usuarios, pudiendo recibir información sobre su actividad en la plataforma.

Sin embargo, este no será el tema principal del proyecto, sino que lo serán los rankings. Para fomentar la competitividad entre los usuarios, se desea ampliar y optimizar el sistema de clasificaciones, pudiendo proporcionar datos en tiempo real sobre la clasificación de los envíos. También se desea poder ofrecer clasificaciones más específicas, pudiendo filtrar los datos por lenguaje de programación utilizado.

1.1. Motivación

Se ha intentado aplicar los conocimientos aprendidos en la carrera, respetando y aplicando las metodologías y patrones de diseño, así como todos los conceptos de programación aprendidos, de modo que además de intentar realizar un código operativo, además éste sea claro, escalable, mantenible y eficiente. Dichas propiedades son fundamentales a la hora de conseguir que un código pueda perdurar en el tiempo y ser corregido, modificado y ampliado con un coste en tiempo y esfuerzo relativamente bajo.

Algunos de los conceptos de la ingeniería del software que se han usado para conseguirlo son la cohesión, el acoplamiento, o la escalabilidad. Además, se han usado patrones como los DAOs (Data Access Objects), que encapsulan el acceso a la base de datos, desacoplando su implementación de las funcionalidades que ofrece a las otras capas de la aplicación. Diversas estructuras de datos comunes (como vectores, matrices y listas) y otros menos comunes (como los árboles de Fenwick) se han usado, junto con diversos algoritmos, para garantizar un manejo de los datos eficiente en cada uno de los contextos y requisitos.

1.2. Objetivos

El objetivo de este trabajo ha sido desarrollar un framework que aumente las posibilidades de interacción entre los usuarios de la plataforma '¡Acepta el reto!'. Dicho framework se integrará en el futuro en el backend del servidor, ampliando su funcionalidad y mejorando su eficiencia en algunos puntos que empiezan a ser críticos en la plataforma dada la cantidad de usuarios, problemas y envíos que maneja en la actualidad.

El framework será transparente a los usuarios y administradores, intentando minimizar los recursos en tiempo y memoria que utiliza, de manera que su inclusión en el sistema real sea factible.

También es indispensable que el framework sea modular, mantenible y escalable, ya que la plataforma '¡Acepta el reto!' crece y evoluciona con el tiempo con correcciones, ampliaciones y actualizaciones. Mantener estas tres propiedades en la medida de lo posible es fundamental para que dichos cambios que puedan venir en el futuro tengan requieran un esfuerzo relativamente bajo para su implementación e integración.

Se considerará además condición necesaria la corrección de los sistemas implementados, demostrando mediante pruebas empíricas que el funcionamiento es correcto y similar en cada una de las simulaciones. Además, se hará hincapié en la eficiencia de los rankings, comparando la implementación propuesta con la existente en la plataforma.

El framework quedará integrado en el backend, dejando como trabajo futuro la implementación del frontend y su inclusión en el portal online.

1.3. Estructura de la memoria

En los capítulos siguientes podremos obtener información sobre el proyecto. La estructura contará con:

- Un capítulo titulado ***Estado del arte*** donde podremos obtener los antecedentes de los jueces online, los rankings, los botones de 'Me gusta' y los seguidores.
- Un capítulo titulado ***Punto de partida*** donde relataremos brevemente el estado actual de la plataforma '¡Acepta el reto!' sobre la que vamos a trabajar.
- A continuación, tendremos varios capítulos abarcando los distintos problemas que abarca el proyecto:
    - En el capítulo ***Primera aproximación de la plataforma a las redes sociales*** estudiaremos e implementaremos los sistemas de 'Me gusta' y seguidores.
    - En el capítulo ***Obtención de la posición en el ranking en tiempo real***, abarcaremos la problemática de obtener la posición que ocupa un envío entre todos los realizados al problema en tiempo real, lo que actualmente es inviable en la plataforma.
    - En el capítulo ***Optimización de la clasificación*** estudiaremos qué estructuras de datos podemos implementar para que el mantenimiento de los rankings sea eficiente y viable a largo plazo, eliminando uno de los puntos críticos que la plataforma tiene en la actualidad.
    - En el capítulo ***Rankings por lenguaje*** estudiaremos qué opciones tenemos para poder crear rankings filtrados por un lenguaje concreto y cómo esta implementación afecta al funcionamiento de la aplicación y las estructuras creadas en capítulos anteriores.
    - En el capítulo ***Reevaluaciones***, estudiaremos cómo este evento que suele ocurrir de manera esporádica afecta al funcionamiento de la plataforma y a las

estructuras creadas en capítulos anteriores, y como estos pueden ser adaptados para gestionar las reevaluaciones de manera correcta y autónoma.

- Posteriormente, en el capítulo ***Estudio de la eficiencia de los sistemas implementados*** analizaremos el funcionamiento, la corrección y la eficiencia de los sistemas resultado de los capítulos anteriores.
- Finalmente terminaremos con un capítulo ***Conclusiones***, en el cual se analizará el resultado del trabajo así como las proposiciones a futuro que han ido surgiendo durante la realización del mismo.

1. Introduction

'¡Acepta el reto!' is an online judge, in other words, a programming exercises repository able to compile and execute user summited solutions to those exercises. Nowadays, the 20 users with the fastest submissions are displayed in a ranking, and users know the position in the ranking they achieved in the moment of summiting the solution. However, this information provided is very limited and, in the second scenario, it doesn't update with time, so it becomes out-dated. Moreover, the platform doesn't implement any user interaction facility, and more user interaction has been demanded to achieve a more social environment inside the platform. Nevertheless, the lack of user interaction is less urgent than the necessity of better and more efficient rankings.

Among other options to increase the possibilities of user interaction, we want to give support to Like buttons for exercises, and probably for other entities in the platform in the future. Also, we want to give users the possibility to follow each other in order to receive information about their activity on the platform.

However, this won't be this project's main issue, but rankings will be. In order to encourage user competitiveness, we want to extend and optimize rankings, so users can receive real-time feedback about their position in the ranking. We also want to offer more specific rankings, so users can filter data by programming language.

1.1. Motivation

Efforts have been made apply all the knowledge learned during my studies, respecting and applying all the methods and design patterns, as well as all the programming concepts learned in order to achieve, not only a code that is functional, but also clear, scalable, maintainable and efficient. Those properties are crucial in order to obtain a code that can last throughout time and be corrected, modified and extended with a relatively low cost of time and effort.

Some Software Engineering concepts that have been applied to reach this goal are cohesion, coupling or scalability. Also, design patterns like Data Access Objects have been used. Data Access Objects encapsulate database access, uncoupling its implementation from the functionalities provided to other layers of the application. Various common data structures (like vectors, matrixes, and lists) and some less common ones (like Fenwick trees) have been used among various algorithms to guarantee an efficient data management in each context and requirement.

1.2. Goals

This project main goal is to develop a framework that increases interaction possibilities among '¡Acepta el reto!' platform users. This framework will be integrated into the server's backend in the future, extending its functionality and improving its efficiency in some points that nowadays are becoming critical in the platform due to the number of users, problems, and submissions that it manages nowadays.

The framework will be transparent to users and administrators, trying to minimize time and memory resources used, so its integration in the real system is possible.

The framework must be modular maintainable and scalable as an essential part of it, since '¡Acepta el reto!' platform grows and evolves through time due to corrections, extensions, and updates. Maintaining those three properties to the extent possible is fundamental, so future changes require a relatively low effort for their implementation and integration.

Correction of implemented systems will be considered a necessary condition, proving through empirical tests that it functions correctly and similarly in each one of the simulations. Also, special attention will be paid to rankings efficiency, comparing the proposed implementation with the already existing one.

This framework will be integrated into the platform's backend, leaving the frontend implementation and its inclusion into the platform for future work.

1.3. Memory structure

In the following chapters we can obtain information about the project. The structure will include:

- A chapter called ***State of Art*** where we can obtain the predecessors of online judges, rankings, Like buttons and followers.
- A chapter called ***Starting Point***, where we will briefly explain the current state of the platform '¡Acepta el reto!' we will work upon.
- Then we will narrate in a few chapters all the diverse problems this project take on:
    - In a chapter called ***First Approach of the Platform to Social Networks*** we will analyze and implement like and follow systems.
    - In a chapter called ***Obtaining Real-Time Position in the Ranking*** we will take on the problem of obtaining the position of a submission among all submissions made to a problem in real time, which nowadays is impracticable in the platform.
    - In a chapter called ***Ranking Optimization*** we will study which data structures we can implement in order to make ranking maintenance efficient and viable in the long term, erasing one of the critical points of the platform.
    - In a chapter called ***Rankings by Language*** we will analyze which options we have in order to make rankings filtered by a specific language and how this implementation affect the rest of the application and all the structures implemented in previous chapters.
    - In a chapter called ***Reevaluations***, we will analyze how this fairly frequent event affects the normal functioning of the platform and all the structures implemented in previous chapters, and how those can be adapted to manage reevaluations correctly and autonomously.
- Then, in a chapter called ***Study of the Efficiency of the implemented systems***, we will analyze the functioning, correction and efficiency of the systems implemented in former chapters.

- Finally we will conclude this document with a chapter called ***Conclusions***, in which the final result of the project will be analyzed, as well as future proposals that have emerged during the implementation.

2. Estado del arte

2.1. Jueces en línea

Los jueces en línea son sitios web que contienen una serie de problemas de programación. A diferencia de un repositorio de problemas, los jueces en línea son capaces de recibir un código fuente, compilarlo y ejecutarlo con varios casos de prueba (algunos públicos y otros ocultos) con cuyas salidas se verifica la corrección del algoritmo. En caso de que la ejecución haya sido correcta, se extraen diversos parámetros de la ejecución, como el tiempo y la memoria consumidos. Con estos datos, se proporciona al usuario el veredicto sobre la ejecución, proporcionando, en caso de ejecución correcta, los datos adicionales que el juez haya calculado y considere relevantes.

El equivalente "no digital" de este sistema sería el profesor o tutor que propone ejercicios al alumno y posteriormente devuelve feedback sobre su corrección. Los jueces en línea permiten realizar esta tarea de manera masiva a través de Internet.

Respecto a sus características, los jueces, aunque son numerosos y variados, suelen tener algunas características comunes:

- Entre los lenguajes soportados suelen estar C++ y Java por ser los más comunes. Sin embargo, muchos jueces soportan decenas de lenguajes.

- Los problemas pueden escribirse en un único fichero de código, por lo que suelen ser bastante específicos y poco extensos. Esto hace que no sean representativos de la realidad de trabajar en proyectos grandes y/o en equipo.

- Sus problemas generalmente están enfocados a aprender las bases de la programación o algoritmos concretos, por lo que suelen utilizar entrada y salida por consola.

- Algunos problemas pueden requerir conocimientos avanzados de matemáticas o algoritmia. Sin embargo abundan los problemas que abarcan conceptos mucho más básicos.

- Suelen tener restricciones respecto al código enviado. Generalmente está prohibido el uso de llamadas al sistema y de hilos.

Dicho esto, cada juez es distinto y ofrece diferentes posibilidades. Los hay sencillos, que únicamente permiten realizar problemas, corregirlos y (posiblemente) añadir los resultados a un ranking. Otros sin embargo permiten una interacción más variada donde los usuarios pueden configurar su perfil, relacionarse, obtener logros o consultar diversos rankings y estadísticas.

Algunos jueces han albergado concursos de programación, ya sea convocados por la propia plataforma del juez o por otras entidades que contratan la plataforma o colaboran con ella para realizar el concurso. Sin embargo, la finalidad principal de estos es el autoaprendizaje, ya sea para personas que empiezan en la programación o que quieren reforzar sus conocimientos en algoritmia o estructuras de datos.

Veamos algunos ejemplos de jueces:

2.1.1. UVa Online Judge

UVa Online Judge [1] es un juez en línea de problemas de programación gratuito desarrollado en la Universidad de Valladolid [2]. Este juez es pionero al llevar funcionando desde abril de 1997 y contener aproximadamente 4900 problemas, categorizados de manera variada y estructurada. La mayoría de los problemas son de concursos, ya que UVa Online Judge aloja problemas de más de 375 concursos, algunos de los cuales actuó como juez. Admite soluciones en ANSI C, C++, C++11, Java, Pascal y Python 3. Dispone de estadísticas para cada uno de los problemas individuales, y también sobre usuarios los individuales de manera pública.

2.1.2. Sphere Online Judge

Sphere online judge

PROBLEMS STATUS RANKS DISCUSS CONTESTS sign in

Become a true programming master

Learn how to code and build efficient algorithms

19502590 submissions, 513393 registered users, 6280 public problems

Sign up & Start coding!

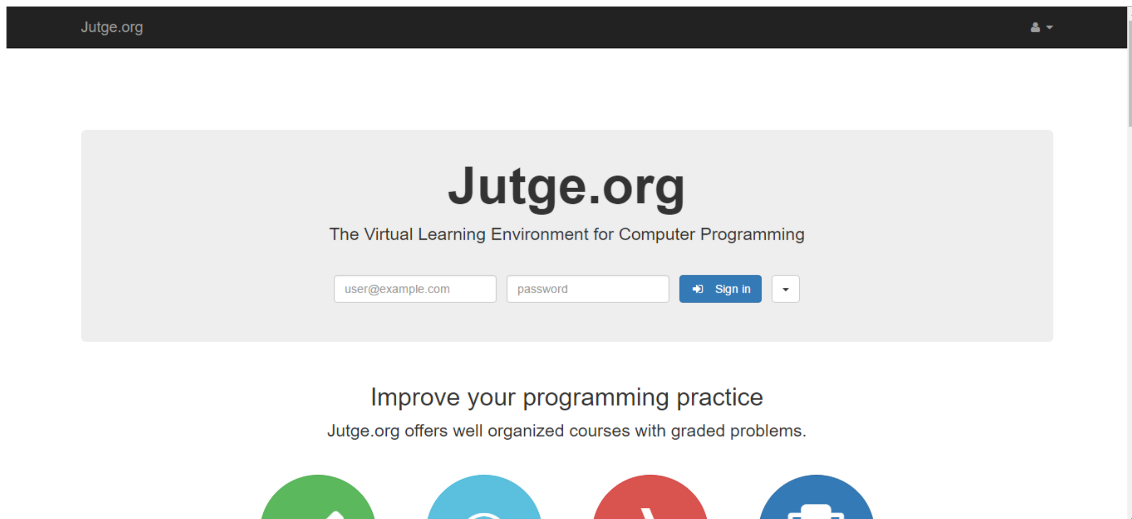
Topic	Users	R	V	A
WA on To and Fro	I	1	1	4m
Need more test cases for PALIN problem	S A	2	15	8h
NHAY Needle in haystack	A	1	14	2d
Charaacter patterns (Act 2)	H	1	19	3d
Life,Universe and Everything	G H	2	42	4d

El juez online Sphere Online Judge [3] admite más de 45 lenguajes de programación incluyendo C, C++, C#, Pascal, Java, Python y Ruby entre otros. Actualmente cuenta con un repertorio de problemas muy extenso, con aproximadamente 13.000 problemas, cuyos enunciados están disponibles en una gran variedad de idiomas. Este repertorio es tan extenso gracias a que los usuarios de la plataforma pueden convertirse en creadores de problemas y enviar propuestas de nuevos ejercicios. Debido a que los usuarios también pueden proponer y crear concursos, este juez ha albergado más de 2400 concursos de muy variada índole desde 2012, albergando desde desafíos fugaces hasta cursos de aprendizaje online a largo plazo.

Sus problemas están separados por en cuatro categorías, dependiendo de si corrigen los problemas según su resultado es correcto o no, o le otorgue una puntuación en función de algún criterio. Esta categorización también depende de si los problemas están específicamente pensados para ser didácticos o no. Los problemas también están categorizados por etiquetas, lo que permite encontrar problemas muy específicos dentro de un ámbito.

La estética de la página es muy sencilla, limpia e intuitiva a diferencia de otros jueces. Además, fomenta en gran medida la parte social, ya que tiene implementados foros y comentarios en los problemas, lo que da lugar a cooperación entre los usuarios. La página también ofrece mucha información sobre los creadores de la plataforma y sobre los numerosos creadores de problemas.

2.1.3. Jutge



Jutge.org [4] es un juez en línea de problemas de programación gratuito y con propósito educativo desarrollado en la Universitat Politècnica de Catalunya [5]. En la actualidad cuenta con más de 2200 problemas, los cuales se pueden resolver hasta en 20 lenguajes distintos, entre los que se encuentran C++, Java, Haskell y Python. Los enunciados de los problemas se encuentran en catalán, inglés y español principalmente, aunque no todos los enunciados se encuentran en todos los idiomas.

Además de ser una plataforma de aprendizaje para estudiantes, también da soporte a profesores para incorporarlo en su método de enseñanza. Para el personal docente, incluye algunas posibilidades como realizar listas de ejercicios para sus estudiantes, obtener estadísticas sobre sus resultados y crear sus propios problemas, además de reutilizar los ya existentes en la plataforma. La plataforma permite además realizar exámenes y concursos, siendo utilizada para la Olimpiada Informática Española [6].

2.2. Rankings

Una clasificación o ranking [4] es una relación de preorden total ($\lesssim$) [8] entre un conjunto de elementos respecto a uno o varios criterios. Una relación de preorden total cumple las siguientes propiedades:

- Transitividad: $\forall x, y, z,\ \ x \lesssim y\ \wedge y \lesssim z \Rightarrow x\ \lesssim z$
- Totalidad: $\forall x, y,\ \ x \lesssim y\ \vee y \lesssim x$
- Reflexividad: $\forall x,\ \ x \lesssim x$. Es consecuencia de la propiedad de totalidad.

Toda relación de preorden total tiene una relación de equivalencia asociada ($\sim$) [9], definida como $x{\sim}y\ \Leftrightarrow\ x \lesssim y\ \wedge\ y \lesssim x$. Una relación de equivalencia cumple las tres propiedades siguientes:

- Transitividad: $\forall x, y, z,\ \ x{\sim}y\ \wedge y{\sim}z \Rightarrow x{\sim}z$
- Simetría: $\forall x, y,\ \ x{\sim}y\ \Rightarrow y{\sim}x$
- Reflexividad: $\forall x,\ \ x{\sim}x$

El orden parcial entre las clases de equivalencia de un preorden total es un orden total estricto ($<$) [10], el cual cumple las siguientes propiedades:

- Transitividad: $\forall x, y, z,\ \ x < y\ \wedge y < z \Rightarrow x < z$
- Asimetría: $\forall x, y,\ \ x < y\ \Rightarrow y \nless x$
- Irreflexividad: $\forall x,\ \ x \nless x$. Es consecuencia de la asimetría.
- Tricotomía: $\forall x, y,\ \ x < y\ \vee\ y < x\ \vee x = y$, siendo exactamente uno de los tres cierto.

Tras esta ordenación, se asigna a cada elemento un número natural comenzando desde el 1 y de manera creciente. Dicho crecimiento no tiene por qué ser estricto (varios elementos pueden compartir el mismo número) ni exhaustivo (pueden existir números no asignados a ningún elemento). Cuando existen varios elementos equivalentes, pueden seguirse distintas estrategias o criterios para la asignación de naturales a los elementos, resultando en distintos tipos de rankings. Además de los nombres estándar, para mayor claridad se suele usar un nombre alternativo, que resulta de obtener el ranking de 4 elementos A, B, C, D donde $A < B{\sim}C < D$:

- **Ranking de competición estándar** (o ranking “1224”): los elementos que son considerados equivalentes obtienen la misma clasificación, saltando **después** tantos naturales como el número de elementos equivalentes menos uno. En este tipo de ranking, la posición de un elemento también se puede obtener como uno más el número de elementos **estrictamente menores** que él.

- **Ranking de competición modificado** (o ranking “1334”): los elementos que son considerados equivalentes obtienen la misma clasificación, saltando **antes** tantos naturales como el número de elementos equivalentes menos uno. En este tipo de ranking, la posición de un elemento también se puede obtener como uno más el número de elementos **menores o iguales** que él.

- **Ranking denso** (o ranking “1223”): los elementos que son considerados equivalentes obtienen la misma clasificación, continuando con el siguiente número natural para el

siguiente elemento no equivalente, sin dejar huecos en los naturales. En este tipo de ranking, la posición de un elemento también se puede obtener como uno más el número de **clases de equivalencia** cuyos elementos son **estrictamente menores** que él.

- **Ranking ordinal** (o ranking “1234”): todos los elementos reciben clasificaciones distintas recorriendo todos los números naturales sin saltar ninguno. El orden entre los elementos equivalentes se puede realizar de manera aleatoria o arbitraria, pero generalmente se suele usar un criterio de ordenación secundario que sea un orden estricto total (al menos en la práctica) como una ordenación alfabética o temporal.

- **Ranking fraccional** (o ranking “1 2.5 2.5 4”): Es un punto medio entre los rankings de competición estándar y modificado, y es el único que en vez de números naturales utiliza números fraccionales. Los elementos que son considerados equivalentes obtienen la misma clasificación, que es la media de las clasificaciones que obtendrían en el ranking ordinal. En este tipo de ranking, la posición de un elemento también se puede obtener como uno más el número de elementos **estrictamente menores** que él más la mitad del número de elementos **iguales** que él.

Las clasificaciones o rankings se usan en muchos contextos, siendo muy populares en contextos lúdico-competitivos como los torneos deportivos. La clasificación que obtienen los competidores puede realizarse de maneras diversas:

- **Calificación individual**: cada competidor realiza una prueba independiente de la que se obtiene una puntuación. Esta puntuación puede ser una marca alcanzada o una puntuación otorgada por un árbitro o juez. Este sistema no requiere que los participantes compitan en el mismo lugar o en el mismo momento, pudiendo resultar en competiciones de duración indefinida y abiertas a un público muy amplio, especialmente si se realiza por Internet. En los torneos de duración indefinida, se puede permitir a los competidores participar varias veces para que intenten superar su marca y ascender en la clasificación.

- **Carrera**: todos los competidores se enfrentan simultáneamente sobre la pista, percibiendo cada uno su posición en tiempo real y pudiendo ajustar su esfuerzo en función de la situación de sus competidores. En las carreras, se puede observar a simple vista quien es el ganador: el competidor con menos tiempo es el primero en alcanzar la línea de meta.

- **Título**: un competidor tiene que desafiar al actual campeón para ganar el campeonato. Cualquier competidor puede desafiar al campeón en cualquier momento.

- **Partidos a dos**: los competidores, ya sean personas individuales o equipos, compiten dos a dos, resultando en cada partido un ganador y un perdedor, habiendo posibilidad de epate o no según la competición. Dependiendo de cómo se obtenga la clasificación final a partir de los partidos individuales, existen varios subtipos que pueden mezclarse:

  - **Liguilla**: los competidores se enfrentan a cada uno de los demás al menos en una ocasión, sumando puntos en sus victorias. La clasificación se obtiene posteriormente con estos puntos.

- **Sistema suizo**: Se fija a priori un número de rondas y los competidores se enfrentan a otros competidores de igual puntuación en la misma ronda. Según avanzan las rondas, se produce un filtrado cada vez más fino de los jugadores hasta llegar a una clasificación final.

- **Eliminatorias**: es una variante del sistema suizo. Los competidores se enfrentan por parejas durante una serie de rondas, y tras cada ronda los perdedores abandonan la competición. Se realizan las rondas necesarias para hasta eliminar a todos los competidores menos uno.

2.3. Botones 'Me gusta'

Los Botones de 'Me gusta' [11] son una funcionalidad dentro del software de comunicación donde los usuarios pueden expresar que les gusta un contenido o que lo apoyan. Se encuentra en el nivel más bajo de las posibilidades de feedback. Generalmente, los servicios en Internet que contienen una opción de 'Me gusta' suelen mostrar el número de usuarios que han dado 'Me gusta' al contenido, añadiendo a veces una lista parcial de los usuarios concretos. Esta alternativa da una funcionalidad más limitada pero más rápida y resumida que otras formas de interacción como los comentarios.

Las variaciones sobre este sistema son numerosas. Algunos sitios permiten la opción de 'No me gusta' para votar en contra. Otros sitios permiten votaciones más complejas, permitiendo votaciones dentro de un rango de puntuaciones, mostrando generalmente la media de todas las votaciones realizadas.

La primera implementación de 'Me gusta' fue en FriendFeed [12] en 2007, un agregador en tiempo real de medios, redes sociales, blogs y en general cualquier web o plataforma que contara con actualizaciones RSS o Atom. Sin embargo, la implementación más famosa y reconocible es la de Facebook [13] [14]. Facebook incorporó los botones 'Me gusta' en febrero de 2009. En febrero de 2016 se extendió su funcionalidad, añadiendo otras reacciones como 'Me encanta', 'Me entristece' o 'Me enfada' entre otras.

En Facebook, se puede dar 'Me gusta' a multitud de contenidos: actualizaciones de estados, comentarios, fotos, publicaciones compartidas por amigos y anuncios.

Varias redes sociales, entre ellas Facebook, desarrollaron plug-ins que podían ser añadidos a cualquier página web para añadir un botón 'Me gusta' ligado a la cuenta de Facebook del visitante.

2.4. Seguidores

Los botones de ‘Seguir’ son una funcionalidad dentro del software de comunicación donde los usuarios pueden recibir actualizaciones de contenido de una o varias páginas de contenido, ya sea dentro de la misma plataforma o en plataformas distintas. Las actualizaciones pueden ser en tiempo real o periódicas, y pueden recibirse en la propia plataforma o a través de email.

Sus primeros antecedentes son las fuentes web (o web feeds), a través de los cuales los usuarios pueden suscribirse a una página o plataforma de Internet, y recibir las actualizaciones a través de agregadores. El contenido de las fuentes webs suele ser HTML o links a páginas o medios digitales.

Dos de los ejemplos más comunes de esta funcionalidad son Twitter [15] y Facebook. Twitter es una red social de microblogging, donde las publicaciones tienen un máximo de 140 caracteres. Los usuarios poseen un perfil en el que publicar sus posts, que aparecen en orden cronológico inverso (primero los más recientes). Los usuarios también pueden seguir a otros usuarios. La funcionalidad principal de Twitter es el ‘timeline’, el cual es un feed en el que aparecen todas las publicaciones de los usuarios a los que el usuario sigue, originalmente en orden cronológico inverso. El ‘timeline’ permite recibir las actualizaciones de los usuarios en tiempo real, aunque es necesario “refrescar” el ‘timeline’ de manera activa.

Facebook también incorpora en su página principal un feed con todas las interacciones que han realizado los amigos o las páginas a las que sigues, originalmente también en orden cronológico inverso. Actualmente, tanto Facebook como Twitter usan algoritmos de inteligencia artificial para ordenar los posts de sus ‘timelines’ según lo relevantes que sean para el usuario.

3. Punto de partida

El portal de internet '¡Acepta el reto!'[16] es un almacén y juez en línea de problemas de programación que acepta soluciones en C, C++ y Java. El proyecto nació como iniciativa de dos profesores de la Facultad de Informática de la Universidad Complutense de Madrid. Tras haber contribuido a poner el marcha ProgramaMe [17] y desarrollar problemas similares para realizar evaluación continua a sus alumnos, decidieron poner en marcha el desarrollo de un juez online similar a otros existentes (el más conocido el de la universidad de Valladolid [1]) pero centrado en problemas en español para alumnos de Ciclos Formativos y primeros años de universidad. De esta manera, alumnos de la Facultad de Informática de años posteriores o cualquier usuario que quisiera practicar la programación podría aprovechar estos problemas para su aprendizaje.

En el curso académico 2013-2014 comenzaron la implementación del backend, y delegaron la implementación del frontend a estudiantes, los cuales desarrollaron dos versiones preliminares del sitio. Ambas sirvieron como prueba de concepto y permitieron poner a prueba la infraestructura. Finalmente el 17 febrero de 2014, se pone en marcha '¡Acepta el reto!' con el primer envío a la misma. Desde entonces, el número de envíos a la página ha ido creciendo de manera exponencial, contando con 1000 envíos el 6 marzo de 2014, 30.000 el 16 marzo de 2015, 65.536 (2^{16}) envíos el 15 de marzo de 2016 y 131.072 (2^{17}) el 5 de abril de 2017. Además, el 21 de diciembre de 2016, se organizó el primer concurso online utilizando '¡Acepta el reto!' como juez [18] [19].

Se puede interactuar con la web, leer los problemas, consultar algunas estadísticas y leer información sobre '¡Acepta el reto!' y su funcionamiento, pero para subir soluciones a problemas y que sean evaluadas es necesario estar registrado.

El proyecto "Implementación de medallas sociales en jueces en línea" forma parte de una serie de mejoras y ampliaciones que quieren incluirse en '¡Acepta el reto!' con la finalidad de potenciar la parte social del sitio web. Durante el periodo de vida de la plataforma, las posibilidades de interacción entre los usuarios han sido relegadas a un segundo plano, ya que ampliar y mejorar el juez y el funcionamiento de la página ha sido más prioritario. En la actualidad, se permite a los usuarios algunas opciones de personalización, como introducir

algunos datos personales o subir una foto, pero la única interacción posible entre ellos es la competición a través de los rankings.

Actualmente, en mayo de 2017, el portal cuenta con más de 300 problemas, más de 6000 usuarios registrados y más de 138.000 envíos. La plataforma se encuentra en un estado de funcionamiento óptimo, con una cantidad de envíos diarios alta. Los problemas propuestos son aplicaciones de consola que reciben unos datos por la entrada estándar (generalmente en formato numérico) e imprimen el resultado por la salida estándar (generalmente también numéricamente, aunque también en texto simple). Los problemas están agrupados en volúmenes de 100, aunque los problemas de un volumen no tienen por qué estar relacionados entre sí más allá de la proximidad en su fecha de publicación. También se encuentran clasificados en categorías a través de diversos criterios. Sin embargo, la clasificación en cada criterio no tiene por qué ser exhaustiva (un problema puede no entrar en ninguna categoría) ni excluyente (un problema puede entrar en varias categorías). Además, la categorización puede constar de varios niveles, permitiendo subcategorías y anidamiento de las mismas.

La solución propuesta por un usuario a un ejercicio sólo es visible para el propio usuario, ya que el código se considera privado. Los envíos realizados por los usuarios son almacenados en una cola para posteriormente ser evaluados. El juez comprueba su corrección ejecutando el código usando como entrada unos casos de prueba ocultos y comparando la salida con la esperada. Los veredictos posibles, que pueden ser consultados en la página [20], son los siguientes:

- Aceptado (AC): La solución enviada ha superado todos los casos de prueba dentro de los límites de tiempo y memoria, coincidiendo la salida obtenida con la esperada.

- Error de presentación (PE): La solución es correcta, pero entre la salida obtenida y la esperada existen diferencias de formato en el uso de espacios, tabuladores y saltos de línea.

- Respuesta incorrecta (WA): El programa se ha ejecutado completamente, pero no ha superado los casos de prueba al existir diferencias entre los resultados esperados y obtenidos.

- Error de Compilación (CE): El juez no ha sido capaz de compilar el código debido generalmente a errores de sintaxis, aunque también es posible que se deba al uso de librerías no estándar.

- Error en ejecución (RTE): El código se ha podido compilar, pero la ejecución ha terminado de manera abrupta antes de procesar todos los casos. Las causas de la finalización abrupta pueden ser muy variadas, aunque algunas como las excepciones, los accesos a punteros inválidos o los accesos a arrays fuera del rango suelen ser comunes.

- Límite de tiempo superado (TLE): El juez ha sido capaz de compilar y ejecutar el código, pero ha tardado más tiempo del permitido en dar una respuesta, por lo que se ha cancelado su ejecución.

- Límite de memoria superado (MLE): El juez ha sido capaz de compilar y ejecutar el código, pero ha usado más memoria de la permitida, por lo que se ha cancelado su ejecución.

- Límite de salida superado (OLE): El juez ha sido capaz de compilar y ejecutar el código, pero ha escrito más datos de salida de los permitidos, por lo que se ha cancelado su ejecución.

- Función restringida (RF): El juez ha interrumpido la ejecución al encontrar alguna operación inadecuada o peligrosa para el servidor.

- En cola (IQ): El juez aún no ha evaluado el envío.

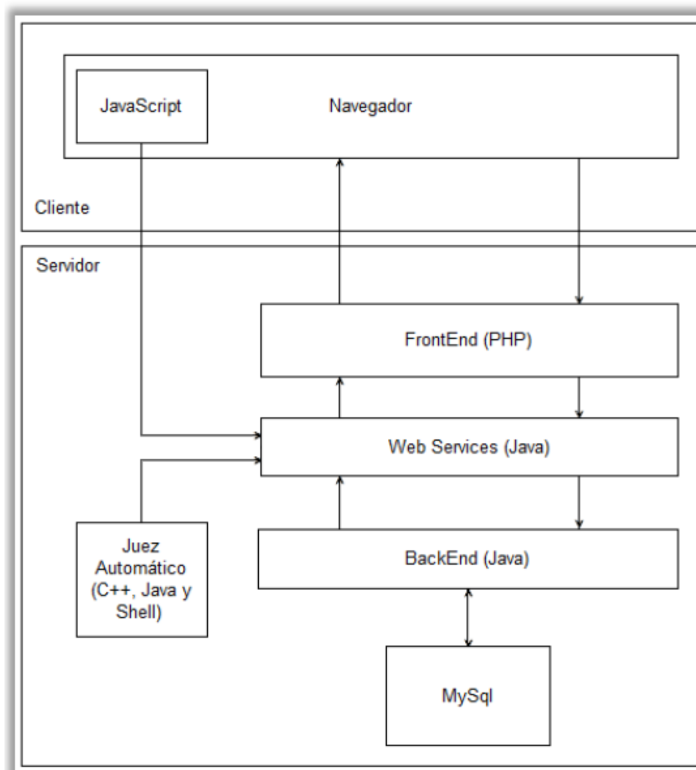
- Error interno (IE): El juez ha sufrido algún problema ajeno al envío o está en mantenimiento, por lo que se deberá volver a realizar el envío más tarde.

A pesar de que en principio sólo es necesario evaluar los problemas una vez, situaciones excepcionales pueden requerir que un problema tenga que volver a ser evaluado. Es lo que llamamos "reevaluaciones". Una reevaluación suele estar causada por un error en el servidor en el momento de la corrección, o por un error en el enunciado o en los casos de prueba, que invalida todas las correcciones del problema hechas hasta el momento. Dado que las reevaluaciones son sucesos inevitables, aunque ocurren con poca frecuencia, será algo que tendremos que tener en cuenta durante el desarrollo del proyecto.

3.1. Estructura de '¡Acepta el Reto!'

'¡Acepta el reto!' está dividió en varias capas. La parte por la que accede el usuario cuando se conecta con su navegador es el frontend. Dicho frontend está implementado en PHP [21] y HTML [22] con JavaScript [23]. De esta manera, HTML y JavaScript actúan de intermediarios entre la lógica de la aplicación y el usuario, proporcionándole una interfaz gráfica intuitiva con la que comunicarse con la plataforma y, más concretamente, con los servicios web. Estos servicios web, a su vez, actúan de intermediarios entre el frontend y el backend.

En el backend, se encuentran tanto la lógica de la aplicación como la base de datos permanente. La base de datos es una base de datos MySQL [24] gestionada gracias a Hibernate [25]. Sobre ello se encuentran los DAOs implementados en Java [26], además de cierta lógica de la aplicación implementada en el mismo lenguaje. Los elementos anteriores se encuentran integrados en un sistema Apache Tomcat [27] [28], el cual soporta gran parte de la carga de trabajo, ya que mantiene el sitio web y el juez en funcionamiento a través de una serie de demonios que se encuentran activos permanentemente.



El alcance de este proyecto es integrarse en el backend, optimizando y ampliando el sistema de rankings. El único ranking existente en la aplicación, los 20 usuarios con envíos más rápidos para un determinado problema, fue implementado en los inicios de la plataforma y se concibió como una implementación temporal, priorizando que el sistema funcionara y fuera sencillo de implementar a que fuera eficiente y escalable. Debido a los volúmenes de información que maneja la plataforma actualmente, esta implementación resulta lenta y obsoleta, requiriendo mucho tiempo de CPU y pudiendo llegar a colapsar la plataforma en momentos con mucha demanda. Por ello, una revisión y reimplementación es crítica y necesaria para que la aplicación pueda seguir creciendo. También se realizarán ampliaciones al sistema de rankings, permitiendo a los usuarios consultar la posición en tiempo real de sus envíos o la clasificación para un problema y lenguaje de programación específicos. Además, incluyéndose también dentro del objetivo de fomento de la interacción social, se encuentra la incorporación de nuevas funcionalidades que potencien la interacción social entre los usuarios, como los botones de 'Me gusta' y los seguidores.

Por todo esto, se ha realizado un estudio del funcionamiento de la web para encontrar la mejor manera de integrar las nuevas funcionalidades con las ya existentes. Dado que la plataforma lleva en funcionamiento varios años, las modificaciones deben ser compatibles con la base de datos ya existente y con la información que ésta contiene. También debe tenerse en cuenta los procesos y funcionalidades ya existentes para ser compatibles con el frontend, el servicio web y Tomcat a la espera de que estos sean ampliados.

Respecto a la implementación, se desea que la gestión de los 'Me gusta' y los seguidores sea similar a la de otros elementos de la plataforma. Además, las optimizaciones en los rankings deben ser transparentes a los usuarios y administradores, requiriendo pocas o ninguna modificación en el resto de capas de la aplicación.

Al comienzo del proyecto, se puso a mi disposición dos proyectos de Java que contenían las partes de la plataforma necesarias para la realización del mismo:

- ACR-Model: este proyecto consta de la lógica más profunda de la aplicación y los Data Access Objects para la comunicación con la base de datos permanente.
- ACR-Model-Tools: este proyecto auxiliar consta de varias clases auxiliares que permiten construir y poblar una base de datos local, tanto a pequeña escala como a una escala comparable a la del sistema real, y realizar pruebas tanto de corrección como de carga sobre las clases de ACR-Model.

Además, de los proyectos anteriores se contaba con dos versiones distintas: la primera, la versión actualmente en funcionamiento de la plataforma, y la segunda, una versión ampliada del proyecto resultado de un TFG anterior. Dicha versión expandía la funcionalidad social de '¡Acepta el reto!' con un sistema de logros. El código aportado por este TFG ha sido fundamental, ya que el sistema de eventos que presenta y sobre el que se construyen los logros ha sido una pieza clave para realizar algunas pruebas de las nuevas funcionalidades y, sobre todo, para poder optimizar los rankings.

El sistema de eventos incorporado en el backend de la versión alternativa de '¡Acepta el reto!' recoge un historial de los sucesos relevantes que han ocurrido en la plataforma a lo largo del tiempo. Estos eventos pueden ser utilizados por cualquier entidad, aunque están especialmente pensados para las entidades que realicen tareas de forma periódica, de manera que puedan obtener información de todos los sucesos relevantes que han ocurrido en la plataforma desde su última ejecución. Dada la enorme cantidad de eventos que se producirán debido al tráfico actual en la plataforma, la tabla de la base de datos donde están recogidos los eventos será volátil, borrando periódicamente eventos antiguos que ya no sean de utilidad para la plataforma al haber sido tratados por todas las entidades que hagan uso de ellos.

El sistema de logros incorporado en el backend de la versión alternativa de '¡Acepta el reto!', el cual es la razón de la creación del sistema de eventos, otorga a los usuarios medallas o logros que pueden mostrar en sus perfiles. Dichos logros se obtienen realizando diversas tareas, como resolver un determinado número de problemas, resolver un problema a la primera o, incluso, fallar muchas veces un problema. Aunque aún no está integrado en el sistema real de '¡Acepta el reto!' al no estar todavía implementado en el frontend, el sistema de logros es correcto, completo y funcional en el backend. Por ello, se ha hecho uso de él en parte de las pruebas de corrección sobre el código implementado.

4. Primera aproximación de la plataforma a las redes sociales

Actualmente, las posibilidades de interacción entre los usuarios son nulas más allá de la competitividad otorgada por las clasificaciones. A vista de la demanda de los propios usuarios de más posibilidades de interacción y comunicación entre ellos, se ha propuesto incorporar a la plataforma diversos sistemas que permitan dicha interacción, como botones de 'Me gusta', posibilidad de seguir a usuarios, comentarios, mensajes privados o integración con redes sociales como Facebook o Twitter.

Para poder iniciar mi trabajo e ir familiarizándome con el código, se me asignaron un par de tareas sencillas y asequibles de entre las mencionadas para dar a '¡Acepta el reto!' un carácter más social: botones de 'Me gusta' y de 'Seguir'. Aunque están desconectadas del grueso del trabajo, querían implementarse en la plataforma en un futuro cercano, como se ha expuesto en el párrafo anterior. Dado que este trabajo se centra en el backend de la plataforma, dicha funcionalidad no está completa, ya que para su total integración requerirá ampliaciones en el resto de capas de la aplicación en el futuro. Sin embargo, esta integración se sale del alcance de este proyecto.

4.1. Implementación del sistema de 'Me gusta'

La funcionalidad de los botones de 'Me gusta' en la aplicación fue propuesta para añadir algo de interacción social a la aplicación. En principio se pensó para que los usuarios pudieran dar 'Me gusta' a los problemas, aunque no se descarta la posibilidad de dar 'Me gusta' a otras entidades en el futuro. Por ello, se plantearon inicialmente dos posibles implementaciones:

- Crear una tabla de pares (id usuario, id problema) donde cada pareja representara un 'Me gusta' de un usuario a un problema.
- Crear una tabla de tuplas (id usuario, id entidad, tipo entidad) más genérica, que permitiera asociar un 'Me gusta' de un usuario a cualquier otra entidad del modelo.

Otras implementaciones, como almacenar los 'Me gusta' en las tablas de Usuarios o Problemas, fueron directamente descartadas, ya que el número de 'Me gusta' es variable y aumentaría el tamaño de las filas de dichas tablas indefinidamente. Guardar únicamente en las tablas de Usuarios y/o Problemas el número de 'Me gusta' dados o recibidos también fue descartado, ya que no se podría realizar la correspondencia usuario-problema para cada 'Me gusta' concreto.

De las dos implementaciones planteadas, aunque la segunda resulta más genérica permitiendo ampliar la funcionalidad de los 'Me gusta' de manera trivial, esta implementación presenta problemas de integridad referencial. No se puede realizar una correspondencia entre la columna ID Entidad y la columna de ID de otra tabla ya que, dependiendo del campo Tipo Entidad, ID Entidad hará referencia a tablas distintas. Usando la primera implementación, se puede aprovechar la integridad referencial que proporcionan las claves externas de SQL, dejando abierta a estudio la posibilidad de ampliar los 'Me gusta' a otras entidades.

Elegida la implementación, fue necesario crear una nueva tabla en la base de datos con los siguientes campos:

- User_id: clave externa que referencia la clave primaria ID de la tabla de usuarios.

- Problem_id: clave externa que referencia la clave primaria InternalID de la tabla de problemas.

Ambas juntas forman la clave primaria de la tabla, haciendo que las tuplas (id usuario, id problema) sean únicas. La tabla también viene representada por la clase **Like.java**. Además, se ha añadido la clase **LikeDAO.java** que proporciona una interfaz para la comunicación con la base de datos, permitiendo las siguientes operaciones:

- Añadir un 'Me gusta' a la base de datos.
- Extraer todos los 'Me gusta' de la base de datos.
- Extraer todos los 'Me gusta' dados por un usuario concreto.
- Contar los 'Me gusta' dados por un usuario concreto.
- Extraer todos los 'Me gusta' recibidos por un problema concreto.
- Contar los 'Me gusta' recibidos por un problema concreto.
- Comprobar si un usuario concreto ha dado 'Me gusta' a un problema concreto.

También se han integrado los 'Me gusta' con el sistema de eventos. Se ha creado un nuevo tipo de evento LIKE en **EventType.java**, un nuevo subtipo LIKE_EXECUTED en **Event.java** y una función en **EventDAO.java** para crear e insertar el evento en la base de datos.

4.1.1. Validación

Para probar el sistema de 'Me gusta' se ha hecho uso del sistema de logros implementados. Hemos creado varios logros relacionados con los 'Me gusta', en concreto:

Nombre de la clase	Título	Descripción
DiezLikes.java	Usuario amable	El usuario ha dado Like 10 veces.
CienLikes.java	Programador agradecido	El usuario ha dado Like 100 veces.
MilLikes.java	"¡Os traigo amor!"	El usuario ha dado Like 1000 veces.

Para realizar las pruebas, se ha reutilizado la clase **PueblaBD.java**, que a su vez llama a **CreateShema.java** para volver a crear la estructura de la base de datos, y posteriormente se puebla la base de datos con 3500 usuarios, 1200 problemas y los logros anteriores.

También se ha creado la clase **TestLikesAndFollows.java** basada en la clase **TestAchievements.java** usada para probar el sistema de eventos y logros. Dicha clase asigna una gran cantidad de 'Me gusta' a diversos usuarios, intercalado periódicamente el demonio que actualiza los logros. Dado que los logros se asignan correctamente, damos por verificada la implementación.

4.2. Implementación del sistema de seguidores

La opción de seguir a otros usuarios ampliaría la interacción entre estos dentro de la aplicación de '¡Acepta el reto!', permitiendo recibir las actividades que realizan ordenadas cronológicamente en un feed. En el análisis de las posibles implementaciones para que los usuarios se pudieran seguir, se vio que compartía muchísimas características con el sistema de 'Me gusta', con la salvedad de sólo tiene sentido que los usuarios sigan a otros usuarios, por lo que no había ninguna necesidad de generalizar el esquema.

En consecuencia, el esquema que se siguió era prácticamente idéntico al de los 'Me gusta'. Se creó una nueva tabla con dos campos Follower_id y Follow_id, ambos claves externas que referencian a la clave primaria ID de la tabla de usuarios. Además, ambas juntas forman la clave primaria, haciendo que las tuplas (follower_id, follow_id) sean únicas. La tabla viene representada también por la clase **Follow.java**, y la clase **FollowDAO.java** proporciona una interfaz para la comunicación con la base de datos, permitiendo las siguientes operaciones:

- Añadir un Follow a la base de datos.
- Extraer todos los Follow de la base de datos.
- Extraer todos los usuarios a los que sigue un usuario concreto.
- Contar los usuarios a los que sigue un usuario concreto.
- Extraer todos los usuarios que siguen a un usuario concreto.
- Contar los usuarios que siguen a un usuario concreto.
- Comprobar si un usuario concreto ha seguido a otro usuario concreto.
- Obtener todos los eventos que hagan referencia a los usuarios seguidos por un usuario concreto (obtener el feed para un usuario concreto).

También se han integrado los Follows con el sistema de eventos. Se ha creado un nuevo tipo de evento FOLLOW en **EventType.java**, un nuevo tipo FOLLOW_PERFORMED en **Event.java** y una función en **EventDAO.java** para crear e insertar el evento en la base de datos. Para obtener el feed de un usuario, ha sido necesario añadir otra función en **EventDAO.java** que devuelva la lista de eventos que hagan referencia a alguno de los usuarios dentro de una lista.

4.2.1. Validación

Para probar el sistema de seguidores se ha hecho también uso del sistema de logros implementados. Hemos creado varios logros relacionados con los seguidores, en concreto:

Nombre de la clase	Título	Descripción
DiezFollowsEmitidos.java	Admirador	El usuario ha dado Follow a 10 usuarios.
CienFollowsEmitidos.java	Cotilla	El usuario ha dado Follow a 100 usuarios.
MilFollowsEmitidos.java	Acosador	El usuario ha dado Follow a 1000 usuarios.
DiezFollowsRecibidos.java	Programador reconocido	El usuario es seguido por 10 usuarios.
CienFollowsRecibidos.java	VIP (Very Important Programmer)	El usuario es seguido por 100 usuarios.
MilFollowsRecibidos.java	Superestrella informática	El usuario es seguido por 1000 usuarios.

Para realizar las pruebas, se han reutilizado de nuevo las clases **CreateSchema.java** y **PueblaBD.java**, añadiendo los logros anteriores. También se ha realizado las pruebas utilizando la clase **TestLikesAndFollows.java** conjuntamente con las pruebas de los 'Me gusta'. La clase asigna una gran cantidad de seguidores a diversos usuarios, intercalado periódicamente el demonio que actualiza los logros. Dado que los logros se asignan correctamente, damos por verificada la implementación.

5. Obtención de la posición en el ranking en tiempo real

Actualmente, cuando un usuario realiza un envío que da como veredicto 'Aceptado (AC)', '¡Acepta el reto!' calcula la posición que ha obtenido en la clasificación todos los envíos realizados al problema. Para ello, los envíos se ordenan de menor a mayor por tiempo de ejecución, con precisión de milisegundos. En caso de empate, se ordenan por ID del envío, lo que resulta en una ordenación por fecha de envío de más antiguo a más reciente. De esta manera, se obtiene un ranking ordinal sobre el que se obtiene la posición que ocupa la solución propuesta en la clasificación en el momento del envío.

Últimos envíos

Envío	Usuario	Resultado	Lenguaje	Tiempo	Memoria	Pos	Fecha
137156	danilop	AC	C++	0.112	1680	200	Hace 5 días
137155	danilop	WA	C++	0.096	1680	-	Hace 5 días
136860	DanielR	AC	Java	0.786	1749	338	Hace 8 días
136311	primagaz	AC	Java	0.828	1732	365	Hace 12 días
135382	Julio_A_Laria	AC	Java	0.714	1739	305	Hace 22 días
135381	Julio_A_Laria	WA	Java	0.854	1732	-	Hace 22 días
132745	ivanspasov	AC	Java	0.699	1739	299	2017-04-19 03:23:09
131653	whathefckjava	AC	Java	0.419	1685	275	2017-04-09 14:58:04
131652	whathefckjava	AC	Java	0.449	1735	278	2017-04-09 14:56:18
129681	pedrojanula	AC	C	0.08	1112	179	2017-03-30 14:02:18

Últimos envíos del problema 313: Fin de mes.
Se han marcado las posiciones (en el momento del envío) de las soluciones aceptadas.

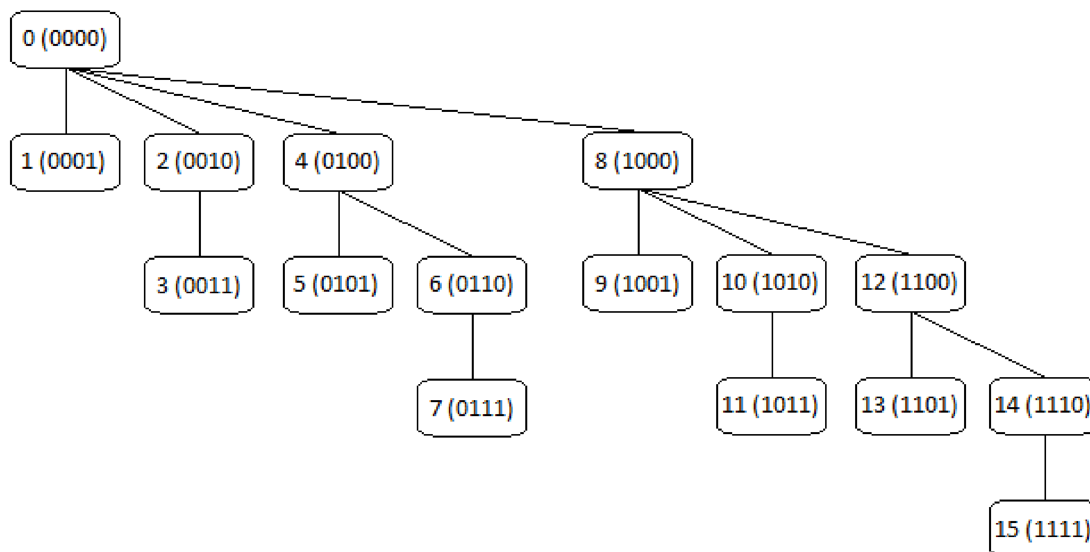
Sin embargo, esta posición no se actualiza con el tiempo, sino que permanece estática y no varía cuando otros usuarios realizan envíos más rápidos. Por ello, se plantea la necesidad de obtener la posición que ocupa nuestro envío en tiempo real, pudiendo percibir en todo momento la posición que ocupamos dentro del ranking, estemos en los primeros puestos o no. Conocer esta información podría fomentar la interacción y competitividad entre los usuarios, ya que podrían conocer cuándo son superados por otros usuarios sin necesidad de mirar el ranking de los 20 primeros (si es que aparecen allí).

Traducido a la implementación, en la actualidad en la tabla de envíos (Submissions) existe una columna con el puesto que obtuvo el usuario en el ranking en el momento de la corrección, pero dicho puesto es estático y no se actualiza con envíos posteriores. Además, el cálculo de este puesto resulta lento y costoso, ya que requiere una consulta que cribe los mejores envíos del problema en la tabla de envíos de la base de datos, la cual almacena más de 138.000 envíos. Queremos que el usuario pueda consultar en qué puesto se encuentra su envío en tiempo real, y el tiempo de consulta sea lo más rápido posible.

La opción obvia sería realizar la consulta antes mencionada, lo cual es impracticable. El tamaño de la tabla de envíos haría que el filtrado, la ordenación y la búsqueda fuera excesivamente costosas en tiempo. Esto nos obliga a plantearnos otras opciones que, aunque puedan gastar algo de memoria, mejoren el tiempo necesario para realizar la consulta de manera significativa.

5.1. Árboles de Fenwick

La opción que se nos planteó fue usar árboles de Fenwick [29]. Los arboles de Fenwick son una estructura de datos que permite calcular las sumas de los *k* primeros elementos de un array en tiempo logarítmico, y permite actualizar dichos elementos también en tiempo logarítmico. En el árbol de Fenwick (entendido como un vector indexado desde el 1), cada nodo de índice ***i***, con representación binaria ***b(i)***,tiene como padre al nodo ***b(i) - LSB[b(i)]***, donde ***LSB[b(i)]*** es la potencia de 2 del bit a 1 menos significativo de ***b(i)***. Esto es, por ejemplo, el padre del nodo 13 (1101) sería el nodo 12 (1100), cuyo padre sería el nodo 8 (1000). De esta manera, del nodo 0 (que sería un nodo fantasma), colgarían las potencias de 2: 1 (1), 2 (10), 4 (100), 8 (1000)… y de estos colgarían los nodos que son suma de dos potencias de 2, la de su padre y una más pequeña. Por ejemplo, del nodo 8 (1000) colgarían el 9 (1001), el 10 (1010) y el 12 (1100).



Estructura del árbol de Fenwick para un vector de 15 elementos: v[1..15]
La definición del árbol resulta en una estructura de árbol binomial [30].

Definimos la función ***P(i)***, que dado un índice ***i*** devuelve el índice de su padre, que tiene como representación binaria ***b(i) - LSB[b(i)]***. Si tomamos esta estructura del árbol de Fenwick, un vector indexado desde el 1 y la función ***P(i)***, entonces en cada nodo se almacena la suma de los elementos cuyo índice se encuentra en el intervalo ***(P(i), i]***, o lo que es lo mismo, ***[P(i)+1, i]***. De esta manera, para calcular la suma de los ***k*** primeros elementos del array habría que sumar todos los nodos del camino del nodo de índice ***k*** hasta la raíz. Como el árbol tiene una altura logarítmica respecto al tamaño del array, el coste de la operación es ***O(log n)***. Igualmente, para modificar el elemento ***i*** del array sumando o restando una cantidad ***delta***, la actualización del árbol de Fenwick requiere actualizar todos los elementos del árbol que contengan al elemento ***i*** en su suma parcial. Esto requiere saltar al hermano derecho, o en su defecto al hermano derecho de su padre, actualizando todos los nodos que encontremos por el camino sumando o restando delta. Esto se puede traducir a iterar ***i := i + LSB[i]*** actualizando todos los nodos por el camino hasta salirnos del rango de índices. El coste de esta operación también sería ***O(log n)***.

A la hora de la implementación, la más común suele ser representar el árbol en un array plano, y navegar a través de él usando operaciones en el índice. Además, se suele aprovechar que los

ordenadores usan notación de complemento a 2 en los enteros para calcular ***LSB[i]*** como ***i & (-i)***, donde ***&*** representa el AND lógico bit a bit.

5.2. Aplicación en el proyecto

En el problema que nos concierne, la obtención del puesto de un envío en tiempo real podría obtenerse si, por cada problema, existiese un vector indexado por tiempo de ejecución que en cada posición tuviese el número de envíos con exactamente ese tiempo de ejecución. La obtención de la posición en el ranking (obviando los empates, que comentaremos más adelante) se podrían obtener sumando todos los elementos del vector con índice menor. Para optimizar este cálculo, se construirá un árbol de Fenwick basado en este vector de frecuencias, de manera que el árbol será lo que se mantendrá en memoria para realizar las consultas.

La implementación del árbol de Fenwick se ha realizado en la clase **FenwickTree.java**, y permite en tiempo logarítmico las dos operaciones comentadas anteriormente: suma de prefijos y actualización de elementos. Además, el árbol de Fenwick puede construirse vacío, a partir de un vector de frecuencias o a partir de una representación del árbol en String, consistente en los elementos del vector separados por espacios hasta el último elemento relevante del vector de frecuencias. Además, el árbol es capaz de crecer a demanda manteniendo sus propiedades.

A pesar de que los árboles de Fenwick se mantendrán en memoria en el servidor, el cual se ejecuta indefinidamente, es necesario poseer una copia de seguridad por si se cae el servidor o es necesario reiniciarlo. Para ello, se realizará periódicamente una copia de seguridad en la base de datos de todos los árboles, especificando el problema al que hacen referencia y el número de envío hasta el que están actualizados. El árbol en sí se guardará en formato texto, donde el array que representa al árbol se traducirá en un String de números separados por espacios. Será necesario crear una nueva tabla **ProblemFTree** en la base de datos, una clase **ProblemFTree.java** que lo represente y otra que gestione las transacciones con la base de datos, llamada **ProblemFTreeDAO.java**. Los campos de la tabla son los siguientes:

- Problem: clave primaria y clave externa que hace referencia a la clave primaria InternalID de la tabla Problems.
- LastSubmission: clave externa que hace referencia a la clave primaria ID de la tabla Submissions
- Ftree: campo en formato texto que contendrá el árbol de Fenwick.

La gestión de los árboles para obtener el puesto en tiempo real se realiza en la clase **FenwickTreeManager.java**. La clase implementada como Singleton contendrá un array de árboles indexado por ID interno del problema. Dicha clase contiene métodos para leer los árboles de la base de datos, guardarlos en la base de datos, actualizarlos con los últimos envíos o crearlos desde cero a partir de todos los envíos aceptados del problema. Además, esta clase será la que se encargue de proporcionar el número de envíos aceptados de un problema cuyo tiempo de ejecución sea menor a un tiempo dado. Al ser esta última operación eficiente en tiempo, ahora es practicable realizar consultas sobre la posición de los envíos en tiempo real. Además, se puede aprovechar esta estructura para calcular el puesto en el momento del envío, de manera que ya no requiera una consulta costosa a la base de datos.

5.3. Ordenación de envíos con el mismo tiempo de ejecución

Ahora bien, aunque esta estructura resuelve muchos problemas a la hora de averiguar el puesto de un envío según su tiempo de ejecución, no lo soluciona al completo. En caso de que dos envíos hayan obtenido el mismo tiempo de ejecución, deben ordenarse por fecha de envío (es decir, por ID de envío), obteniendo una posición mejor el envío más antiguo. Sin embargo, dicha ordenación se puede obtener en el momento del envío y es válida indefinidamente: a mismo tiempo de ejecución, el envío estará por detrás de todos los que ya se hayan realizado ya y delante de todos los que realicen en el futuro.

Esta solución, trasladada a la implementación, puede plasmarse como una nueva columna en la tabla de envíos, que indique el puesto que ocupa entre los envíos con mismo tiempo de ejecución (donde 0 es el primer puesto). En el momento en el que el envío sea aceptado y se calcule su tiempo de ejecución, se obtiene el número de envíos con exactamente ese tiempo de ejecución y se añade a la fila del envío en la tabla. Esto se puede obtener haciendo una consulta a la tabla de envíos, o mejor, al árbol de Fenwick (el número de envíos con un tiempo de ejecución ***t*** es la diferencia entre el número de envío con tiempo menor o igual que ***t*** menos el número de envíos con tiempo estrictamente menor que ***t***). De esta manera, se puede hacer una ordenación total de los envíos.

Para ello, se ha requerido añadir dicha columna a la tabla **Submissions** de la base de datos y a la clase que la representa, **Submission.java**. También ha sido necesario modificar **SubmissionDAO.java** para gestionar la nueva columna y utilizar los árboles de Fenwick para calcular el puesto.

6. Optimización de la clasificación

La clasificación de '¡Acepta el reto!' muestra los 20 envíos más rápidos (de usuarios distintos) para un problema ordenados de forma creciente por tiempo de ejecución (con precisión de milisegundos) y, en caso de empate, por fecha de envío en orden también creciente (ordenándolos por ID de envío). De esta manera se obtiene un ranking ordinal como el siguiente:

Clasificación

Pos	Envío	Usuario	Lenguaje	Tiempo	Memoria	Fecha
1	61366	ronaldo	C	0.012	1120	2016-03-02 23:32:39
2	63823	Botarga	C++	0.012	1692	2016-03-10 21:55:34
3	116555	mfernandez	C++	0.012	1684	2017-02-24 21:56:08
4	3321	alex.b	C	0.016	988	2014-03-19 02:19:05
5	6212	marco.d	C	0.016	988	2014-04-23 19:49:15
6	6394	B01001101	C	0.016	988	2014-04-25 17:51:01
7	1182	Rohan080	C	0.02	988	2014-03-07 17:11:15
8	37244	nachete	C	0.02	1120	2015-06-01 12:35:35
9	126074	albertobsd	C	0.02	1116	2017-03-16 21:19:56
10	116	raxkin	C	0.024	988	2014-02-26 17:11:34
11	486	Scofield_84	C	0.024	988	2014-03-01 02:42:27
12	857	cuellar_machine	C	0.024	988	2014-03-04 19:00:11
13	1896	marcant94	C	0.024	988	2014-03-12 12:55:00

Parte de la clasificación para el problema 100: Constante de Kaprekar.

Es necesario recalcar la diferencia entre este ranking y la posición proporcionada por los árboles de Fenwick. Los árboles de Fenwick proporcionan la posición que ocupa un envío entre todos los envíos realizados de un problema, sean de usuarios distintos o no. Esto hace que se tengan en cuenta todos los distintos envíos que haya realizado un mismo usuario. Sin embargo, en la clasificación que trataremos ahora, cada usuario puede aparecer una única vez como máximo. En caso de que un usuario haya realizado más de un envío al problema, sólo se verá reflejado en la clasificación el que haya obtenido el mejor tiempo de ejecución. Esto añade una complejidad extra al ser necesario filtrar los envíos por usuario y seleccionar el mejor de cada uno.

La clasificación es hoy día el único punto de interacción entre los usuarios, fomentando la competitividad y la autosuperación. Sin embargo, la clasificación actual se queda escasa al ofrecer una información limitada y muy poco diversa. Además, la actualización de la clasificación no es instantánea, de manera que usuarios que han conseguido buenos resultados tienen que esperar varios minutos hasta ver su resultado reflejado en el ranking. Dicha información tampoco está siempre disponible: durante algunos concursos el abundante tráfico de información en la plataforma hizo que se tuviera que desactivar el ranking por consumir demasiados recursos. Por ello se nos plantea realizar una revisión de esta parte de la plataforma, intentando optimizarla de manera que sea capaz de funcionar de manera eficiente incluso cuando el tráfico en la plataforma es muy alto y, a ser posible, ofreciendo resultados en tiempo real.

Desde el punto de vista de la implementación, la obtención de dicha clasificación se realiza con un demonio, que periódicamente realiza consultas SQL en la base de datos, actualizando los rankings de los problemas uno a uno. Sin embargo, no se obtienen los 20 primeros puestos, sino que se calcula el puesto para todos los usuarios que hayan realizado un envío al problema. Se ha puesto como exigencia que el nuevo diseño debe calcular el ranking para todos los usuarios que hayan realizado algún envío al problema. Dado el tamaño actual del sistema y el número de envíos que se han realizado a la página a lo largo del tiempo, la obtención de la clasificación consume muchos recursos, siendo uno de los puntos críticos de la plataforma. Por ello se ha planteado revisar esta parte de la implementación con el fin de optimizarla.

6.1. Diseño

Mientras que para la obtención de la posición de un envío en el ranking en tiempo real desde un principio se vio que los arboles de Fenwick cumplían los requisitos pedidos de manera rápida y eficiente, para esta parte del proyecto la solución a implementar no estaba clara. Por ello, se han planteado diversas estructuras de datos que se podrían mantener en memoria para conseguir optimizar la clasificación.

6.1.1. Opción 1: Tablas Hash

La primera opción que se barajó se basó en el vector de frecuencias del punto anterior. Se podría mantener otro vector en el que en cada posición, en vez de almacenarse el número de envíos, almacenara los IDs de los envíos (con posiblemente otra información adicional). Para optimizar la memoria, dicho vector se podría implementar como una tabla Hash. Analicemos su rendimiento:

- Para su creación, sería necesaria una consulta SQL de los envíos más rápidos (de usuarios distintos), y su inserción en tiempo constante en la tabla.
- Para su actualización, si demoramos la eliminación de duplicados a la consulta, sería insertar el nuevo envío en tiempo constante.
- Para la consulta de los 20 primeros (o de cualquier rango de puestos), sería necesario un recorrido de la tabla en orden para extraer los envíos. En el caso peor, podría conllevar un recorrido de toda la tabla. Si realizamos la eliminación de duplicados en la consulta, tendríamos que consultar si ya hemos extraído un envío del mismo usuario para no añadirlo también al ranking y eliminarlo de la tabla. Además, cuando se hayan extraído los envíos necesarios, habría que seguir recorriendo la tabla para eliminar todos los demás repetidos. De esta manera, cada consulta requeriría un recorrido completo de la tabla.
- La consulta del puesto de un usuario específico no presenta otra opción que el recorrido (lineal) de la tabla para encontrarlo.

Como podemos ver, esta implementación contiene muchos problemas de gestión, de consistencia y eficiencia, por lo que no es una opción a tener en cuenta.

6.1.2. Opción 2: Montículos

Otra aproximación sería utilizar un montículo de mínimos para almacenar el ranking. Utilizar montículos tiene la ventaja que su ordenación se puede realizar en tiempo ***O(n log n)***. De esta manera, se puede mantener el ranking parcialmente ordenado, y ordenarlo completamente en el momento de la consulta. Una primera aproximación de las operaciones sería la siguiente:

- Para su creación, se realizaría una consulta SQL y se realizarían las inserciones necesarias que, aunque en teoría serían en tiempo logarítmico, si los insertamos en orden creciente el coste de las inserciones se vuelve constante.
- Para su actualización, si demoramos la ordenación total hasta la consulta, sería realizar la inserción del nuevo envío en el montículo. Si la eliminación de duplicados se realiza en la actualización. sería necesario un recorrido del montículo y, en caso de que se encontrara coincidencia, se puede sustituir y hacer subir o bajar el nuevo elemento para preservar las propiedades del montículo. La actualización por tanto sería lineal.
- En la consulta de los 20 primeros (o en cualquier rango de puestos), se extraería el mínimo del montículo de manera repetida hasta completar el ranking, teniendo cuidado de no introducir varios envíos del mismo usuario si esto no se asegura en la actualización. Dado que el montículo se modifica al realizar las consultas, es necesario duplicarlo primero.
- Para la búsqueda del puesto de un usuario, como el array no está totalmente ordenado, no queda más remedio que extraer los elementos uno a uno hasta encontrar el deseado.

A esta implementación inicial se le pueden realizar diversas modificaciones para mejorar sus propiedades y eficiencia:

- En vez de duplicar el montículo en la consulta, se puede utilizar el ranking obtenido para crear un nuevo montículo. De esta manera, aprovechamos las consultas para eliminar envíos de un mismo usuario y obtenemos no solo un montículo, sino un array totalmente ordenado.
- Aprovechando la mejora anterior, al realizar una consulta, si no se ha recibido ningún envío nuevo desde la anterior consulta, podemos volcar directamente el montículo en el ranking sin necesidad de ningún procesamiento.

Aunque esta implementación mejora significativamente la gestión de los envíos, sigue presentando problemas. Si tenemos en cuenta que la ordenación por montículo no es una ordenación natural, es decir, que su tiempo de ejecución y/o complejidad no necesariamente decrece cuanto más ordenados estén los datos, obtenemos nuevos problemas en la eficiencia. Un único envío que entrara en el ranking provocaría una ordenación ***O(n log n)*** en la consulta del montículo. Además, la ordenación se dispararía siempre que hubiera nuevos envíos, es decir, constantemente. Por ello, se nos plantea buscar una nueva implementación que aproveche estas propiedades.

6.1.3. Opción 3: Vector ordenado por inserción

Vistas las ventajas y problemas de la solución anterior, se decidió descartar e intentar otra aproximación: mantener los rankings completamente ordenados en memoria, e utilizar la inserción para actualizarlos. El algoritmo de inserción es un algoritmo de ordenación natural, consiguiendo complejidad lineal para datos ordenados o casi ordenados. Si nos basamos en la implementación del montículo y la analizamos, vemos que realmente no ganamos eficiencia en tiempo demorando la ordenación hasta la consulta. En cambio, si mantenemos el vector siempre ordenado, obtenemos una gran ventaja: por cada nuevo envío, bastará con insertarlo en la posición adecuada, lo cual tendrá complejidad lineal como máximo. Además, podemos aprovechar el recorrido del vector para ver si hay otro envío en el ranking del mismo usuario.

La propuesta de implementación sería la siguiente:

- Para su creación, se realizaría una consulta SQL y se volcarían los resultados en un vector.
- Para su actualización, recorremos el vector comprobando que no haya otro envío del mismo usuario con tiempo menor, y si es así descartamos el envío. En caso de que no exista o tenga peor tiempo, insertamos el nuevo envío ordenado y extraemos el que deba salir del ranking.
- En la consulta de los 20 primeros (o de cualquier rango de puestos), basta con volcar el vector en el ranking.
- Para la búsqueda del puesto de un usuario, sería necesario un recorrido del vector.

Con esta implementación, se solucionan varios problemas de eficiencia, ya que la consulta del ranking es lo más eficiente. Sin embargo, tanto la actualización como la consulta de un puesto tendrán coste lineal, y los rankings irían creciendo de tamaño con el tiempo. La actualización será una operación frecuente, pero en cualquier caso, la inserción de un nuevo envío en el ranking requiere la actualización de alguna manera de los puestos posteriores. Esto nos plantea intentar encontrar alguna solución mejor, aunque en eficiencia esta pueda ser de las mejores.

6.1.4. Opción 4: Vector + tabla Hash

Otra opción que se planteó fue añadir al vector una tabla Hash indexada por usuario para hacer más rápida la consulta del puesto de los usuarios. En el caso de que se guardara como valor la posición del usuario, la consulta sería constante, pero las modificaciones en la clasificación requerirían recalcular los puestos de potencialmente muchos usuarios en la tabla Hash. En caso de que se guardara el tiempo de ejecución, no sería necesario modificar la tabla y se podría realizar una búsqueda binaria en el vector para buscar al usuario. Sin embargo, ambas opciones requieren mantener actualizadas dos estructuras de datos distintas, lo que genera problemas de coherencia y puede afectar al rendimiento, especialmente por la cantidad de memoria utilizada.

6.1.5. Opción 5: Optimización de la base de datos

Otra opción si queremos optimizar la memoria al máximo, sería cambiar la estructura de la base de datos y de las consultas para mejorar el tiempo de actualización de los rankings sin tener ninguna estructura en memoria. Se podría aprovechar el sistema de eventos para recalcular el ranking únicamente de los problemas que hayan tenido envíos AC. Seguiría realizándose con un demonio de forma periódica y el cálculo se realizaría desde cero a partir de todos los envíos en la base de datos. Además, si mantenemos en una nueva columna un booleano que especifique si el envío que representa la fila de la tabla es el más rápido del usuario, se podrían filtrar los envíos por esa columna en vez de ordenarlos por tiempo de ejecución y escoger el mejor.

Esta opción tiene como ventaja que permite no gastar nada de memoria, lo que es importante ya que los rankings potencialmente pueden crecer de manera indefinida. Además, sólo se actualizarían las clasificaciones que fueran realmente necesarias. Sin embargo, el proceso de actualización de cada clasificación sigue siendo precario y costoso, y los rankings no estarían actualizados en tiempo real.

6.2. Implementación

De entre las muchas propuestas planteadas, el vector ordenado por inserción es la que proporciona mejores resultados de tiempo, pudiendo ser rápido cuando el número de usuarios siga creciendo. Sin embargo, preocupa la cantidad de memoria que pueda gastar. Por ello, se plantearon dos alternativas de modificaciones de esta implementación para poder ahorrar memoria:

- Para disminuir la cantidad de memoria utilizada, sólo se mantendrán en memoria los rankings que hayan recibido nuevos envíos, los cuales se leerán a demanda. Los rankings que no hayan sido modificados se consultarán en la base de datos directamente. Tras realizar las copias de seguridad en la base de datos, los rankings en memoria se borrarán, liberando espacio. Esta modificación requerirá de algún sistema de marcas en la base de datos para saber hasta qué número de envío está actualizado el ranking en la base de datos

- Para reducir al máximo la cantidad de memoria utilizada, todos los rankings se mantendrán en la base de datos en todo momento. Las consultas se realizarán siempre a la base de datos, y para las actualizaciones se cargará el ranking en memoria, se modificará e inmediatamente se volverá a guardar en la base de datos. Esto reduce es uso de la memoria a cero; sólo se usará temporalmente mientras dure la actualización de un ranking. Sin embargo, aumenta el número de transacciones con la base de datos respecto a la anterior solución.

Dado que las consultas a la base de datos en general son costosas en tiempo, se ha decidido implementar la primera. Además, dado que las consultas de Hibernate devuelven una lista implementada como ArrayList en caso de que haya varios resultados, aprovecharemos esa misma estructura en vez de usar la clase Vector de Java para mantener los rankings y evitar realizar copias. Esto no nos supone ningún problema, ya que las clases ArrayList y Vector son muy parecidas al estar implementadas ambas como un array, permitiendo accesos a todas las posiciones en tiempo constante. Para su recorrido, la interfaz List nos permite acceder a cualquier elemento y usar iteradores que puedan avanzar y retroceder de manera arbitraria.

Ahora bien, para poder guardar el número de envío hasta el que está actualizado, no sirve la misma estrategia que para el árbol de Fenwick. Como cada árbol de Fenwick estaba almacenado en una única fila de su tabla en la base de datos, bastaba con añadir un nuevo campo. Sin embargo, los rankings ocupan potencialmente muchas filas de la tabla de rankings, por lo que tener el mismo campo repetido en las distintas filas, además de ser poco eficiente, podría generar problemas de inconsistencia. Por ello, se ha decidido crear una nueva tabla **ProblemRankingMark**, que almacene para cada problema el envío hasta el que está actualizado. Esta tabla se verá reflejada en el código Java en la clase **ProblemRankingMark.java**, y su gestión se realizará mediante la clase **ProblemRankingMarkDAO.java**, que permitirá cargar y guardar la marca en la base de datos.

En la lógica de la aplicación, se ha creado la clase **RankingManager.java**, cuya funcionalidad es similar a la de **FenwickTreeManager.java**. La clase implementada como Singleton contendrá un array de árboles indexado por ID interno del problema, y contendrá métodos para leer los rankings de la base de datos, guardarlos en la base de datos, actualizarlos con los últimos envíos o crearlos desde cero a partir de todos los envíos aceptados del problema. Además,

esta clase se encargará de proporcionar los rankings a las demás clases, y tendrá métodos para consultar la posición que ocupa un usuario en el ranking para un problema determinado. Como siempre existirá alguna versión del ranking actualizada, ya sea en memoria o en la base de datos, los cambios en el ranking se apreciarán en tiempo real.

7. Rankings por lenguaje

Como hemos comentado anteriormente, ofrecer diversas clasificaciones y estadísticas ofrece a los usuarios más información sobre el problema y sus envíos, y aumentaría la competitividad entre éstos y, como consecuencia, el uso de la plataforma. Se quería aumentar la funcionalidad de la aplicación añadiendo nuevos rankings que ofrecieran otras estadísticas más variadas. Una de las opciones es ampliar los rankings anteriores para obtener estadísticas similares a las existentes pero para un lenguaje de programación concreto. En concreto, poder obtener:

- Los mejores 20 envíos (o en general cualquier rango) de usuarios distintos para un problema y lenguaje determinados.
- Obtener la posición de tu envío de un problema en tiempo real, de entre todos los envíos que se hicieron con el mismo lenguaje.

Implementar estas funcionalidades requiere una revisión del código ya implementado para, siempre que sea posible, permitir nuevas consultas igual de eficientes que las ya existentes sin mermar el coste de las ya implementadas.

7.1. Clasificación por lenguaje

Para la obtención de los 20 primeros puestos (u otro rango) para un problema y lenguaje, la primera opción que se nos plantea es copiar la estructura, de manera que cada una gestione los envíos de un lenguaje determinado. Como '¡Acepta el reto!' actualmente admite 3 lenguajes de programación distintos (C, C++ y Java), se crearían 3 rankings distintos para cada uno de los lenguajes. Una duda que se nos plantea es si mantener el ranking común. Analicemos las ventajas de cada uno:

Ventajas de eliminar el ranking común:

- Se ahorraría memoria.
- Cada envío se gestionaría en un único ranking, reduciendo a la mitad el tiempo de gestión y evitando inconsistencias entre el ranking común y el específico del lenguaje.
- El ranking común se puede obtener mediante una mezcla de los rankings específicos, lo que tendría coste lineal respecto al tamaño del ranking. No depende del número de lenguajes en el sentido de que la mezcla se aborta cuando se obtengan los ***n*** envíos más rápidos de entre todos los lenguajes, siendo ***n*** el límite superior del rango de posiciones que se quiere consultar.
- Desde el punto de vista de la implementación, mantener el ranking común presenta problemas a la hora de almacenarlo en la base de datos con la estructura que tenemos actualmente. Si aprovechamos la estructura actual, cambiaría la gestión de los lenguajes, al tener que instanciar un lenguaje genérico o "sin especificar". Esto afecta al código relativo a los lenguajes en el resto de la aplicación. Si queremos evitar esto, habrá que rehacer toda la organización de los rankings en la base de datos.

Ventajas de mantener el ranking común:

- El ranking común estaría siempre en memoria disponible para volcarlo en el frontend, ahorrando las múltiples comparaciones que requiere la mezcla, que crecen de manera lineal con el número de lenguajes.

Como podemos conseguir buenos resultados sin reimplementar los rankings desde cero, no merece la pena mantener el ranking común. Esta modificación requiere varios cambios en la implementación:

- Se ha añadido un nuevo campo **lang** a la tabla **ProblemRanking**. Este campo ahora forma parte de la clave primaria junto con el problema y la posición. Se ha añadido dicho campo también a la clase **ProblemRanking.java**.
- Se ha añadido igualmente otro campo **lang** a la tabla **ProblemRankingMark**. Este campo ahora forma parte de la clave primaria junto con el problema. Se ha añadido dicho campo también a la clase **ProblemRankingMark.java**.
- Se ha añadido una nueva función a **ProblemRankingDAO.java** que devuelva la clasificación asociada a un problema y lenguaje. También se ha modificado la que devolvía la clasificación asociada a un problema para que devuelva todos los rankings de los diversos lenguajes.

Sin embargo el cambio más notorio está en el **RankingManager.java**. Las clasificaciones dejan de estar organizados en un array indexado por problema y pasan a estar organizados un una matriz indexada primero por problema y después por lenguaje. De esta manera se puede consultar sobre la clasificación asociadas a un problema y un lenguaje concretos, o asociadas sólo al problema, mezclando los resultados de cada uno de los lenguajes.

7.2. Puesto en tiempo real (y en el momento del envío) por lenguaje

Para la obtención del puesto en tiempo real filtrado por lenguaje se puede usar la misma aproximación: mantener un árbol de Fenwick por lenguaje. Igualmente, se nos plantea la duda de si mantener un árbol común. Analicemos las ventajas de cada opción, que son prácticamente iguales:

Ventajas de eliminar el árbol común:

- Se ahorraría memoria.
- Cada envío se gestionaría en un único árbol, reduciendo a la mitad el tiempo de gestión y evitando inconsistencias entre el árbol común y el específico del lenguaje.
- La posición en tiempo real común se puede obtener mediante la suma de las posiciones en tiempo real específicas por lenguaje, lo que tendría coste ***O(L log N)***, donde L es el número de lenguajes y N el tamaño del árbol.

Ventajas de mantener el árbol común:

- La posición en tiempo real común estaría siempre disponible mediante una única consulta de tiempo ***O(log N)***, con N el tamaño del árbol.

Dado que el número de lenguajes es ínfimo en comparación al tamaño del árbol, la obtención de la posición en tiempo real para todos los lenguajes tiene costes similares en ambas

implementaciones, por lo que la mejor opción es mantener únicamente los árboles específicos de un lenguaje y eliminar el común.

Esta modificación requiere varios cambios en la implementación, análogos a los anteriores:

- Se ha añadido un nuevo campo **lang** a la tabla **ProblemFtree**. Este campo ahora forma parte de la clave primaria junto con el problema. Se ha añadido dicho campo también a la clase **ProblemFtree.java**.
- Se ha añadido una nueva función a **ProblemFtreeDAO.java** que devuelva el árbol asociado a un problema y lenguaje. También se ha modificado la que devolvía el árbol asociado a un problema para que devuelva todos los árboles de los diversos lenguajes.

Sin embargo, el cambio más notorio está en el **FenwickTreeManager.java**. Los árboles dejan de estar organizados en un array indexado por problema y pasan a estar organizados en una matriz indexada primero por problema y después por lenguaje. De esta manera se puede consultar sobre la posición asociadas a un problema y un lenguaje concretos, o asociadas sólo al problema, sumando los resultados de cada uno de los lenguajes.

Estas modificaciones generan un problema con los empates. A mismo tempo de ejecución, dependiendo del ranking que se esté consultando (el general o el específico por lenguaje) el número de envíos con mismo tiempo de ejecución y fecha anterior varía. Por ello, habrá que calcularlos para el propio lenguaje y para el resto de lenguajes y almacenarlos ambos en la fila del envío en la base de datos.

Aprovechando esta modificación, se puede añadir un nuevo campo a la fila para almacenar el puesto que se obtuvo en el momento del envío entre los del mismo lenguaje. Esta pequeña modificación permitirá dar más información a los usuarios sobre sus envíos.

8. Reevaluaciones

Un suceso que ocurre en la plataforma de manera esporádica es la necesidad de volver a corregir un problema que ya había sido evaluado por el juez. Dicho suceso suele ser necesario cuando se produce un fallo en el servidor, dando como resultado de la corrección 'Error Interno (IE)', o existe un error en el enunciado o en los casos de prueba del ejercicio, haciendo que las correcciones que el juez haya realizado sean erróneas. La plataforma debería ser capaz de dar soporte a este tipo de situaciones, de manera que no generen mayores problemas a los administradores. Es más, lo óptimo sería que la intervención de los administradores sea mínima o nula.

Además, ahora que contamos con estructuras que se actualizan en tiempo real a partir de los últimos eventos o sucesos ocurridos en la plataforma, tener en cuenta las reevaluaciones es fundamental. Si la información en estas estructuras no es corregida, afectará a todas las entidades posteriores que hagan uso de esa información errónea, generando un efecto en cadena y corrompiendo cada vez más y más datos de la plataforma. Por esta razón, es bastante crítico tener en cuenta las reevaluaciones, identificarlas y gestionarlas adecuadamente.

Este suceso, que actualmente en el código de '¡Acepta el reto!' no está contemplado y la plataforma no es capaz de gestionarlo, requiere cambios en diversos puntos de la plataforma.

8.1. Ampliación del sistema de eventos

Dado que la corrección de un envío por primera vez y la reevaluación de un envío son situaciones distintas, debería reflejarse en el sistema de eventos de manera distinta. Por ello, en **Event.java** se ha creado un nuevo subtipo de evento SENT_REEVALUATED dentro de los eventos de tipo SENT y una función en **EventDAO.java** para añadir uno de estos eventos a la base de datos. Dentro de los parámetros del evento se guardará la información necesaria para que otras entidades puedan gestionarla. Concretamente, se guardará:

- El usuario al que pertenece el envío.
- El ID del envío.
- El veredicto anterior.
- En caso de que el veredicto fuese AC, el tiempo de ejecución anterior.

También se ha añadido una función en **EventDAO.java** que devuelva los eventos de un tipo y subtipo concretos a partir de un ID dado. Esto nos será útil en puntos siguientes para extraer de la base de datos los eventos SENT_REEVALUATED a partir de un momento dado.

8.2. Ampliación en los árboles de Fenwick

Con la implementación vista hasta ahora, los árboles de Fenwick no cuentan con ningún mecanismo para hacer frente a estas situaciones. Por ello es necesario realizar una pequeña ampliación de su funcionalidad para que pueda gestionarlas correctamente. Para ello, se ha añadido a la tabla **ProblemFtree** y a su representación en Java, **ProblemFtree.java**, un nuevo campo que almacene el evento hasta el que está actualizado el árbol. Aparte de pequeñas modificaciones para la gestión del campo en la creación, carga y almacenamiento, se han creado dos nuevas funciones:

- En los momentos en los que el árbol se esté actualizando en tiempo real, se ha creado una función que permite eliminar un envío en el árbol de Fenwick. Además, actualizará el campo de evento, sustituyéndolo por el evento que ha causado la extracción. El campo de envío no se actualizara, ya que se trata de un envío anterior ya procesado.

- Durante la carga del árbol desde la base de datos, y tras actualizarlo con los últimos envíos, también se actualizará con los últimos eventos, extrayendo el envío del árbol si el veredicto antiguo era AC y añadiéndolo si el nuevo veredicto es AC. Gracias a los parámetros que hemos guardado en el evento, podemos extraer la información del estado anterior. Además, a partir del ID, podemos extraer de la base de datos el envío para consultar el estado actual.

Durante la creación del árbol desde cero, todos los envíos reevaluados contienen la información más reciente, por lo que no es necesario consultar la tabla de eventos.

8.3. Ampliación en las clasificaciones

De igual manera que el caso anterior, no hay ningún mecanismo en la gestión de los rankings que permita gestionar las reevaluaciones y corregir los datos afectados. Por ello, es necesario realizar una pequeña ampliación de su funcionamiento que tenga en cuenta este evento. Dado que el **RankingManager** sigue un diseño análogo al del **FenwickTreeManager**, la ampliación hecha en el caso anterior puede trasladarse con pequeñas modificaciones a los rankings. Se añadirá a la tabla **ProblemRankingMark** y a su respectiva clase Java un nuevo campo que almacene hasta qué evento está actualizado el árbol. Aparte de pequeñas modificaciones para la gestión del campo en la creación, carga y almacenamiento, se han creado dos nuevas funciones:

- En los momentos en los que el ranking se esté actualizando en tiempo real, se ha creado una función que permite eliminar un envío en él, buscando en la base de datos el siguiente mejor envío del usuario para insertarlo, si existe. Además, actualizará el campo de evento, sustituyéndolo por el evento que ha causado la extracción. El campo de envío no se actualizara, ya que se trata de un envío anterior ya procesado.

- Durante la carga del ranking desde la base de datos, y tras actualizarlo con los últimos envíos, también se actualizará con los últimos eventos, extrayendo el envío del ranking si el veredicto antiguo era AC y añadiéndolo si el nuevo veredicto es AC. Gracias a los parámetros que hemos guardado en el evento, podemos extraer la información del estado anterior. Además, a partir del ID, podemos extraer de la base de datos el envío para consultar el estado actual.

Durante la creación del ranking desde cero, todos los envíos reevaluados contienen la información más reciente, por lo que no es necesario consultar la tabla de eventos.

8.4. Modificaciones en la corrección de envíos

Todas las modificaciones anteriores convergen en la función *putSubmissionResult()* de la clase **SubmissionDAO.java**. En esta función, que implementa la modificación de un envío para añadir su veredicto y todas las operaciones asociadas a éste, se han realizado las modificaciones clave para poder gestionar las reevaluaciones. Además, dada la longitud de la

función, se ha aprovechado para extraer varias funcionalidades a funciones auxiliares. El hilo de ejecución quedaría de la siguiente manera:

1. Primero, igual que antes, se comprueba que el veredicto no sea ‘En cola (IQ)’ ni ‘Ejecutando (RU)’. Después, ahora en una función auxiliar, se extrae el envío de la base de datos y se comprueba que tenga veredicto ‘Ejecutando (RU)’. En caso contrario, se ha producido algún error, por lo que se debe abortar.

2. Después, se comprueba si es una reevaluación en una función auxiliar. Teniendo en cuenta que el juez evalúa los envíos en orden, si es una evaluación corriente todos los envíos posteriores estarán ‘En cola (IQ)’, mientras que si es una reevaluación algún envío posterior tendrá veredicto. De esta manera, la identificación de reevaluaciones la puede realizar y gestionar la plataforma automáticamente sin necesidad de que un administrador interfiera.

3. En caso de sea un envío normal, se crea el evento SENT_EVALUATED igual que antes. En caso contrario, se crea el evento SENT_REEVALUATED y, en caso de que el veredicto anterior fuera ‘Aceptado (AC)’, se elimina el envío del árbol de Fenwick y del ranking correspondiente. Además, en caso de que el veredicto fuera o sea ‘Aceptado (AC)’, hay que corregir los campos de número de envíos con el mismo tiempo de ejecución. Para ello, tanto para el veredicto antiguo como el nuevo, si es o fue ‘Aceptado (AC)’, se extraen los envíos posteriores del mismo problema y lenguaje con mismo tiempo de ejecución y se suma o resta una unidad según el caso.

4. Tras gestionar la posibilidad de una reevaluación, se calcula en una función auxiliar la posición en la clasificación y el número de empates con ayuda de los árboles de Fenwick, tanto genéricos como específicos del lenguaje. Después, se añade el envío a los árboles de Fenwick y a los rankings.

5. Por último, igual que en la versión anterior, se actualiza el envío en la base de datos con la nueva información.

9. Estudio de la eficiencia de los sistemas implementados

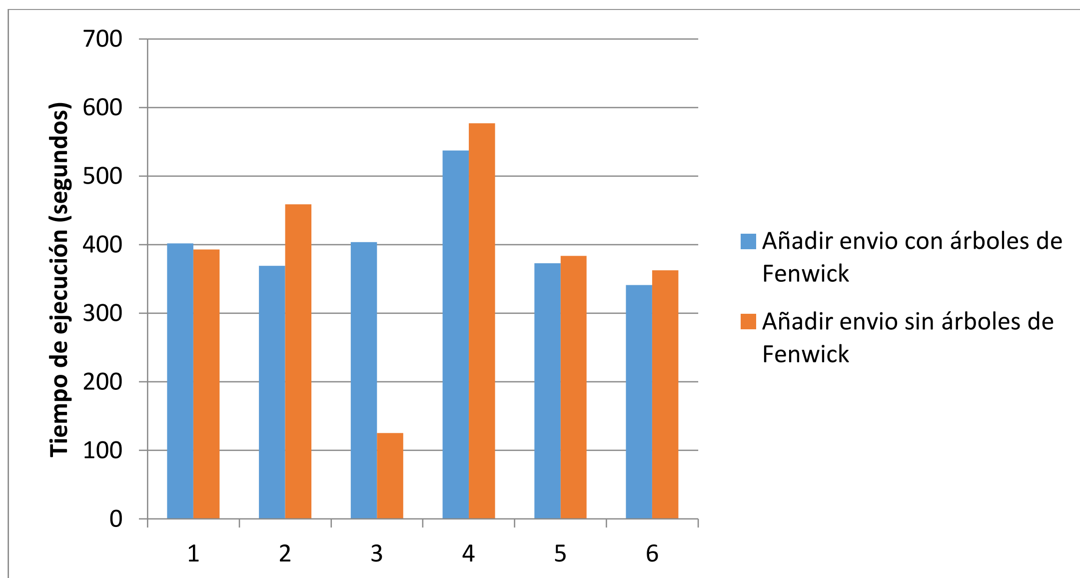
El objetivo del trabajo era la implementación del framework que potenciara la interacción social entre los usuarios. Sin embargo, dicho framework no está integrado en la aplicación web de '¡Acepta el reto!' al no estar implementada la parte del frontend. El alcance del proyecto sólo cubría la parte del backend, saliéndose el frontend del alcance del proyecto. Dedicar tiempo a su implementación hubiera afectado muy negativamente al desarrollo del backend. Por ello, ha sido necesario demostrar la validez del trabajo creado. Dado que la razón principal de la realización del proyecto fue mejorar la eficiencia, vamos a realizar diversas pruebas para demostrar la mejoría en los tiempos de ejecución.

9.1. Estudio de la eficiencia de los árboles de Fenwick

Los árboles de Fenwick, como hemos visto, son una estructura que se aloja en memoria RAM y realiza copias de seguridad a la base de datos permanente. Además, la estructura se actualiza en tiempo real, lo que consumirá tiempo de ejecución. Vamos a realizar una comparativa entre los tiempos de ejecución de realización de nuevos envíos, corrección de envíos y consulta de posición en tiempo real.

Las pruebas se dividirán en dos partes. En la primera, con la base de datos vacía y los árboles de Fenwick operativos, se realizarán 7000 envíos, se corregirán y se realizarán 100.000 consultas a los árboles. En la segunda, se borrarán los contenidos de la base de datos para dejarla vacía de nuevo y se volverán a realizar las mismas operaciones pero tal y como se hacían antes de este proyecto: se desactivarán los árboles de Fenwick y se volverán a realizar 7000 envíos, se corregirán y se realizarán las 100.000 consultas de nuevo, pero esta vez consultando la base de datos y realizando un filtrado en la tabla de envíos. Se han medido los tiempos de ejecución de cada una de las 6 operaciones y se han comparado dos a dos.

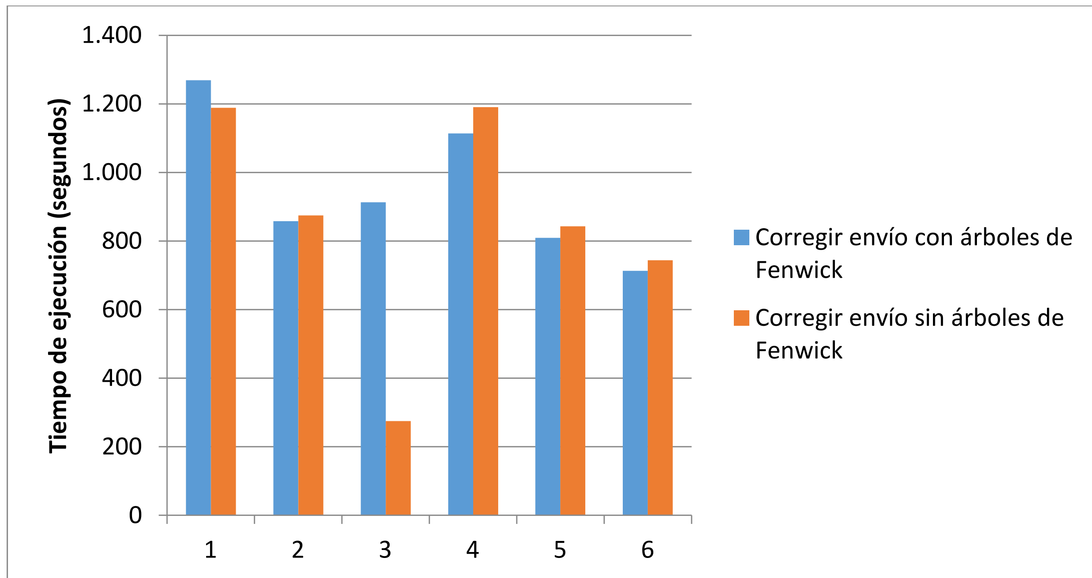
9.1.1. Inserción de nuevos envíos



En la gráfica superior se observa el tiempo de ejecución en segundos requerido para realizar 7000 envíos nuevos, tanto con árboles de Fenwick como sin ellos. Exceptuando una de las ejecuciones, en la que se han obtenido datos anómalos, se observa que el tiempo de ejecución

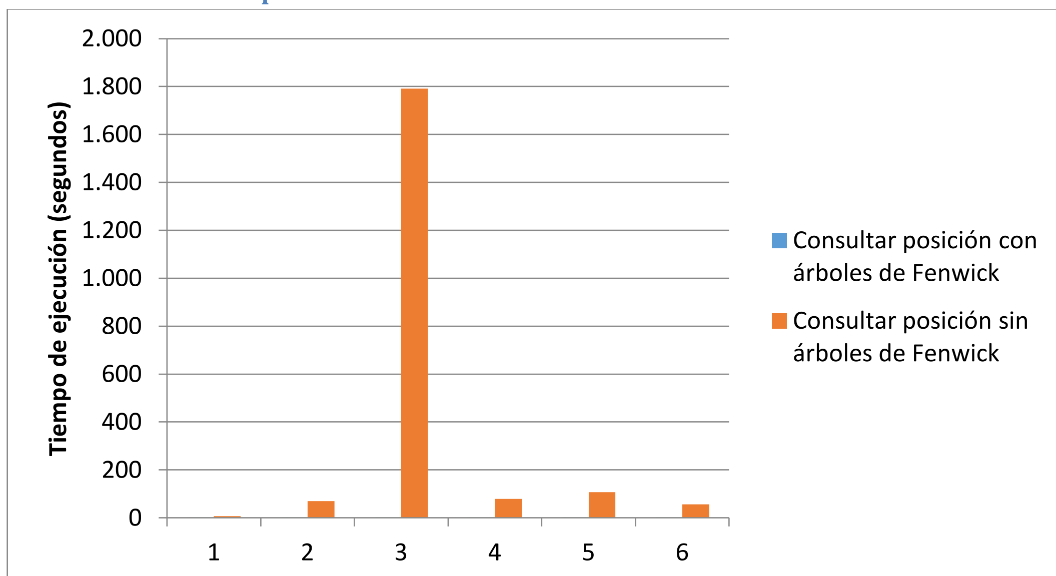
es muy parecido, pudiendo achacarse las variaciones al distinto rendimiento de la máquina entre ambas partes de la prueba. Realmente, al realizar un envío no se utilizan los árboles de Fenwick, por lo que el resultado cumple con lo esperado.

9.1.2. Corrección de envíos



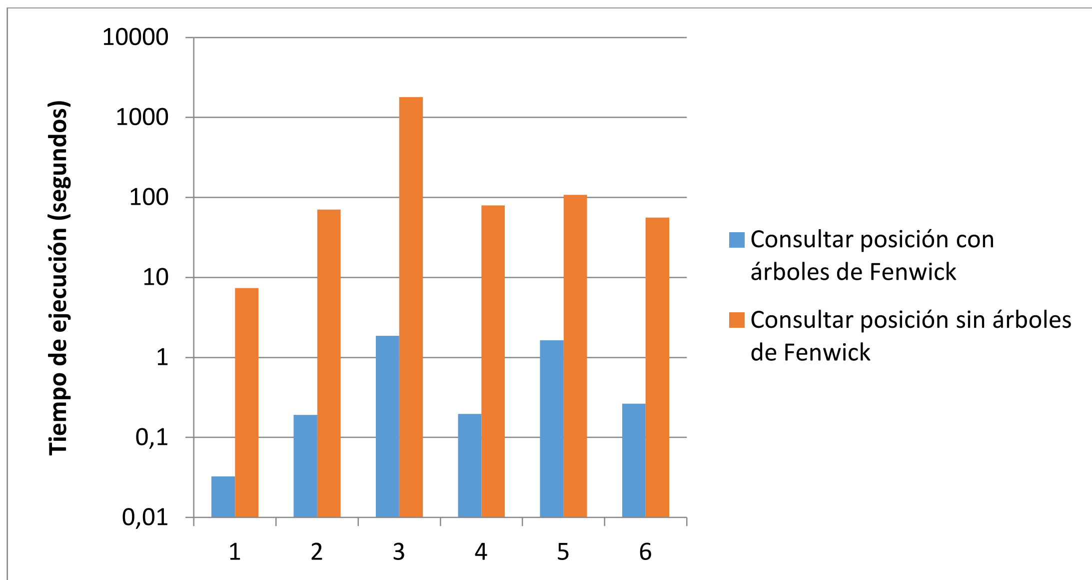
En la gráfica superior se observa el tiempo de ejecución en segundos requerido para corregir los 7000 envíos anteriores, tanto con árboles de Fenwick como sin ellos. Exceptuando una de las ejecuciones, en la que se han obtenido datos anómalos, se observa que el tiempo de ejecución es muy parecido, incluso algo superior cuando no se utilizan los árboles de Fenwick. Esto posiblemente se deba a la escasa carga de trabajo que representan los árboles de Fenwick y a la diferencia de rendimiento de la maquina entre ambas partes de la prueba.

9.1.3. Consulta de posiciones



La grafica anterior representa el tiempo de ejecución en segundos requerido para realizar 100.000 consultas de posiciones de envíos, usando los árboles de Fenwick o realizando una consulta a la tabla de envíos. En los resultados obtenidos el tiempo de ejecución usando los

árboles ha dividido el tiempo de ejecución entre 65 y 960 veces según la ejecución, respecto al tiempo obtenido utilizando la base de datos. Para una mejor apreciación, se muestran los mismos resultados con una escala logarítmica:



9.1.4. Conclusiones

Analizando los datos obtenidos, observamos que los árboles de Fenwick mejoran significativamente la eficiencia. Su actualización en tiempo real durante la corrección de los envíos presenta una carga de tiempo mínima, mientras que se acelera de forma muy considerable el tiempo de consulta.

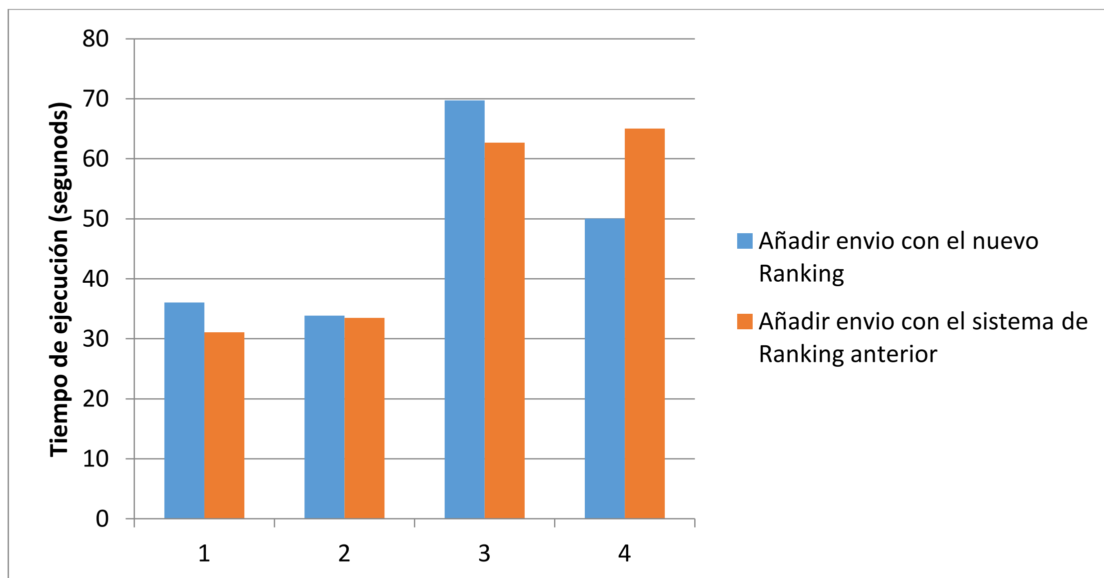
9.2. Estudio de la eficiencia del sistema de rankings

Los rankings, al contrario que los árboles de Fenwick, son una estructura que se aloja en la base de datos permanente, haciendo un uso provisional de la memoria. Las actualizaciones se realizan en tiempo real, con la posibilidad de requerir consultas a la base de datos, lo que consumirá tiempo de ejecución. De igual manera, vamos a realizar una comparativa entre los tiempos de ejecución entre el sistema de rankings antiguo y el actual para las mismas tres operaciones: realización de nuevos envíos, corrección de envíos y consulta de rankings.

Las pruebas se estructurarán de igual manera que con los árboles de Fenwick, pero se han realizado con cantidades algo menores. En la primera, con la base de datos vacía y el nuevo sistema de rankings operativo, se realizarán 700 envíos, se corregirán y se realizarán 10.000 consultas de rankings. En la segunda, se borrarán los contenidos de la base de datos para dejarla vacía de nuevo y se volverán a realizar las operaciones tal y como se hacían antes de este proyecto: se desactivará el nuevo sistema de rankings y se volverán a realizar 700 envíos, se corregirán y se realizarán las 10.000 consultas de nuevo, pero esta vez creando el ranking concreto desde cero a partir de todos los envíos. Se han medido los tiempos de ejecución de cada una de las 6 operaciones y se han comparado dos a dos.

Aunque se han realizado las pruebas de esta forma para poder hacer una comparación de eficiencia, el sistema antiguo de rankings realmente no funciona de la manera que se especifica. En la implementación anterior, el ranking consultado no se actualiza en cada consulta, sino que periódicamente (cada 10-15 minutos) se actualizan todos los rankings de todos los problemas, creándolos a partir de todos los envíos de la base de datos. Hay que tener en cuenta este detalle para la interpretación de los resultados. En estas pruebas no se está valorando si el sistema propuesto es más rápido que el actualmente implementado en la plataforma, sino que se está valorando si el sistema propuesto sería más eficiente que el sistema de la plataforma si este se utilizara para obtener rankings en tiempo real.

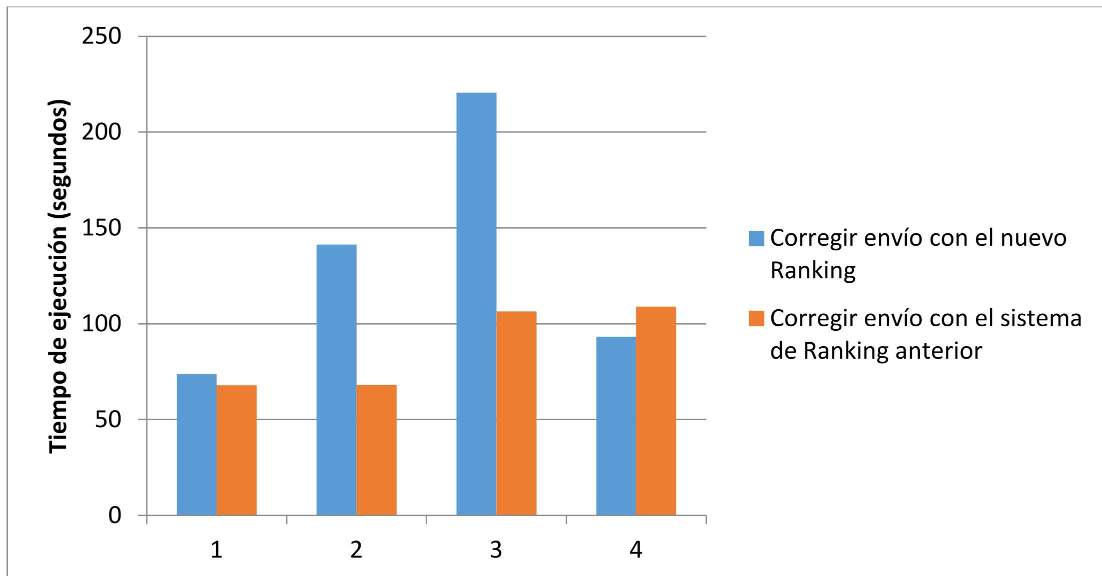
9.2.1. Inserción de nuevos envíos



En la gráfica superior se observa el tiempo de ejecución en segundos requerido para realizar 700 envíos nuevos, tanto con el nuevo sistema de rankings como sin él. En general, se observa

la diferencia entre los resultados es pequeña, y las variaciones entre estas diferencias es arbitraria y dependiente más de las condiciones de la máquina que del procedimiento utilizado. Es por ello que el nuevo sistema puede ser viable a gran escala en un sistema real. Realmente, al realizar un envío no se utilizan el sistema de rankings, por lo que el resultado cumple con lo esperado.

9.2.2. Corrección de envíos

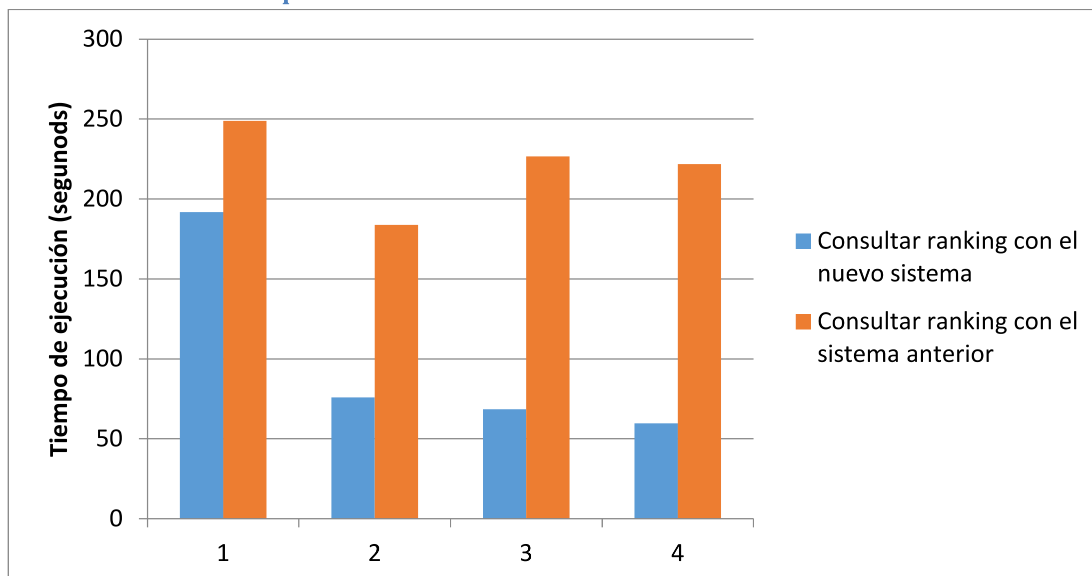


En la gráfica superior se observa el tiempo de ejecución en segundos requerido para corregir los 700 envíos anteriores, tanto con el nuevo sistema de rankings como sin él. Aunque los resultados son dispares, en las pruebas realizadas el tiempo de ejecución con el nuevo ranking como mucho duplica el tiempo de ejecución usando el ranking antiguo.

Hay que tener en cuenta que el funcionamiento de los nuevos rankings es distinto al del sistema de árboles de Fenwick. Los árboles de Fenwick se cargan en memoria al arrancar la aplicación, quedando disponibles en la memoria RAM indefinidamente. Sin embargo, los rankings se cargan de la base de datos a demanda, y se borran de la memoria RAM cada vez que se hace una copia de seguridad. Por ello, es comprensible la penalización que se obtiene en este caso al corregir un envío, al ser necesario realizar una consulta a la base de datos que con los árboles de Fenwick no era necesaria.

Dado el tráfico actual de la plataforma, no se requerirán muchas actualizaciones de rankings y, y por tanto, no se requerirán muchas consultas a la base de datos permanente. Teniendo en cuenta esto, la penalización obtenida en la corrección de envíos puede ser asumible para el sistema.

9.2.3. Consulta de posiciones



En la gráfica superior se observa el tiempo de ejecución en segundos requerido para realizar 10.000 consultas de posiciones de envíos, tanto con el nuevo sistema de rankings como actualizando el ranking concreto en la base de datos. Aunque los datos son dispares, se puede observar que la consulta tradicional puede llegar casi a cuadruplicar el coste de la consulta usando el nuevo sistema. La diferencia de tiempos obtenida entre ambos sistemas no es tan drástica como la obtenida con los árboles de Fenwick, pero sí existe una mejora en la eficiencia en la consulta.

9.2.4. Conclusiones

La comparación de la eficiencia real entre los dos sistemas no es fácilmente comparable, ya que las funcionalidades que ofrecen son distintas. En las pruebas realizadas se observa un aumento de la eficiencia en la consulta de los rankings con el nuevo sistema. Por el contario, el aumento en la eficiencia en las consultas se ve ligeramente mermado por la penalización en la corrección. Aunque las pruebas se han hecho con un número significativo de datos y consultas, es muy probable que no se corresponda con la carga de trabajo de la plataforma y la distribución de dicho trabajo en el tiempo. Para obtener unos datos mucho más fiables, sería necesario comprobar en la plataforma real su eficiencia y como resulta el equilibrio entre la mejora y la penalización en la eficiencia entre ambas operaciones.

En cualquier caso, el nuevo sistema podría beneficiar a la plataforma. Esta implementación permitiría a los usuarios consultar el ranking y ver resultados que sean un fiel reflejo del estado de la plataforma en el momento, y no un reflejo de un estado anterior calculado previamente. También se elimina la actualización periódica de todos los rankings que, dado el número actual de envíos y problemas en la plataforma, consume una cantidad considerable de tiempo. Además, se mitigaría la carga de trabajo de la plataforma en la consulta masiva de rankings por los usuarios en un periodo de tiempo corto, por lo que en periodos de alto tráfico (como un concurso de programación) no tenía por qué ser necesario desactivarlos para evitar colapsar el servidor. Por último, dado que los rankings se encuentran en la base de datos la mayoría del tiempo, el nuevo sistema no ocuparía demasiada memoria.

10. Conclusiones

Con el proyecto terminado se puede afirmar que se han cumplido los objetivos. Se ha aumentado ligeramente la interacción social entre los usuarios en el backend, y se ha mejorado la eficiencia y las funcionalidades que ofrecen los rankings y otros sistemas asociados. Además, el desarrollo es mantenible y escalable, permitiendo ampliaciones o mejoras en el futuro. También el nuevo código cuenta con una alta cohesión y un bajo acoplamiento, lo que facilita su modificación o eliminación, minimizando el esfuerzo y tiempo que estas tareas requerirían.

Se ha obtenido un sistema eficiente en memoria, pero especialmente en tiempo. Dicho sistema se encuentra probado y listo para integrarse en el backend de '¡Acepta el reto!'. Los árboles de Fenwick y los rankings además funcionan de forma autónoma y transparente para los usuarios y administradores, siendo capaces de manejar sucesos excepcionales como las reevaluaciones.

Mediante la aplicación del sistema y la siguiente validación, consecuencia de las simulaciones realizadas sobre datos reales de la página web, podemos afirmar que el proyecto se encuentra totalmente operativo y listo para ser integrado en la página web. Dicha integración únicamente requeriría una ampliación en el frontend para ofrecer las funcionalidades de 'Me gusta' y 'Seguir' a los usuarios, y para poder consultar los nuevos rankings. Los árboles de Fenwick y la mejora en los rankings existentes son transparentes también al frontend, por lo que no requieren modificaciones en este.

10.1. Valoración personal

Estoy satisfecho con la realización de este proyecto, ya que me ha permitido aplicar mis conocimientos a un sistema real. Esto incluye el hecho de buscar y pensar soluciones factibles y eficientes a problemas reales, como por ejemplo la ineficiencia del sistema de rankings, que resultaba un punto crítico de la aplicación.

Ha sido interesante el desarrollo de los árboles de Fenwick y de los rankings, ya que me han permitido aplicar conocimientos de *Estructuras de datos y algoritmos*, analizando la eficiencia de diversas estructuras de datos para resolver un problema concreto obteniendo buenos resultados en el uso de tiempo y memoria.

Finalmente, puedo afirmar que se han aplicado principios básicos de *Ingeniería del Software* al obtener una aplicación mantenible, escalable, con alta cohesión y bajo acoplamiento.

10.2. Trabajo futuro

Durante la realización del proyecto se han puesto de manifiesto una serie de opciones que puede ser interesante integrar en el futuro:

- Integración del sistema de 'Me gusta' y seguidores en el frontend de la web, de forma que sus funcionalidades sean accesibles a los usuarios.
- Integración del feed en el frontend de la web.
- Creación de un sistema de mensajes privados o chat, de manera que los usuarios puedan comunicarse entre sí de manera confidencial.

- Creación de un sistema de comentarios para los problemas que permita a los usuarios colgar mensajes públicos relacionados con la resolución de un problema.
- Creación de un sistema de pistas, de manera que usuarios que hayan fallado la respuesta puedan recibir ayuda del juez o de otros usuarios que tuvieron problemas similares.
- Integración de '¡Acepta el reto!' en las distintas redes sociales como Facebook o Twitter.
- Integración de los nuevos rankings en el frontend, de manera que los usuarios puedan consultar la clasificación de los problemas filtrada por lenguaje.
- Integración de las nuevas funcionalidades que ofrece los árboles de Fenwick en el frontend, de manera que los usuarios puedan consultar en tiempo real la posición que ocupan sus envíos.

10. Conclusions

Now that the project is finished, we can confirm that all goals have been achieved. Social interaction among users has increased in the backend, efficiency has been improved and new functionalities have been added to the rankings and other systems associated with it. Also, the development has archived scalability and maintainability, allowing improvements and expansions in the future. Moreover, all the new code has high cohesion and low coupling, which makes modification or removal much easier, minimizing the amount of time and effort required to do so.

The result system is memory efficient, but more important, time efficient. This system has been tested and is ready to be integrated in '¡Acepta el reto!' backend. Fenwick trees and rankings also work autonomously and are transparent to both users and administrators, and are able to manage exceptional issues like reevaluations.

Because of this system application and latter validation, consequence of the simulations performed using real data from the platform, we can confirm that this project is fully operational and is ready to be integrated in '¡Acepta el reto!'. Such integration would only require an extension in the frontend in order to offer Like and Follow functionalities to users, and in order to allow users access to language-filtered rankings. Fenwick trees and ranking optimization are transparent also to the frontend, requiring no modifications.

10.1. Personal assessment

I'm satisfied with the development of this project, because it has allowed me to apply my knowledge to a real system. This includes thinking and searching for feasible and efficient solutions to a real problem, like the current ranking system inefficiency, which was a critical point of the web application.

Developing Fenwick trees and rankings has been interesting because I've been able to apply my knowledge in *Algorithm and Data Structures*, analyzing the efficiency of diverse data structures in order to solve a concrete problem, obtaining in the end good marks in time and memory use.

Finally, I can confirm that basic *Software Engineering* principles have been applied in order to obtain a maintainable and scalable application that satisfies high cohesion and low coupling.

10.2. Future work

During the development of this project, some other improvement options have come up, in order to be implemented in the future:

- Integration of Like and Follow systems in the web frontend, so their functionalities are accessible to users.
- Integration of Feed in the web frontend.
- Development of a chat or private message system, so users can communicate confidentially among them.
- Development of a comment system for problems, so user can post public messages about its resolution.

- Development of a clue system, so users can get help from the judge or other users about a problem resolution when the answer is wrong.
- Integration of social networks like Twitter or Facebook into '¡Acepta el reto!'.
- Integration of language-specific rankings into the frontend, so users can look it up.
- Integration of Fenwick trees new functionalities into the web frontend, so users can look up their submissions position in the ranking in real time.

Bibliografía

1. Uva Online Judge: Juez Online de la Universidad de Valladolid. [Online] Disponible en: https://uva.onlinejudge.org/

2. Universidad de Valladolid. [Online] Disponible en: http://www.uva.es/export/sites/uva/

3. Sphere Online Judge. [Online] Disponible en: http://www.spoj.com/

4. Jutge.org. [Online] Disponible en: https://jutge.org/

5. Universitat Politècnica de Catalunya. [Online] Disponible en: https://www.upc.edu/

6. Olimpiada Informática Española. [Online] Disponible en: https://olimpiada-informatica.org/

7. Ranking. [Online] Disponible en: https://en.wikipedia.org/wiki/Ranking

8. Preorden total. [Online] Disponible en: https://en.wikipedia.org/wiki/Weak_ordering#Total_preorders

9. Relación de equivalencia. [Online] Disponible en: https://en.wikipedia.org/wiki/Equivalence_relation

10. Orden estricto total. [Online] Disponible en: https://en.wikipedia.org/wiki/Total_order#Strict_total_order

11. Botón de 'Me gusta'. [Online] Disponible en: https://en.wikipedia.org/wiki/Like_button

12. FriendFeed. [Online] Sitio oficial cerrado. Información disponible en: https://es.wikipedia.org/wiki/FriendFeed

13. Facebook. [Online] Disponible en: https://www.facebook.com/

14. Botón de 'Me gusta' (Facebook). [Online] Disponible en:https://en.wikipedia.org/wiki/Facebook_like_button

15. Twitter. [Online] Disponible en: https://twitter.com/

16. '¡Acepta el reto!' [Online] Disponible en: https://www.aceptaelreto.com/

17. ProgramaMe: Concurso de programación para ciclos formativos. [Online] Disponible en: https://www.programa-me.com/2017/reg/

18. Primer concurso navideño de entretenimiento del CEEDCV. [Online] Disponible en: https://programa-me.com/2017/reg/navidad/

19. Centre Específic D'Edicació a Distància de la Comunitat Valenciana. [Online] Disponible en: https://ceedcv.org/

20. Veredictos posibles - '¡Acepta el reto!' [Online] Disponible en: https://www.aceptaelreto.com/doc/verdicts.php

21. PHP. [Online] Disponible en: https://secure.php.net/
22. Hypertext Markup Language - Wikipedia. [Online] Disponible en: https://en.wikipedia.org/wiki/HTML
23. Javascript. [Online] Disponible en: https://www.javascript.com/
24. MySQL. [Online] Disponible en: https://www.mysql.com/
25. Hibernate. [Online] Disponible en: https://hibernate.org/
26. Java. [Online] Disponible en: https://www.java.com/
27. Apache Tomcat. [Online] Disponible en: https://tomcat.apache.org/
28. Apache Software Foundation. [Online] Disponible en: https://www.apache.org/
29. Árbol de Fenwick. [Online] Disponible en: https://en.wikipedia.org/wiki/Fenwick_tree
30. Árbol binomial. [Online] Disponible en: https://letslearncs.com/binomial-trees/