Universidad de Valladolid

Facultad de Ciencias

TRABAJO FIN DE GRADO

Grado en Estadística

Solución a un problema de Rendimiento en MySQL

Optimización de UVa Online Judge

Autor:Sara Esteban Mínguez

Tutor/es: Miguel Ángel Revilla Ramos

Cesar González Ferreras

ÍNDICE

1.- RESUMEN **3**
2.- ABSTRACT **4**
3.- INTRODUCCIÓN **5**
3.1 HISTORIA 5
4.- OBJETIVO DEL PROYECTO **8**
5.- IDENTIFICACIÓN DEL PROBLEMA **9**
5.1 NAVEGACIÓN CATEGORIZADA POR LA BASE DE DATOS DE PROBLEMAS 9
5.2 ESTADÍSTICAS POR PROBLEMA 11
5.3 ENVÍOS DE RESOLUCIÓN DE PROBLEMAS 12
5.4 RANKINGS Y ESTADÍSTICAS 13
6.- OPTIMIZACIÓN DE BASE DE DATOS MYSQL **14**
6.1 BASES DE DATOS RELACIONALES 14
6.2 NORMALIZACIÓN DE LA BASE DE DATOS 14
6.3 MECANISMOS DE ALMACENAMIENTO DE DATOS 16
6.4 INDICES 17
6.5 GESTIÓN DE ÍNDICES 18
6.6 TAREAS DE AJUSTE SQL 19
6.6.1 QUERY CACHE 19
6.6.2 ESTADISTICAS 21
6.6.3 IDENTIFICACIÓN DE SQL LENTAS 22
6.7 OPTIMIZADOR SQL 24
6.7.1 OPTIMIZADOR BASADO EN COSTOS 25
6.7.2 INTERPRETAR PLANES DE EJECUCIÓN 25
6.8 HERRAMIENTAS DE BENCHMARKING 27
6.9 CURSORES Y PROCEDIMIENTOS ALMACENADOS 29
7.- APLICACIÓN CASO PRÁCTICO **30**
7.1 OPTIMIZACIÓN DE CONSULTAS 30
7.1.1 ANALIZAR EXPLAIN PLAN 31
7.1.2 CONCLUSIONES 35
7.2 OPTIMIZACIÓN DE OTROS OBJETOS DE LA BASE DE DATOS 36
7.2.1 PROCEDIMIENTO "actualiza_estadisticas_usuario" 40
7.2.2 PROCEDIMIENTO "actualiza_estadisticas_problema" 41
7.2.3 PROCEDIMIENTO " actualiza_ranking " 42
7.3 RESULTADO 43
8.- VERIFICACION DE RESULTADOS **45**
8.1 HERRAMIENTA SHOW STATUS 45
8.2 HERRAMIENTA MYSQLSLAP 48
9.- CONCLUSIONES **54**
10.- BIBLOGRAFIA **56**

1.- RESUMEN

El presente trabajo se enmarca en el campo de la adecuada gestión de datos para la optimización de un proyecto de formación asistida por ordenador. Sobre la base de que el proceso de aprendizaje debe hacerse de forma automática, los usuarios necesitan poder acceder al gran volumen de información que se acumula en el momento preciso y de la forma más ágil y rápida posible.

Tras dar una visión completa de los aspectos implicados en el trabajo de optimización de bases de datos relacionales y más concretamente del gestor de base de datos MySQL, se procede a analizar el efecto sobre el rendimiento a través de distintas herramientas.

Para ilustrar todo el trabajo utilizamos un caso real, un juez automático que está funcionando en la Universidad de Valladolid (UVa) desde hace casi 20 años y ha recibido más de 16 millones de programas informáticos para comprobar si resuelven o no alguno de los 4500 problemas planteados.

Desde el punto de vista institucional, el juez automático UVa, es una marca líder reconocida como una de las más activas y de mayor alcance internacional entre los participantes en concursos de programación. Actualmente su desarrollo es un proyecto propio de la Fundación General de la Universidad de Valladolid.

2.- ABSTRACT

The current work deals with the subject of proper data management in the context of a computer-aided training project. Based on the principle that the learning process is an automated task, the users need a real-time, fast and agile access to a great volume of cumulated information.

After providing a comprehensive view of the more significant aspects related to the optimization of relational databases and, more specifically, the MySQL database manager, we proceed to evaluate the effect on the system's performance through a variety of analysis tools.

We have illustrated our work through a real use case: an automated judge that is currently running on the University of Valladolid (Uva) since 20 years ago, and who has received more than 16 million of computer programs with the aim of testing whether they are capable of solving any of the more than 4.500 problems raised.

From an institutional point of view, the automated judge UVa is a leading brand, recognized as one of the most active and presenting one of the largest international reach among the participants in programming competitions. Its development is a current project of the General Foundation of the University of Valladolid.

3.- INTRODUCCIÓN

A día de hoy, el denominado 'online judge' de la Universidad de Valladolid (UVa-OJ) es un servicio ofrecido a través de Internet para toda la comunidad académica, y aunque en un principio fue creado para proporcionar a los usuarios una forma de preparación para los diversos concursos de informática y programación que existen en el mundo, actualmente es una herramienta muy completa de aprendizaje asistido por ordenador en el campo de las ciencias de la computación.

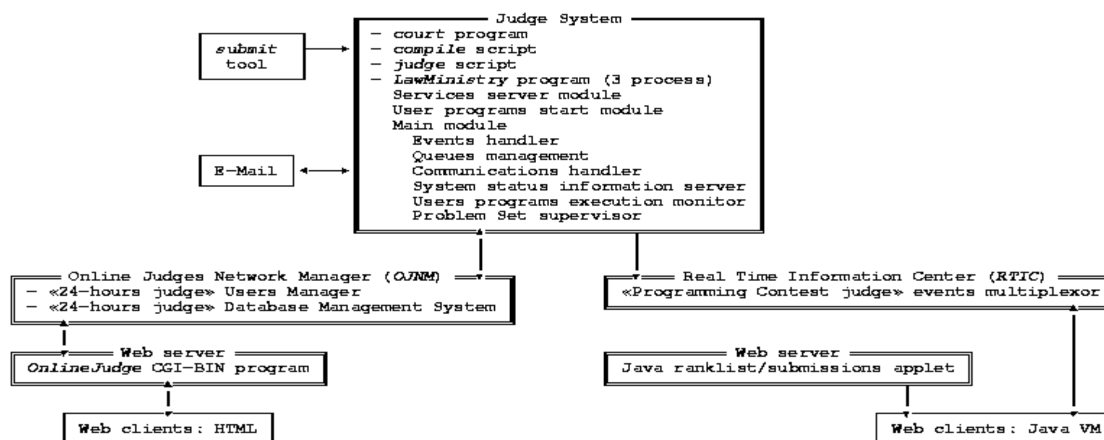
Es decir, esencialmente es una aplicación web para practicar las habilidades de programación y los conocimientos de algoritmos (esencialmente matemáticos) en un entorno competitivo para estudiantes universitarios. Esto permite no sólo el autoaprendizaje, sino también el aprendizaje colaborativo con miles de personas de todos los países del mundo.

La popularidad actual del sistema a nivel mundial es detectable por cualquiera de los medios que el mundo de las comunicaciones nos ofrece. El UVa-OJ es, casi con seguridad, el primer 'sitio' web que existió con estas prestaciones y, pese a la abundante competencia actual, sigue siendo el principal referente en la comunidad internacional en el campo de la educación en informática y algoritmos.

3.1 HISTORIA

Desde la creación del juez han cambiado, como es lógico, muchas cosas por no decir todas. Pero el principal cambio se dio en el año 2007 cuando se cambió el almacenamiento de los envíos y se comenzó a utilizar la base de datos MySQL. Los cambios anteriores habían sido de imagen, de consolidación del juez frente a los ataques de hackers, la incorporación de nuevas herramientas e información, pero todo el progreso estaba lastrado por la capacidad para manejar la información.

El diseño original, era esencialmente, el que puede verse en el siguiente esquema:



La parte RTIC no se llegó a desarrollar nunca, porque era muy problemático dar una información en tiempo real por diversos motivos: la velocidad de la máquina, la velocidad de la red y sobre todo la dificultad del acceso a los datos de una forma segura.
Lo que hacía de 'base de datos' era un fichero binario donde cada envío se almacenaba en 20 bytes que una vez descomprimidos contenían la siguiente información (aquí están los 10 primeros envíos de la historia y los 10 últimos de la vieja plataforma):

Event-ID	YYYYMMDDhhmmssmmm	ST	USER	PROBL	SOURCE	MAXMEM	CPU	SG	ALGORITH
0	19970415143148000	AC	1000	100	C	392	2.453	0	None
1	19970415160440000	SE	1001	?	C	0	0.000	0	None
2	19970415160531000	RE	1001	108	C	0	0.020	11	None
3	19970415161102000	RE	1001	108	C	0	0.010	11	None
4	19970416160933000	WA	1001	100	C	388	2.367	0	None
5	19970416161547000	WA	1001	100	C	388	5.646	0	None
6	19970418090211000	WA	1001	100	C	384	5.630	0	None
7	19970418090254000	WA	1001	100	C	392	5.623	0	None
8	19970430031739000	AC	1000	100	C	388	2.406	0	None
9	19970506102716000	CE	1002	138	C	0	0.000	0	None
5899114	20070906171446400	PE	37046	488	C++	388	2.047	0	None
5899115	20070906171514000	WA	50251	10035	C++	0	0.047	0	None
5899116	20070906171535400	AC	48210	841	C++	1096	0.264	0	None
5899117	20070906171600400	TL	54908	100	JAVA	2548	10.025	9	None
5899118	20070906171617100	TL	52686	11124	C	4428	10.018	9	None
5899119	20070906171707000	WA	26416	142	C++	404	0.121	0	None
5899120	20070906171708500	AC	47505	11049	JAVA	0	0.027	0	None
5899121	20070906171717300	RE	44804	11192	C++	0	0.002	11	None
5899122	20070906171900100	AC	48210	841	C++	1096	0.270	0	None
5899123	20070906171943000	PE	37046	488	C++	392	2.041	0	None

El primer límite que se daba era el de la cantidad de usuarios. Los 2 bytes reservados para contener esta información limitaban a 65536 los posibles usuarios. Como creció enseguida, se contuvo con métodos diversos hasta que no quedó más remedio que cerrar la inscripción.
Esta situación tan absurda obligó a incorporar una base de datos relacional, con el consiguiente proceso de migración desde un formato primario a una base de datos relacional.

Desde la creación de la plataforma actual se siguen modificando muchas cosas, porque tanto los servidores como los servicios web y la forma en que los usuarios interaccionan con la información y con otras personas, ha ido cambiando de forma paulatina pero imparable. Ya que a medida que aumenta la velocidad de proceso o la facilidad de uso, las demandas de los usuarios son mayores y es por ello por lo que actualmente se está desarrollando una nueva plataforma

Mientras tanto, las principales dificultades han venido de la gestión de la base de datos para intentar que su rendimiento sea óptimo. Cierto es que el cambio de la velocidad de proceso como el de las comunicaciones (actualmente el juez está en la nube) han disimulado estos problemas que nunca llegaron a ser tan críticos como en el pasado. Pero el valor del material ya existente es tan grande que cualquier aportación, por pequeña que sea, multiplica sus efectos de forma espectacular. Por eso se fueron añadiendo nuevas tablas y la base de datos se fue complicando sin recurrir a ninguna reconstrucción global para sacar el máximo provecho del carácter relacional de MySQL.

De hecho se tuvo que recurrir a desarrollos externos para paliar algunas de las deficiencias más importantes:

- El acceso a estadísticas personalizadas se hace por medio de la herramienta uHunt que también proporciona un servicio de 'chat'.
- Como una alternativa a la información binaria (acierto o fallo de los códigos que envían los usuarios en su intento de resolver los tests planteados) surge uDebug, que permite tantear con datos propios de cada usuario el nivel de corrección de su programa. Esto es un avance fundamental.
- El acceso desde cualquier entorno, desde la línea de comando a extensiones en el Chrome, y cualquier dispositivo móvil han sido abordados también por muchos programadores externos.

En resumen, la llegada masiva de las redes sociales, las nuevas formas, tanto tecnológicas como conceptuales, de acceso a la Red, y las expectativas de los usuarios ante su participación en un servicio web obliga a buscar un nuevo concepto de juez en línea. El usuario ya no quiere ser un usuario en el sentido estricto de la palabra. Ya no se conforma únicamente con recibir información. La nueva generación busca participar en la creación de esos contenidos, en pos de una mayor visibilidad social. Hoy en día, la web se ha convertido en una meritocracia, más aún en un entorno competitivo como el UVa Online Judge. La gente quiere ser vista, independientemente de que tal atención sea merecida o no, pero eso abre nuevas vías de colaboración. Pero todas ellas pasan por el hecho de poder acceder a los datos que se necesitan en el momento preciso, de manera rápida y ágil.

4.- OBJETIVO DEL PROYECTO

El objetivo principal del presente proyecto, es conseguir disponer de la información tanto de los rankings, como de las estadísticas de los problemas y los usuarios en tiempo real, sin necesidad de esperas, en base al ajuste de MySQL y a la optimización tanto de las consultas como de los procedimientos que se utilizan para realizar dichos cálculos.

Consiguiendo de ésta manera, que la comunidad de estudiantes que utiliza la herramienta del juez online, disponga de toda la información necesaria de manera instantánea, generando en dicha comunidad una mayor vinculación con el proyecto UVa online judge

5.- IDENTIFICACIÓN DEL PROBLEMA

Para poder identificar el problema que se nos plantea, lo primero que vamos a realizar, es una visión rápida de lo que permite la herramienta del juez alojada en https://uva.onlinejudge.org/

Ésta herramienta Online permite:

- Navegación Categorizada por la base de datos de problemas
- Estadísticas por problema y usuario
- Envíos de códigos(en cinco lenguajes de programación) para resolver los problemas
- Rankings para los problemas y para los usuarios en distintos periodos de tiemp

Veámoslo más en detalle

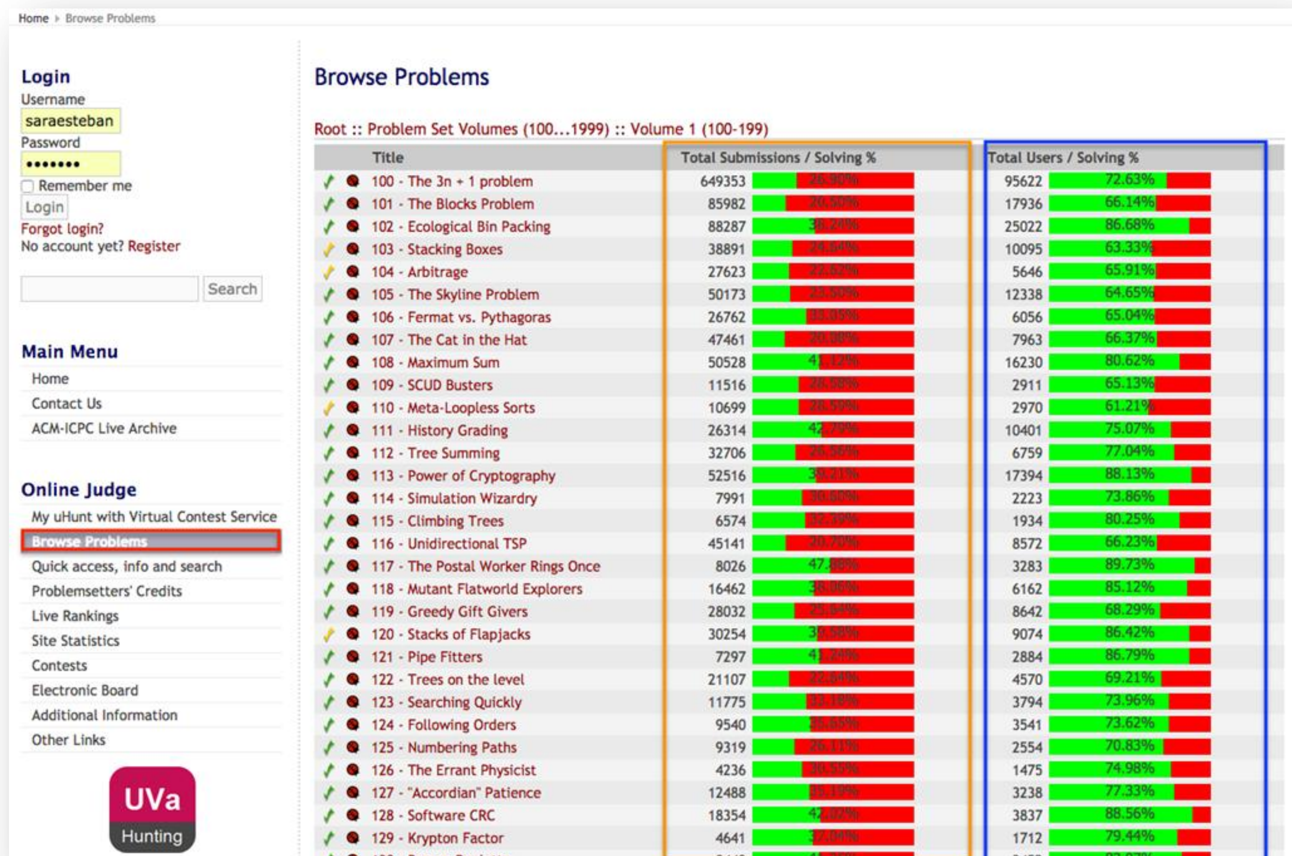
5.1 NAVEGACIÓN CATEGORIZADA POR LA BASE DE DATOS DE PROBLEMAS

Para acceder a la base de datos de problemas accederemos a la opción de menú que mostramos a continuación

Podremos hacer búsquedas de problemas que aparecen categorizadas y podremos navegar por ellas como se muestra a continuación

5.2 ESTADÍSTICAS POR PROBLEMA

Una vez hemos accedido a la categoría de problemas que nos interesa veremos lo siguiente



Y aquí podremos elegir entre los cientos de programas que hay en las categorías, como observamos, a primera vista ya se puede observar la dificultad de cada una de los problemas, en la columna con recuadro naranja nos da el valor en porcentaje de los envíos correctos, respecto del total de envíos realizados, mientras que en la columna azul, nos da información en porcentaje de los usuarios que han resuelto el problema respecto del total de usuarios que han enviado la solución

5.3 ENVÍOS DE RESOLUCIÓN DE PROBLEMAS

Una vez hemos seleccionado el problema que queremos resolver tendré que acceder al menú Quick Submit y enviar la información tal y como mostramos a continuación

De ésta manera hemos enviado la solución al problema planteado. Esto lo podremos hacer cuantas veces nos parezca, y podremos enviar soluciones a todos los problemas que nos encontramos en la base de datos.

Como vemos esto nos sirve de entrenamiento, pero, **¿Cómo de buenos o de malos somos en la resolución de estos problemas?**

5.4 RANKINGS Y ESTADÍSTICAS

Para poder medirnos respecto del resto de personas que están practicando, podemos acceder al menú estadísticas que se observa en las figuras posteriores y navegar entre las tres posibilidades que se nos plantean:

- Últimos 50 envíos
- Estadísticas por Usuario
- Estadísticas anuales

Y precisamente , ésta es la sección que genera que una base de datos no optimizada, pueda generar que la aplicación pueda no llegar a funcionar , debido a que consultas y procedimientos "pesados" puedan demorar tanto la respuesta que puedan provocar fallos de "Timeout" en la aplicación del juez online.

Hay que pensar que estamos sacando rankings y estadísticas sobre tablas que en algunos casos poseen varios millones de registros, por lo que para que la base de datos responda en milisegundos, requisito necesario para un funcionamiento usable y correcto de la aplicación, es necesario tener la base de datos optimizada.

Una vez hemos centrado el problema, tener una base de datos optimizada, vamos a pasar a plantear las soluciones al problema planteado.

6.- OPTIMIZACIÓN DE BASE DE DATOS MYSQL

MySQL es un sistema de gestión de bases de datos relacional, multihilo y multiusuario. La gestión y optimización de los accesos a disco en MySQL viene determinado por su arquitectura, en la que el motor de almacenamiento es un elemento independiente y transparente a los usuarios. Existen distintos motores, lo más utilizados son MyISAM e InnoDB.

6.1 BASES DE DATOS RELACIONALES

Las bases de datos relacionales se basan en el uso de tablas (relaciones). Las tablas se representan como una estructura formada por filas y columnas.
Cada columna almacena información sobre una propiedad determinada de la tabla (llamada atributo).
Cada fila posee una ocurrencia o ejemplar de la relación representada por la tabla.
El modelo relacional de base de datos se basa en un modelo formal especificado de acuerdo a la teoría de conjuntos. Una base de datos relacional puede considerarse como un conjunto de relaciones o tablas de la forma R(A1,...,An) donde R es el nombre de la relación, que se define por una serie de atributos Ai.

El primer paso para el diseño de una base de datos relacional (diseño conceptual) es hacer un modelo entidad relación.
Se trata de un diagrama que pretende representar las entidades relevantes de un sistema de información así como sus interrelaciones y propiedades.

El diseño lógico parte del diseño conceptual y da como resultado una descripción de la base de datos en términos de estructuras de datos que puede procesar un tipo de SGBD, en el caso relacional, consiste en definir las tablas que existirán, las relaciones entre ellas.

6.2 NORMALIZACIÓN DE LA BASE DE DATOS

Para explicar el proceso de normalización es necesario saber que sobre las tablas relacionales se pueden definir diferentes restricciones. La integridad de entidad es una restricción que nos indica que cada entidad representada por una tupla tiene que ser diferente de las demás en su relación, es decir debe haber algunos atributos cuyos valores identifiquen unívocamente las tuplas.

Estos atributos componen lo que se llama clave primaria y su valor no puede ser nulo. La integridad referencial indica que una clave ajena solo debe contener valores que o bien sean nulos o bien hagan referencia a valores de una clave primaria.

El fin de la normalización en una base de datos es mejorar la integridad de los datos a través de la minimización de la redundancia y la inconsistencia.

Las formas normales nos permiten garantizar que el esquema es conforme con el modelo relacional. En teoría existen cinco formas normales, pero en la práctica sólo se aplican las tres primeras.

La aplicación de formas normales requiere una comprensión perfecta del concepto de dependencia funcional. Un dato depende funcionalmente de otro si el conocimiento del segundo permite determinar el valor del primero.

- Primera forma normal (1FN): se dice que una tabla está en primera forma normal si todas las columnas contienen valores simples. Ejemplo: si una tabla de clientes contiene un campo teléfonos en el que se guardan los diferentes números de teléfono de un cliente, esta tabla no está en primera forma normal, habrá que definir las columnas trabajo y móvil para estructurar mejor los datos.

- Segunda forma normal (2FN): se dice que una tabla está en segunda forma normal si está en primera forma normal y todas las columnas no claves dependen funcionalmente de la primera.

- Tercera forma normal (3FN): se dice que una tabla está en tercera forma normal si está en segunda y no existe dependencia funcional entre dos columnas no claves.

La creación de tablas debe parecerse lo más posible al modelo de datos y normalizarse hasta la 3FN. Aunque a veces nos puede interesar la desnormalización de forma selectiva para mejorar el rendimiento, se debe estudiar la posibilidad de desnormalizar algunas tablas de modo que se evite el uso excesivo de joins. Una única tabla también nos va a permitir estrategias de indexación más eficientes.

Un buen modelo de datos supone que las consultas se escriben de forma más eficaz.

La creación de índices se puede dejar para más adelante, aunque en un principio claves primarias, únicas y ajenas se pueden crear en esta fase.

6.3 MECANISMOS DE ALMACENAMIENTO DE DATOS

MySQL es una base de datos muy rápida en la lectura cuando utiliza el motor no transaccional MyISAM, pero puede provocar problemas de integridad en entornos de alta concurrencia en la modificación.

InnoDB es un mecanismo de almacenamiento de datos de código abierto para la base de datos MySQL, incluido como formato de tabla estándar en todas las distribuciones de MySQL AB a partir de las versiones 4.0. Su característica principal es que soporta transacciones de tipo ACID y bloqueo de registros e integridad referencial. InnoDB ofrece una fiabilidad y consistencia muy superior a MyISAM, la anterior tecnología de tablas de MySQL, si bien el mejor rendimiento de uno u otro formato dependerá de la aplicación específica.

A la hora de escoger entre MyISAM e InnoDB como tecnología de almacenamiento de datos en MySQL mostramos algunas de las diferencias entre los dos:

- InnoDB se recupera de un problema volviendo a ejecutar sus logs, mientras que MyISAM necesita repasar todos los índices y tablas que hayan sido actualizados y reconstruirlos si esos cambios no han sido escritos en disco. El primer proceso requiere más o menos el mismo tiempo siempre, mientras que el segundo aumenta con el tamaño de la base de datos.
- MyISAM deja al sistema operativo la tarea de hacer la caché de las lecturas y escrituras de los registros, mientras que InnoDB realiza él mismo la tarea, combinando cachés de registro y de índice. InnoDB no envía directamente los cambios en las tablas al sistema operativo para que las escriba, lo que puede hacerlo mucho más rápido que MyISAM en ciertos escenarios.
- InnoDB almacena físicamente los registros en el orden de la clave primaria, mientras que MyISAM los guarda en el orden en que fueron añadidos. Cuando la clave primaria se escoge de acuerdo con las necesidades de las consultas más habituales esto puede suponer una mejora sustancial del rendimiento. Por otro lado, si los datos se insertan en un orden que difiera sustancialmente del orden de la clave primaria, se obliga a InnoDB a reordenar mucho los datos para mantenerlos en el orden adecuado.
- InnoDB no dispone de la compresión de datos de la que disfruta MyISAM, de modo que tanto el espacio en disco como la caché en la memoria RAM pueden ser más grandes. Este problema se ha reducido en MySQL 5.0, reduciéndolo en aproximadamente un 20%.
- Cuando opera con transacciones ACID, InnoDB debe escribir en disco al menos una vez por cada transacción, aunque puede combinar las escrituras de varias inserciones concurrentes. Para los discos duros típicos, esto supone un límite de aproximadamente 200 transacciones por segundo, por lo que aumentarlas exige controladores de disco con caché de escritura y sistema de alimentación ininterrumpido para mantener la integridad. InnoDB ofrece diversos modos de funcionamiento que reducen este efecto, pero conllevan una pérdida de integridad transaccional. MyISAM no tiene ese problema porque no soporta transacciones.

6.4 INDICES

Un índice es un objeto de la base de datos, una estructura independiente de los datos de la tabla que nos proporciona una ruta de acceso más rápida a los datos de las tablas.

La creación de un índice mejora la operación de recuperación de datos pero penaliza las operaciones de actualización e inserción de datos, por lo que son críticos para un buen rendimiento.

La forma más fácil de entender cómo funciona un índice en MySQL es pensar en un índice de un libro. MySQL utiliza el índice de una manera similar. Un índice contiene los valores de una columna especificada o columnas de una tabla, si el índice tiene más de una columna, el orden de las columnas es muy importante, ya que solo puede buscar de manera eficiente en un prefijo más a la izquierda del índice.

Los tipos de índice según su estructura de almacenamiento pueden ser :

- Índices B-Tree: Es un índice con estructura de árbol, formado por un conjunto de nodos cada uno de los cuales contiene valores de índice ordenados que apuntan a registros de datos en disco. Los árboles balanceados tienen la característica de que cualquier búsqueda tiene el mismo tiempo de resolución. Este tipo de índices facilitan la resolución de consultas basadas en búsquedas exactas por índice, rangos que usan cláusulas como between y operadores de comparación. Debido a que los nodos del árbol están ordenados ,pueden ser utilizados para búsqueda de valores y de la misma manera ayudar a ordenar las filas por los mismos criterios

- Hash: Un índice hash se construye sobre una tabla hash y es útil solo para búsquedas exactas que utilizan cada una de la columnas del índice para cada fila, el motor de almacenamiento calcula un código hash de las columnas indexadas para cada fila, así el índice se compone del código hash y un puntero a cada fila de la tabla hash. Estos índices son muy rápidos, pero no pueden evitar la lectura de la tabla ya que solo guardan punteros a la fila. No permiten la búsqueda parcial de las columnas que forman el índice ya que la función hash se crea para todo el índice conjunto. Solo soporta operaciones de igualdad.

- R-Tree: Índices utilizados para datos espaciales y multidimensionales.

Los índices se implementan en la capa de motor de almacenamiento, no la capa de servidor. Por lo tanto, no están normalizados. La indexación funciona de forma ligeramente diferente en cada motor, y no todos los motores admiten todos los tipos de índices.
Para el motor InnoDB solo están disponibles los índices B-Tree.

Los índices pueden ser:

- Índices de Única Columna: están formados por una columna.
- Índices Compuestos o Concatenados: están formado por varias columnas, no necesariamente adyacentes.

6.5 GESTIÓN DE ÍNDICES

Para la mejora en el rendimiento de una base de datos, deberemos seguir las siguientes reglas de indexación:

- Sobre las restricciones de clave ajena, para reducir el tiempo de respuesta en las uniones entre las tablas de clave primaria y ajena
- Sobre las columnas a las que se accede frecuentemente.
- Cuando los valores en la columna son relativamente únicos
- Cuando queramos recuperar con frecuencia menos del 15% de las filas de una tabla de gran tamaño.
- Para mejorar el rendimiento de las uniones de varias tablas, haremos un índice por las columnas que utilicemos para la unión.
- Si la columna contiene valores nulos, pero las consultas normalmente seleccionan todas las filas que tienen un valor.
- Las columnas con selectividad deficiente se pueden combinar para formar un índice compuesto mejorando así la selectividad. Además si todas las columnas seleccionadas en el SELECT forman parte del índice, el sistema puede devolver estos valores sin acceder a la tabla reduciendo así la E/S.
- Un índice compuesto se usará cuando en la cláusula WHERE se hace referencia a toda o a la parte principal de las columnas del índice.

Además de éstas reglas se crearán índices automáticamente sobre las claves primarias y las claves únicas.

Hay que hacer destacar que el orden de las columnas en un índice compuesto puede afectar al rendimiento de la consulta, se deben especificar primero las columnas que se utilizan con más frecuencia, así la primera columna del índice será el campo más usado o el más selectivo.

En una búsqueda parcial de columnas sobre índices compuestos, éstos índices no serán útiles si la búsqueda no coincide con el lado izquierdo de las columnas indexadas, es decir si el índice está compuesto por 3 columnas, no se usará el índice si se busca por la segunda y la tercera, aunque sí será eficiente si hacemos una búsqueda por la primera y la segunda.

Es importante definir columnas como NOT NULL, no se emplearan índices si en el where existen restricciones IS NULL o IS NOT NULL.

Aunque se pueden crear múltiples índices en una tabla, MySQL solo utilizará un único índice para acceder a la tabla, por lo que la sobrecarga de índices en una tabla puede ralentizar la velocidad de actualización de los datos.

6.6 TAREAS DE AJUSTE SQL

Una vez hemos normalizado nuestra base de datos, elegido el motor de almacenamiento e indexado correctamente la base de datos, vamos a proceder a realizar otras tareas de ajuste SQL importantes para hacer que nuestra base de datos funcione de manera óptima.

Para ello realizaremos las siguientes tareas:

- Activar QUERY CACHE
- Lanzar Estadísticas
- Identificar las SQL lentas

6.6.1 QUERY CACHE

"Query Cache" es una funcionalidad de MySQL que almacena en memoria el contenido y resultado de una consulta tipo SELECT, de manera que cuando un cliente vuelva a ejecutar la misma consulta esta no tendrá que procesarse y se servirá directamente de la memoria.

Para activar la caché de consultas lo indicaremos de la siguiente manera, modificando el siguiente parámetro de la base de datos:

```
query_cache_type=1
```

El parámetro podrá tener los siguientes valores:

- query_cache_type=0 indicará que la cache está desactivada
- query_cache_type=1 indicará que la cache está activada
- query_cache_type=2 indicará que la cache está según demanda, es decir hay que especificarlo en la consulta SELECT /* SQL_CACHE */: y con el parámetro query_cache_size=28M

Pero hay que tener una serie detalles en cuenta al hacer uso de esta cache:

- Para mantener la consistencia en los resultados guarda una relación de las tablas a las cuales afectan la query, de forma que si una de esas tablas se ve modificada, la Query Cache se invalida.
- La query se guarda tal y como la hemos escrito, esto es, para la Cache no sería lo mismo "SELECT Nombre,Apellido from tabla where id=2" que "SELECT nombre,apellido from Tabla where id=2".
- No se guardará el resultado de la query cuando no sea determinista. Por lo tanto, funciones como now(), current_date(), etc… no son candidatas a ser cacheadas.
- La Query Cache añade overhead. Las queries de lectura deben leer o escribir de la cache y las de escritura deben borrar las caches asociadas a las tablas modificadas.
- Durante una transacción, las caches asociadas a las tablas modificadas no se podrán usar hasta que no se haga commit o rollback.

Podemos controlar como de eficiente es nuestra Cache mirando el estado de algunas variables de MySQL. Cuando una query se responde de la cache, el valor que aumenta es Qcache_hits. Por el contrario, si la query debe ejecutarse por no estar previamente cacheada, aumentaremos en uno el valor de Com_select. Por lo tanto, para conocer el ratio de acierto podemos hacer uso de esta fórmula:

```
Qcache_hits / (Qcache_hits+Com_select)
```

Otra variable importante a tener en cuenta es:

```
Qcache_lowmem_prunes
```

Que es un contador del número de queries sacadas de la caché por falta de memoria. Si el valor de esta variable sube demasiado a lo largo del tiempo, hay que pensar en aumentar el tamaño de la caché.

Otros valores que debemos configura en el archivo de configuración my.cfg son los siguientes:

- query_cache_size: es el tamaño de la caché. Debe ser un valor múltiplo de 1024 bytes
- query_cache_min_res_unit: MySQL guarda las cachés en la memoria en pequeños bloques, como si de un sistema de ficheros se tratase. En un principio no sabe realmente el tamaño que va a tener el resultado de una query, por lo que según va enviando las filas al cliente, va cogiendo bloques. Los bloques tendrán como mínimo el tamaño aquí indicado. Este valor es importante para evitar la fragmentación de la memoria y así aprovecharla lo mejor posible
- query_cache_limit: si un resultado supera el tamaño aquí indicado, no se cacheará. Pero recordad lo comentado en el punto anterior, MySQL no sabe a priori cuando ocupará, por lo que igualmente irá reservando bloques hasta llegar al query_cache_limit, momento en el cual los bloques escritos se liberarán de nuevo

Para evitar la fragmentación hay que elegir un valor correcto. Este no debe ser muy pequeño, (ya que aunque evitaremos perder memoria, MySQL tendrá que ir cogiendo bloques constantemente), ni muy grande (ya que si nuestras queries devuelven resultados pequeños tendremos bastante fragmentación).

Una de las formas de tener un valor cercano al óptimo es saber la media de tamaño de las queries almacenadas. Para ello podemos aplicar la siguiente fórmula:

```
(query_cache_size – Qcache_free_memory) / Qcache_queries_in_cache
```

Esto es, la memoria usada entre el número de queries en caché.

Para saber si realmente tienes la caché muy fragmentada, puedes hacerte una idea comprobando el número de bloques libres

```
Qcache_free_blocks
```

Si el valor de esta variable está cercana a Qcache_total_blocks / 2, entonces tienes un grave problema de fragmentación.

MySQL también tiene su propio desfragmentador, FLUSH QUERY CACHE moverá todos los bloques utilizados eliminando los espacios en blanco entre ellos.

6.6.2 ESTADISTICAS

Una vez hemos activado la cache, y antes de empezar a analizar las sql que están deteriorando el rendimiento, lo que tenemos que hacer es lanzar las estadísticas, para que le demos herramientas al servidor para que pueda optimizar los accesos a las consultas y reduzcamos los tiempos, para ello haremos las siguientes tareas:

- OPTIMIZE TABLE
- ANALYZE TABLE

Con el comando ANALIZE se generan las estadísticas que se almacenan en el diccionario de datos.

```
ANALYZE TABLE nombre_tabla;
```

Actualiza la información sobre la distribución de los valores de la tabla. Esto nos servirá para que el optimizador pueda tomar decisiones sobre el plan de ejecución de las queries.

6.6.3 IDENTIFICACIÓN DE SQL LENTAS

Para Identificar las SQL lentas que tienen mayor carga, MySQL nos ofrece las siguientes opciones

- Activar el SLOW QUERY LOG
- Usar SHOW PROCESSLIST

Activar SLOW QUERY LOG supone que en el momento que una consulta SQL supere el tiempo de ejecución establecido con la variable long_query_time quedará registrada en el log.

Para activarlo añadimos en el fichero my.cnf las siguientes líneas:

```
log-slow-queries=/var/log/mysql-slow-queries.log

long_query_time = 5

log-queries-not-using-indexes
```

log-slow-queries es el fichero en el que se grabarán las consultas que superen el long_query_time , en este caso 5 segundos, log-queries-not-using-indexes lo activamos si queremos logear las consultas que no utilicen índices.

Por otro lado si lo que queremos ver en tiempo real (porque en ese momento estamos viendo una caída clara del rendimiento del servidor), cuales son las consultas o los procesos que se están ejecutando en ese momento utilizaremos el siguiente comando

```
SHOW FULL PROCESSLIST
```

```
SHOW FULL PROCESSLIST;

+---------+-------+-----------+------+---------+------+-------+-----------------------+

| Id      | User  | Host      | db   | Command | Time | State | Info                  |

+---------+-------+-----------+------+---------+------+-------+-----------------------+

| 1988880 | demo | localhost | NULL | Query   |    0 | NULL  | SHOW FULL PROCESSLIST |
```

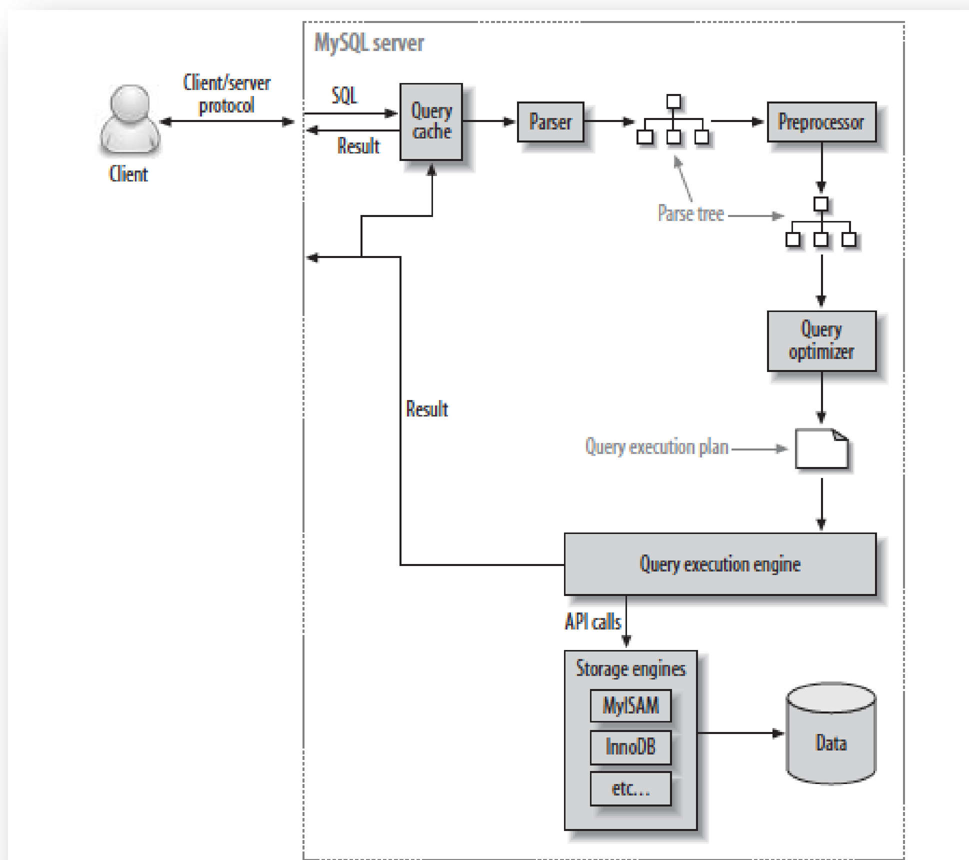
NOMBRE	DESCRIPCION
ID	El ID del proceso del cliente
USER	El nombre del usuarrio que lo esta lanzando
HOST	El equipo del cliente que lo esta lanzando
DB	La base de datos propietaria del proceso
COMMAND	El tipo de comando que se esta ejecutando
TIME	El tiempo que ha transcurrido desde que se ha lanzado el proceso
STATE	El estado del servidor
INFO	La sentencia SQL que esta siendo ejecutada

6.7 OPTIMIZADOR SQL

Para poder mejorar el rendimiento de una consulta debemos conocer cómo trabaja MySQL en el proceso de resolución de consultas.

MySQL sigue el siguiente proceso:

1- MySQL ejecuta un algoritmo de hash sobre la consulta y mira a ver si tiene alguna otra idéntica en su cache, si es así devuelve el resultado que ya tenía.
2- Parseado: si no está cacheada, la parsea. El parseo tiene como objetivo comprobar la corrección sintáctica de la consulta y dividirla en sus elementos más básicos.
3- Planificación: ahora debe decidir cómo va a resolver la consulta.
4- Ejecución: ejecuta el plan diseñado.



MySQL utiliza un optimizador basado en costos para determinar la mejor manera de resolver una consulta. Para ello se basa en la cláusula WHERE, en las estadísticas, indicaciones (hints), e informaciones del esquema.

El optimizador se basa principalmente en el uso de las estadísticas, estas describen la base de datos y los objetos que ella contiene. Como los objetos de una base de datos cambian constantemente, las estadísticas se deben actualizar regularmente, sino es así el optimizador puede seleccionar planes erróneos.

6.7.1 OPTIMIZADOR BASADO EN COSTOS

La técnica del optimizador de consultas, utiliza un algoritmo basado en estadísticas acerca de los objetos de la base de datos.
El optimizador de consultas intenta determinar qué plan de ejecución es más eficaz basándose en estadísticas de tablas o índices.

En particular, el optimizador MySQL decide usar un índice basándose en el porcentaje de datos que espera recuperar .Opta por utilizar un índice para pequeños intervalos mientras que si el intervalo es grande decide un escaneo completo de la tabla

Los pasos que sigue son los siguientes:

- El optimizador genera los posibles planes para la sentencia SQL
- El optimizador estima el costo de cada plan basado en las estadísticas del diccionario de datos, los compara y selecciona el de menor costo.
- Si la estadística cambia, el optimizador adapta su plan de ejecución , por eso es importante que las estadísticas estén actualizadas

6.7.2 INTERPRETAR PLANES DE EJECUCIÓN

El planificador tiene que tomar una decisión muy rápida sobre el mejor camino a seguir, el plan de ejecución es la salida del optimizador, el camino que va a seguir para recuperar los datos de la consulta.
La sentencia EXPLAIN nos muestra la siguiente información:

- El método por el que se va a acceder a las tablas referenciadas en la consulta
- Las operaciones de datos como ordenación, filtrado o agregación
- Información sobre el costo que va a suponer

```
EXPLAIN SELECT columna1 FROM TABLA WHERE columna2=valor
```

EXPLAIN devuelve un registro con:

COLUMNA	EXPLICACIÓN
id	Identifica cada consulta de la consulta
select-type	Puede ser simple, primary, union, dependent union, subselect y derived
table	nombre de la tabla de la que MySQL leerá los registro
type	Indica el tipo de join .Puede ser const, system, eq-ref, ref, range, index y all.
Possible keys	Lista de índices que pueden ser usados
Key	Nombre del índice que usará para resolver la consulta. (sólo uno)
key_len	Tamaño del índice en bytes
Ref	Las columnas que se utilizarán en el índice
Rows	Estimado de filas que considera que tendrá que examinar para resolver la consulta.
Extra	Información adicional

El orden en el que aparecen los registros, después de ejecutar la sentencia explain, indica el orden en el que MySQL utilizará las tablas. Esto es muy importante para el caso de los joins.

Por otro lado hay que destacar que también contamos con herramientas para controlar el comportamiento del optimizador, como es el caso de HINTS, a través de los cuales se puede especificar que camino queremos forzar a seguir, veamos ejemplos de cómo podemos usarlo.

```
SELECT * FROM tabla USE INDEX (idx_col1)  WHERE col1=1 AND col2=2;
```

En este caso estaría sugiriendo que cogiera el índice idx_col1 para resolver la consulta.

Mientras que si queremos forzar el uso del índice idx_col3 ejecutaríamos el siguiente comando.

```
SELECT * FROM tabla FORCE INDEX (idx_col3)  WHERE col1=1 AND col2=2 AND col3=3;
```

El caso opuesto sería el siguiente:

```
SELECT * FROM tabla IGNORE INDEX (idx_col3)  WHERE col1=1 AND col2=2 AND col3=3;
```

6.8 HERRAMIENTAS DE BENCHMARKING

MySQL integra una utilidad llamada **mysqlslap** que permite **estresar** (es decir simular una gran carga en la base de datos debido a usuarios simultáneos conectados, o bien a lanzar un gran volumen de sentencias sql, o bien a la conjunción de ambas cosas) **y hacer benchmarking del servidor MySQL**.

Gracias a ésta herramienta, podremos comparar el rendimiento del servidor ante cambios realizados en las bases de datos, motores de almacenamiento y configuraciones así como con X número de clientes interactuando con las bases de datos, cierto tipo de consultas SQL, etc.…

Si queremos comenzar haciendo test básicos, podemos dejar que sea la propia utilidad la que **genere de forma aleatoria las consultas SQL**, únicamente tenemos que especificar el usuario MySQL con el cual se van a realizar:

```
$ mysqlslap -u demo -pxxxx --auto-generate-sql
Benchmark
            Average number of seconds to run all queries: 0.007 seconds
            Minimum number of seconds to run all queries: 0.007 seconds
            Maximum number of seconds to run all queries: 0.007 seconds
            Number of clients running queries: 1
            Average number of queries per client: 0
```

En este test se ha realizado la conexión de un único usuario (demo) que ha realizado pruebas aleatorias entre las que se incluye la creación de una tabla, la inserción y consulta de datos en ella y la eliminación de la tabla. Si queremos ver con detalle cada una de las consultas que se realizan en el test se puede añadir la cantidad de verbose con el siguiente parámetro

```
$ mysqlslap -vvv -u demo -pxxxx --auto-generate-sql
```

Estos test realmente no sirven de mucho, así que conviene saber cómo personalizarlos, añadir más usuarios concurrentes en las pruebas y consultas SQL personalizadas. El número de usuarios concurrentes se especifica con el parámetro --concurrency, vamos a realizar la misma prueba pero con 150 usuarios concurrentes:

```
$ mysqlslap -u demo -pxxxx –concurrency 150 --auto-generate-sql
Benchmark
            Average number of seconds to run all queries: 0.389 seconds
            Minimum number of seconds to run all queries: 0.389 seconds
            Maximum number of seconds to run all queries: 0.389 seconds
            Number of clients running queries: 150
            Average number of queries per client: 0
```

Otro parámetro interesante es –iterations xxx , mediante el cual podemos especificar el número de veces que se repetirá el test.

```
$ mysqlslap -u demo -pxxxx –concurrency 150 –iterations 4 --auto-generate-sql
Benchmark
            Average number of seconds to run all queries: 0.389 seconds
            Minimum number of seconds to run all queries: 0.389 seconds
            Maximum number of seconds to run all queries: 0.389 seconds
            Number of clients running queries: 150
            Average number of queries per client: 0
```

También podemos modificar el número de consultas a realizar en el test con el parámetro --number-of-queries XXX que serán repartidas entre el número de usuarios que ejecutan el test:

```
$ mysqlslap -u demo -pxxxx –concurrency 150 –iterations 4 --number-of-queries 3000 --auto-generate-sql
Benchmark
            Average number of seconds to run all queries: 0.233 seconds
            Minimum number of seconds to run all queries: 0.217 seconds
            Maximum number of seconds to run all queries: 0.242 seconds
            Number of clients running queries: 150
            Average number of queries per client: 20
```

Otro parámetro de extrema utilidad es –engine , que nos permite especificar el tipo de motor a utilizar para la creación de la tabla (MyISAM, Innodb…) y así poder realizar varios test y verificar cual es el más adecuado.

```
$ mysqlslap -u demo -pxxxx –concurrency 150 –iterations 4 --number-of-queries 3000 --engine=innodb --auto-generate-sql
Benchmark
            Running for engine innodb
            Average number of seconds to run all queries: 1.873 seconds
            Minimum number of seconds to run all queries: 1.724 seconds
            Maximum number of seconds to run all queries: 2.197 seconds
            Number of clients running queries: 150
            Average number of queries per client: 20
```

Para poder personalizar al máximo el test de benchmarking conviene poder especificar la creación de tablas y consultas a realizar.

Con –create especificamos la creación de la tabla y con –query   la consulta a realizar.

```
$ mysqlslap --delimiter=";" \
  --create="CREATE TABLE TABLA (COLUMNA1 int);INSERT INTO TABLA VALUES (23)" \
  --query="SELECT * FROM TABLA" --concurrency=50 --iterations=200
Benchmark
            Average number of seconds to run all queries: 0.038 seconds
            Minimum number of seconds to run all queries: 0.011 seconds
            Maximum number of seconds to run all queries: 0.061 seconds
            Number of clients running queries: 50
            Average number of queries per client: 1
```

Como hemos podido observar, esta es una potente herramienta que nos da MySQL para poder hacer pruebas de rendimiento al nivel que queramos, debemos destacar que este tipo de herramientas los debemos usar en entornos de pruebas para que podamos estresar la base de datos todo lo que queramos, sin que afecte a entornos productivos que puedan provocar una parada del sistema.

6.9 CURSORES Y PROCEDIMIENTOS ALMACENADOS

Los procedimientos almacenados son compilados en la bases de datos, y esto hace que sean más rápidos, ya que un procedimiento almacenado compilado se analiza una sola vez, mientras que una las sentencia sql se analiza cada vez que se lanzan.

Hay que tener especial cuidado que los planes de ejecución no sean anticuados.
Por ejemplo si se añade un nuevo índice, si no recompilamos el procedimiento, el plan de ejecución de este no lo tendrá en cuenta, y no lo tendremos optimizado.

Una buena práctica es dividir el procedimiento en varios subprocedimientos, con el fin de que si hay un cambio en la estructura de una tabla referenciada en el procedimiento, solo se recompile él.

Otras de las ventajas de usar procedimientos son:

- Código reutilizable
- Código más sencillo y fácil de modificar
- Reducir el tráfico de red
- Uso de cursores
- Pool compartido

El objetivo debe ser minimizar el análisis para ello las sentencias SQL se deberían analizar una vez y ejecutarlas varias veces. Para ello usamos cursores, en el desarrollo de la aplicación hay que tener cuidado también en que se comparta el pool compartido, para ello se usan variables de enlace y no literales de cadena para representar las partes de la consulta que cambian de una ejecución a otra.

Si usamos variables de enlace, el optimizador reconocerá que ya ha analizado la sentencia y decidirá usar el mismo plan de ejecución aunque los valores de enlace sean diferentes, sin embargo si se utilizan literales en la consulta, el optimizador utilizará los valores del literal para elegir el mejor plan.

Usando código genérico, procedimientos almacenados y triggers de la base de datos conseguiremos que todo funcione mejor.

7.- APLICACIÓN CASO PRÁCTICO

Una vez ya conocida la parte teórica de cómo podemos optimizar una base de datos MySQL, vamos a proceder a aplicarlo en la aplicación del juez online.

7.1 OPTIMIZACIÓN DE CONSULTAS

Partimos de la base que por defecto MySQL, utiliza herramientas intrínsecas que le permiten analizar, en base a los índices que tenga creados, cual es el camino más optimo para devolver datos de la consulta que le estemos realizando al servidor. Como es obvio no siempre acertará en el 100% de los casos, y es por ello que también nos provee de otros mecanismos para algunos casos en los que observemos que el rendimiento no es el optimo podamos modificar dicho comportamiento.

En primer lugar, hemos obtenido la siguiente consulta que a priori tiene un comportamiento al menos aceptable:

```
select count(distinct userid)from oj_onlinejudge_submisionns where problemid=36
```

si analizamos el tiempo que el servidor tarda en resolver la consulta, observamos que la devuelve en 0.362 segundos , a primera vista podríamos pensar que esta consulta no debería generar un problema de rendimiento y podríamos darla por válida, pero nos encontramos con la situación de que en la aplicación del juez online, esta consulta podría llegar a ser generada múltiples veces en un brevísimo espacio de tiempo, por lo que suponiendo que en un momento dado tuviéramos 100 ejecuciones simultaneas el tiempo de espera pasaría a ser :

```
T=tiempo de espera
n=consultas simultaneas
t=tiempo de ejecución de la consulta
T= n * t = 100 * 0.362 = 36.2 segundos
```

Por lo que esto, sí que empieza a ser un verdadero problema de rendimiento del que nos tenemos que ocupar.

7.1.1 ANALIZAR EXPLAIN PLAN

Vamos a ejecutar el comando explain para poder analizar el comportamiento de la base de datos cuando se encuentra con la petición que estamos trabajando

```
explain select count(distinct userid)from oj_onlinejudge_submisionns where problemid=36;
```

Como vimos en la teoría este comando nos devuelve el siguiente resultado

Columna	Resultado
Id	simple
Select_type	oj_onlinejudge_submisionns
type	All
Posible_keys	null
Key	null
Key_len	null
Ref	null
Rows	9963443
extra	where

Y ahora es el momento de interpretar lo que nos está explicando MySQL.

Si la columna type no nos proporciona un buen tipo de acceso (en este caso nos está devolviendo el valor all, que indica que está haciendo un full table scan) la mejor manera de resolver el problema es por lo general mediante la adición de un índice apropiado

Adicionalmente en esta salida de Explain vemos que MySQL tiene que recorrer toda la tabla para ofrecernos la información que le estamos pidiendo, es decir para devolvernos que esta selección de registros contiene 7375 registros, tiene que leer 9963443 registros, por lo que obviamente es totalmente ineficiente ya que el ratio de filas examinadas a filas devueltas es muy grande.

```
Ratio= 9963443 / 7375 =1350.97
```

Otro indicador que nos puede dar una pista de que es necesario crear un índice es cuando en la columna extra aparece “using where”. El uso de “Using where” en la columna extra nos dice que está usando la clausula WHERE para descartar filas una vez que el motor de almacenamiento ya las ha leído.

MySQL tiene 3 maneras de resolver una cláusula WHERE, que son las siguientes de mejor a peor:

- Aplicar la condición en la operación de búsqueda del índice para eliminar las no coincidentes. Esto ocurre en la capa de motor de almacenamiento.
- Utilizar un índice de cobertura ("using index" en la columna extra) para evitar accesos a filas y filtrar las filas no coincidentes después de recuperar cada resultado del índice. Esto sucede en la capa de servidor, pero no requiere la lectura de filas de la tabla, solo del índice.
- Y la última y la peor, recuperar las filas de la tabla y luego filtrar las no coincidentes

Después de analizar la salida de explain que nos ha devuelto el servidor, optamos por la creación de un nuevo índice, para ver si de esta manera mejoramos el tiempo de resolución.

Tras analizar la estructura de la tabla, así como la consulta observamos que un posible candidato para ser índice, podría ser el siguiente:

```
create index problemid on oj_onlinejudge_submisionns (problemid)
```

Una vez creado este índice, volvemos ejecutar el comando explain

```
explain select count(distinct userid)from oj_onlinejudge_submisionns where problemid=36;
```

con el siguiente resultado

Columna	Resultado
Id	simple
Select_type	oj_onlinejudge_submisionns
type	ref
Posible_keys	problemid
Key	problemid
Key_len	4
Ref	const
Rows	73050
extra	null

Para este nuevo plan de ejecución vemos que el tipo de acceso (type=ref) en el que MySql va a buscar las filas de la tabla oj_onlinejudge_submisionns es un acceso por índice no único (problemid).

Las filas que estima leer son 73050, es decir el nuevo ratio es bastante bueno.

```
Ratio= 73050/ 7375 = 9.90
```

El tiempo que se tarda en ejecutar son 0,172 segundos, por lo que volviendo a hacer la simulación de tiempo anterior tendríamos:

```
T=tiempo de espera
n=consultas simultaneas
t=tiempo de ejecución de la consulta
T= n * t = 100 * 0.172 = 17.2 segundos
```

Por lo que observamos, obtendríamos una mejora de rendimiento notable (más del 50%), pero todavía no aceptable, por lo que deberíamos buscar un nuevo candidato a índice para mejorar el resultado.

Volviendo a visualizar la consulta, podemos intuir que si hiciéramos un índice compuesto, con las siguientes columnas (`problemid`,`userid`) , es posible que tuviéramos una notable mejora en el rendimiento.

Probemos con la creación de éste índice para ver qué resultados obtenemos:

```
create index problemid2 on prueba (problemid,userid)
```

Volvemos ejecutar el comando explain y veamos qué resultado nos ofrece

```
explain select count(distinct userid)from oj_onlinejudge_submisionns where problemid=36;
```

Columna	Resultado
Id	simple
Select_type	oj_onlinejudge_submisionns
type	ref
Posible_keys	problemid,problemid2
Key	problemid2
Key_len	4
Ref	const
Rows	74150
extra	using index

Con éste nuevo resultado, observamos que el índice que usará para acceder es problemid2(\`problemid\`,\`userid\`), y el número de bytes usados por el índice que nos da la salida key-len = 4 nos indica que solo se usará la columna más a la izquierda (problemid) , ya que este índice está formado por 2 int de 4 bytes cada uno

La columna extra: Using index recupera la información solicitada utilizando únicamente la información del índice, no tiene que acceder a la tabla. Esto sucede cuando todas las columnas requeridas forman parte del índice, a esto se le denomina Covering index.

Las filas que estima leer son 74150 aplicando la fórmula del ratio nos queda:

```
Ratio= 74150/ 7375 = 10.05
```

Por lo que no mejoramos bajo éste indicador, pero Covering indexes es una herramienta muy poderosa y pueden mejorar drásticamente el rendimiento.

El tiempo que se tarda en ejecutar son 0,019 segundos, por lo que volviendo a hacer la simulación de tiempo anterior tendríamos:

```
T=tiempo de espera
n=consultas simultaneas
t=tiempo de ejecución de la consulta
T= n * t = 100 * 0.019 = 1.9 segundos
```

Con la creación de éste nuevo índice hemos obtenido una notable mejoría y hemos pasado en ésta simulación de 36.2 segundos a 1.9 segundos.

7.1.2 CONCLUSIONES

Éste ejemplo práctico que hemos desarrollado para la optimización de una consulta que nos generaba serios problemas de rendimiento, podríamos tomarlo como pauta para mejorar el resto de consultas de la aplicación, que nos están generando problemas.

Muchas veces, es fácil de intuir cuales son las consultas a estudiar para que nuestra aplicación no sufra problemas de rendimiento, pero también es cierto que cuanto más se complica o evoluciona una aplicación, no resulta tan sencillo de identificar por donde debemos atajar el problema.

Como se comentaba en la parte teórica, es para estos casos donde MySQL proporciona herramientas que nos van a identificar donde debemos atajar el problema.

Una de estas herramientas anteriormente nombradas es activar SLOW QUERY LOG, dicha herramienta se configura a nivel de servidor, y con las opciones que mostramos ya en la parte teórica, nos permitirá reconocer todas las consultas que tienen una pobre respuesta.

Una vez contamos con ellas, lo siguiente es ir una a una, empezando por las más pesadas, o bien las más usadas y repetir el proceso que hemos comentado en éste caso práctico.

Aunque es equivoco pensar que repitiendo de forma automática éste proceso paso a paso, vamos a conseguir éxito en la mejora de rendimiento en todos los casos. Nos encontraremos en ocasiones que no consigamos optimizar estas consultas hasta el punto que queremos, y esto nos debería hacer reflexionar si sería necesario cambiar algo en el modelo de datos, u obtener la información que necesitamos realizando otras consultas, o utilizando otras herramientas de las que nos provee la base de datos, como es el caso de las vistas, los triggers o disparadores y los procedimientos almacenados.

En el siguiente apartado vamos a tratar como actuaríamos ante esta situación.

7.2 OPTIMIZACIÓN DE OTROS OBJETOS DE LA BASE DE DATOS

Cuando un usuario de la aplicación del juez online, nos envía una solución de uno de los problemas con los que se está ejercitando, el aplicativo realiza una inserción o modificación sobre la tabla oj_onlinejudge_submissions, esto a su vez provoca que se lance un trigger o disparador que actualiza tanto las tablas de las estadísticas de usuario, como la tabla de las estadísticas de los problemas.

Adicionalmente a esto, tenemos un "job" o tarea programada del sistema, que cada cierto tiempo h(actualmente cada 2 horas), lanza un procedimiento almacenado que actualiza la tabla de los rankings de cada problema.

Como podemos intuir, éste proceso es de suma importancia en la aplicación del juez, ya que es el que se encarga de generar todos los rankings, que son los que dan la visión de competición al aplicativo.

Esto como se puede suponer es una de las herramientas más potentes que tiene el sistema y provoca de manera directa que la gente siga optando por esta herramienta a la hora de entrenarse para las competiciones.

Lo idóneo, por tanto, sería que los rankings pudieran ser generados al instante, ya que el usuario una vez mandada la solución, podría conocer sus resultados en tiempo real.

Debido a un problema de rendimiento, se tomó la decisión de lanzar este procedimiento cada 2 horas, ya que si no, la base de datos se colapsaba e impedía el normal funcionamiento de la aplicación.

Nuestro objetivo en éste caso práctico, será minimizar el tiempo de ejecución de éste proceso para que la información de los rankings y estadísticas este sincronizada en todo momento

Actualmente tenemos un trigger o disparador, sobre la tabla oj_onlinejudge_submissions que detallamos a continuación:

```
BEGIN
DELETE from oj_onlinejudge_accepted where id=NEW.id;
IF NEW.verdict = 90 THEN
                INSERT IGNORE INTO oj_onlinejudge_accepted (id, problemid, userid, submittime) select id,problemid,userid,submittime from oj_onlinejudge_submissions where id=NEW.id ;
                INSERT IGNORE INTO oj_onlinejudge_problemrankingpending (problemid,`timestamp`) VALUES (NEW.problemid, UNIX_TIMESTAMP()) ;
END IF ;
    INSERT INTO oj_onlinejudge_userstats (userid) VALUES (NEW.userid) ON DUPLICATE KEY UPDATE userid = NEW.userid;
    UPDATE oj_onlinejudge_userstats SET
      submissions=(SELECT count(1) from oj_onlinejudge_submissions WHERE userid=NEW.userid),
      pbtried=(SELECT count(DISTINCT problemid) FROM oj_onlinejudge_submissions WHERE userid=NEW.userid),
      pbsolved=(SELECT count(DISTINCT problemid) FROM oj_onlinejudge_submissions WHERE userid=NEW.userid AND verdict='90')
      WHERE userid=NEW.userid ;
    INSERT INTO oj_onlinejudge_problemstats (problemid) VALUES (NEW.problemid) ON DUPLICATE KEY UPDATE problemid = NEW.problemid ;
    UPDATE oj_onlinejudge_problemstats SET
      submissions=(SELECT count(1) FROM oj_onlinejudge_submissions WHERE problemid=NEW.problemid),
      users=(SELECT count(DISTINCT userid) FROM oj_onlinejudge_submissions USE INDEX (problemid_2) WHERE problemid=NEW.problemid),
      users_solved=(SELECT count(DISTINCT userid) FROM oj_onlinejudge_submissions WHERE problemid=NEW.problemid AND verdict='90'),
      bestcpu=(select runtime from oj_onlinejudge_problemranking WHERE problemid=NEW.problemid and rank=1),
      ver_10=(SELECT count(1) FROM oj_onlinejudge_submissions WHERE problemid=NEW.problemid AND verdict='10'),
      ver_20=(SELECT count(1) FROM oj_onlinejudge_submissions WHERE problemid=NEW.problemid AND verdict='20'),
      ver_30=(SELECT count(1) FROM oj_onlinejudge_submissions WHERE problemid=NEW.problemid AND verdict='30'),
      ver_40=(SELECT count(1) FROM oj_onlinejudge_submissions WHERE problemid=NEW.problemid AND verdict='40'),
      ver_50=(SELECT count(1) FROM oj_onlinejudge_submissions WHERE problemid=NEW.problemid AND verdict='50'),
      ver_60=(SELECT count(1) FROM oj_onlinejudge_submissions WHERE problemid=NEW.problemid AND verdict='60'),
      ver_70=(SELECT count(1) FROM oj_onlinejudge_submissions WHERE problemid=NEW.problemid AND verdict='70'),
      ver_80=(SELECT count(1) FROM oj_onlinejudge_submissions WHERE problemid=NEW.problemid AND verdict='80'),
      ver_90=(SELECT count(1) FROM oj_onlinejudge_submissions WHERE problemid=NEW.problemid AND verdict='90')
      WHERE problemid=NEW.problemid ;
END
```

Y tenemos un procedimiento que se lanza cada 2 horas cuyo código es el siguiente:

```
BEGIN
DECLARE done INT DEFAULT FALSE;
DECLARE l_problemid INT;
DECLARE cur1 CURSOR FOR SELECT problemid FROM oj_onlinejudge_problemrankingpending ORDER BY `timestamp` ASC limit 100;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
OPEN cur1 ;
REPEAT
FETCH cur1 INTO l_problemid;
IF NOT done THEN

DELETE FROM oj_onlinejudge_problemranking WHERE problemid = l_problemid ;
SET @R=0;
INSERT IGNORE INTO oj_onlinejudge_problemranking (problemid,rank,userid,submissionid,runtime,submittime) SELECT * FROM (SELECT problemid,@R:=@R+1 as rank,userid,id as
submissionid,runtime,submittime FROM (select * from (SELECT id,runtime,`memory`,userid,submittime,problemid FROM oj_onlinejudge_submissions WHERE problemid=l_problemid AND verdict='90' ORDER BY
runtime,submittime,`memory`) a group by userid order by runtime,submittime,`memory`) t) u ;
UPDATE oj_onlinejudge_problemstats SET bestcpu=(select runtime from oj_onlinejudge_problemranking WHERE problemid=l_problemid and rank=1) WHERE problemid=l_problemid;
UPDATE `oj_onlinejudge_problemranking` SET `virtual_leader`= 1 where `problemid`=l_problemid and `runtime`= (select bestcpu from oj_onlinejudge_problemstats where problemid=l_problemid);


DELETE FROM oj_onlinejudge_problemrankingpending WHERE problemid = l_problemid ;


END IF;
UNTIL done END REPEAT ;
CLOSE cur1;


END
```

Una vez analizados en detalle, tanto el trigger, como el procedimiento, descubrimos que un problema importante se encuentra cuando consultamos la tabla oj_onlinejudge_submissions, filtrada por las columnas problemid y verdict.

Observamos que no tenemos ningún índice compuesto en ésta tabla bajo las columnas problemid y verdict y observando los planes de Ejecución, creemos necesario la creación de un nuevo índice en la tabla oj_onlinejudge_submissions:

El índice planteado sería un índice de tipo BTREE con 2 columnas

- Columna problemid
- Columna verdict

```
create index condicion on oj_onlinejudge_submission (problemid ,verdict ).
```

Una vez creado, volvemos a testear si los cambios realizados han surtido efecto, y se nota una evidente mejoría tanto para la ejecución tanto del trigger como del procedimiento.

```
FLUSH STATUS;
SELECT SQL_NO_CACHE id,runtime,`memory`,userid,submittime,problemid
FROM oj_onlinejudge_submissions
WHERE
            problemid = 36
            AND verdict='90'
ORDER BY
            runtime,submittime,`memory`;
```

Con la creación de éste índice, el servidor nos devuelve el resultado en 0,108 segundos. Aunque analizando la consulta observamos que también se está realizando una ordenación de los registros.

Mysql tiene 2 maneras de ordenar, bien usando un filesort o escanear un índice en orden.El escaneo del índice obviamente es mucho más rápido, si es posible siempre es una buena práctica diseñar los índices para que se puedan utilizar para ordenar y buscar filas.Para ello el orden del índice tiene que ser exactamente el mismo que la clausula order by y si existen condiciones en la clausula where tienen que formar la parte más izquierda del índice.

Así que nos planteamos la creación de un nuevo índice, en éste caso lo llamaremos ordenado y la definición sería la siguiente:

```
create index ordenado on oj_onlinejudge_submission (problemid,verdict,runtime,submittime,`memory`).
```

Una vez creado vamos a analizar, como ha mejorado en el rendimiento la creación de éste nuevo índice, lanzamos la siguiente sentencia:

```
FLUSH STATUS;
SELECT SQL_NO_CACHE id,runtime,`memory`,userid,submittime,problemid
FROM oj_onlinejudge_submissions
WHERE
            problemid = 36
            AND verdict='90'
ORDER BY
            runtime,submittime,`memory`;
```

Y se observa una enorme mejoría, ya que el resultado pasa de los 0,108 segundos con el primer índice a 0,0070 segundos con el índice "ordenado", lo que viene a ser que el índice "ordenado" es 15 veces más rápido que el índice "condición".

En éste punto es importante destacar, que crear más índices de los que usemos, es una mala práctica, ya que esto penalizará enormemente todas las operaciones DML, como son las operaciones (insert, update y delete) ya que cada vez que se produce una modificación de datos en la tabla implica que estas modificaciones se tendrán que hacer en todas las tablas de índices que tengamos creadas.

Por ello, tras decidir con que índice nos quedamos, no hay que olvidar borrar aquellos índices que no van a ser usados.

Y es ahora cuando aun teniendo ya optimizada esta consulta, nos encontramos con el hecho, de que solo aplicando esto, todavía no hemos conseguido el objetivo que nos planteamos y tenemos que ir más allá.

Basándonos en las buenas prácticas que comentamos en la teoría, cuando hemos optimizado lo más posible las consultas y no podemos validar la optimización como suficiente, es el momento de replantearse si puedo sacar las consultas de otra manera. Si es así y logro el objetivo, perfecto. Si aún así tampoco lo consigo, es momento de optimizar el proceso con otros objetos que nos dan las bases de datos tales como vístas, triggers o disparadores y procedimientos almacenados.

Bajo éste prisma, vamos a pasar a optimizar triggers y procedimientos almacenados, dividiremos el proceso de actualización en un trigger sobre la tabla oj_onlinejudge_submissions, que llamará a los distintos procedimientos que actualizan los datos estadísticos y de ranking.

Éste trigger lanzara tres procedimientos almacenados que pasamos a nombrar:

1. **actualiza_estadisticas_usuario:** procedimiento que se encarga de actualizar la tabla oj_onlinejudge_userstats, que representa las estadísticas por usuarios

2. **actualiza_estadisticas_problema:** actualiza la tabla oj_onlinejudge_problemstats, para ello se crea un cursor que en una única consulta me calcula las columnas ver_10, ver_20, ver_30,..... evitando así a realizar constantes llamadas a la base de datos.

3. **actualiza_ranking:** actualiza las tablas oj_onlinejudge_problemranking y oj_onlinejudge_problemstats, este procedimiento es el que más carga tiene, los delete y update y en general las operaciones DDL penalizan el rendimiento de la base de datos, ya que todos los índices de la tabla deben ser actualizados junto con la tabla.

Como resultado de lo comentado, describimos como quedaría el trigger o disparador:

```
CREATE DEFINER=`root`@`localhost` TRIGGER `updstats` AFTER UPDATE ON `oj_onlinejudge_submissions`
 FOR EACH ROW BEGIN
DELETE from oj_onlinejudge_accepted where id=NEW.id;
IF NEW.verdict = 90 THEN
    INSERT IGNORE INTO oj_onlinejudge_accepted (id, problemid, userid, submittime) select
id,problemid,userid,submittime from oj_onlinejudge_submissions where id=NEW.id ;
    INSERT IGNORE INTO oj_onlinejudge_problemrankingpending (problemid,`timestamp`) VALUES (NEW.problemid,
UNIX_TIMESTAMP()) ;
END IF ;

    call actualiza_estadisticas_usuario(NEW.userid);

    call actualiza_estadisticas_problema(NEW.userid,NEW.problemid);

    call actualiza_ranking(NEW.problemid);
END
```

A continuación, mostramos el detalle de cada uno de los procedimientos:

7.2.1 PROCEDIMIENTO "actualiza_estadisticas_usuario"

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `actualiza_estadisticas_usuario`(in p_user_id int)
BEGIN
 INSERT INTO oj_onlinejudge_userstats (userid) VALUES (p_user_id) ON DUPLICATE KEY UPDATE userid = p_user_id;
    UPDATE oj_onlinejudge_userstats SET
      submissions=(SELECT count(1) from oj_onlinejudge_submissions WHERE userid=p_user_id),
      pbtried=(SELECT count(DISTINCT problemid) FROM oj_onlinejudge_submissions WHERE userid=p_user_id),
      pbsolved=(SELECT count(DISTINCT problemid) FROM oj_onlinejudge_submissions WHERE userid=p_user_id AND
verdict='90')
      WHERE userid=p_user_id ;
END$$


DELIMITER ;
```

7.2.2 PROCEDIMIENTO "actualiza_estadisticas_problema"

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `actualiza_estadisticas_problema`(in p_user_id int,in p_problem_id
int)
BEGIN
   DECLARE done INT DEFAULT FALSE;

        DECLARE v_verdict text;
        DECLARE v_cuantos ,v_10,v_20,v_30,v_40,v_50,v_60,v_70,v_80,v_90,v_total  INT;
        DECLARE cur1 CURSOR FOR
            SELECT
                verdict ,count(id)  AS 'cuantos'
            from
                oj_onlinejudge_submissions
            WHERE
                problemid=p_problem_id
            group by
                verdict ;


DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
      set v_10=0;set v_20=0;set v_30=0;set v_40=0;set v_50=0;
      set v_60=0;set v_70=0;set v_80=0;set v_90=0;set v_total=0;

     OPEN cur1 ;
        REPEAT


                 FETCH cur1 INTO v_verdict,v_cuantos;
                IF NOT done THEN
                    set v_total=v_total+v_cuantos;

                        CASE v_verdict
                            WHEN '10' THEN set v_10=v_cuantos;
                            WHEN '20' THEN set v_20=v_cuantos;
                            WHEN '30' THEN set v_30=v_cuantos;
                            WHEN '40' THEN set v_40=v_cuantos;
                            WHEN '50' THEN set v_50=v_cuantos;
                            WHEN '60' THEN set v_60=v_cuantos;
                            WHEN '70' THEN set v_70=v_cuantos;
                            WHEN '80' THEN set v_80=v_cuantos;
                            WHEN '90' THEN set v_90=v_cuantos;
                        END CASE;
                END IF;
        UNTIL done END REPEAT ;
    CLOSE cur1;

   INSERT INTO oj_onlinejudge_problemstats (problemid) VALUES (p_problem_id) ON DUPLICATE KEY UPDATE problemid =
p_problem_id;

    UPDATE oj_onlinejudge_problemstats SET
      submissions=v_total,
      users=(SELECT count(DISTINCT userid) FROM oj_onlinejudge_submissions USE INDEX (problemid_2) WHERE
problemid=p_problem_id),
      users_solved=(SELECT count(DISTINCT userid) FROM oj_onlinejudge_submissions WHERE problemid=p_problem_id
AND verdict='90'),
      bestcpu=(select runtime from oj_onlinejudge_problemranking WHERE problemid=p_problem_id and rank=1),


      ver_10=v_10,
      ver_20=v_20,
      ver_30=v_30,
      ver_40=v_40,
      ver_50=v_50,
      ver_60=v_60,
      ver_70=v_70,
      ver_80=v_80,
      ver_90=v_90
      WHERE problemid=p_problem_id ;

END$$
DELIMITER ;
```

7.2.3 PROCEDIMIENTO “ actualiza_ranking ”

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `actualiza_ranking`(In P_PROBLEM_ID INT)
BEGIN
        DECLARE done INT DEFAULT FALSE;
        DECLARE l_problemid ,l_userid,l_submissionid,l_runtime,l_submittime INT;
        DECLARE v_ranking,v_virtual_leader INT;
        DECLARE cur1 CURSOR FOR
            SELECT problemid,userid,id as submissionid,runtime,submittime
            FROM (
                select * from (SELECT id,runtime,`memory`,userid,submittime,problemid
                FROM oj_onlinejudge_submissions
                WHERE
                    problemid = P_PROBLEM_ID
                    AND verdict='90'
                ORDER BY
                    runtime,submittime,`memory`) a
            group by userid
            order by runtime,submittime,`memory`)x;


       DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
        SET v_ranking=0;
     OPEN cur1 ;
        REPEAT


            FETCH cur1 INTO l_problemid,l_userid,l_submissionid,l_runtime,l_submittime ;
                IF NOT done THEN
                SET v_ranking=v_ranking+1;

                if (v_ranking=1) then
                    set v_virtual_leader=1;


                    UPDATE oj_onlinejudge_problemstats
                    SET bestcpu=l_runtime
                    WHERE problemid=l_problemid;



                else
                    set v_virtual_leader=0;
                end if;
               INSERT INTO  `oj_onlinejudge_problemranking`
                                             (`problemid`,
                                             `rank`,
                                             `userid`,
                                             `submissionid`,
                                             `runtime`,
                                             `submittime`,
                                             `virtual_leader`)
                    VALUES
                                         (l_problemid,
                                         v_ranking,
                                         l_userid,
                                         l_submissionid,
                                         l_runtime,
                                         l_submittime,
                                         v_virtual_leader)

                ON DUPLICATE KEY UPDATE
                                rank=v_ranking,
                                submissionid=l_submissionid,
                                runtime=l_runtime,
                                submittime=l_submittime,
                                virtual_leader=v_virtual_leader
                 ;


                END IF;
        UNTIL done END REPEAT ;
    CLOSE cur1;

END$$
DELIMITER ;

DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `lanza_ranking`()
BEGIN
        DECLARE done INT DEFAULT FALSE;
        DECLARE l_problemid INT;
        DECLARE cur1 CURSOR FOR SELECT problemid FROM oj_onlinejudge_problemrankingpending ORDER BY `timestamp` DESC limit 100;



        DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
    OPEN cur1 ;
        REPEAT
            FETCH cur1 INTO l_problemid;
                IF NOT done THEN
                call actualiza_ranking(l_problemid);

                END IF;
        UNTIL done END REPEAT ;
    CLOSE cur1;

END$$
DELIMITER ;
```

7.3 RESULTADO

Una vez realizadas las modificaciones anteriormente citadas, obtenemos una mejora notable en el rendimiento.

Pasaríamos a tener en el trigger, todas las modificaciones en las tablas de estadísticas (usuario, problemas, ranking), y tendríamos dichas tablas sincronizadas en el acto, sin tener la necesidad de pasar el proceso cada 2 horas.

Realizando una comparativa de tiempos observamos lo siguiente:
Con un total de 1.000.000 registros en la tabla oj_onlinejudge_submissions.

Si mantenemos la solución actual:

PROCEDIMIENTO actual	TIEMPO EN MILISEGUNDOS
Trigger `UPDATE ON `oj_onlinejudge_submissions``	=250ms
Procedimiento rankings	=120.000 ms

En la solución planteada sería factible meter todas las actualizaciones en el trigger sin la necesidad de lanzar ningún proceso después cada 2 horas para los rankings ya que obtenemos un tiempo de ejecución del trigger

PROCEDIMIENTO propuesto	TIEMPO EN MILISEGUNDOS
Trigger `UPDATE ON `oj_onlinejudge_submissions``	=400ms

Si el tiempo en el update nos pareciera excesivo, podríamos optar a mantener el trigger sin la actualización de los rankings, lanzando un procedimiento similar al que actualmente se lanza y que anexamos debajo con una mejoría notable como observamos a continuación

PROCEDIMIENTO propuesto en 2 pasos	TIEMPO EN MILISEGUNDOS
Trigger `UPDATE ON `oj_onlinejudge_submissions``	=180ms
Procedimiento rankings	=3.000 ms

Como observamos la mejora del rendimiento es notable y nos da solución al problema planteado.

```
DELIMITER $$

CREATE DEFINER=`root`@`localhost` PROCEDURE `lanza_ranking`()

BEGIN

            DECLARE done INT DEFAULT FALSE;

            DECLARE l_problemid INT;

            DECLARE cur1 CURSOR FOR SELECT problemid FROM oj_onlinejudge_problemrankingpending ORDER BY `timestamp` ASC  limit 100;






            DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

      OPEN cur1 ;

            REPEAT

                  FETCH cur1 INTO l_problemid;

                        IF NOT done THEN

                        call actualiza_ranking(l_problemid);


                        END IF;

            UNTIL done END REPEAT ;

      CLOSE cur1;


END$$

DELIMITER ;
```

8.- VERIFICACION DE RESULTADOS

Cuando ejecutamos una sentencia SQL desde la linea de comandos, el tiempo transcurrido de ejecución puede ser una buena medición, pero en un entorno productivo, existen diversas razones por las que el tiempo de ejecución de la misma consulta puede variar entre ejecuciones, Y esto no solo se debe a lo bien que hemos optimizado la SQL, sino a otros factores como son:

- El Número de I/O (entradas y salidas a disco): dependerá de los datos almacenados en la cache.
- Usuarios Concurrentes: múltiples usuarios pueden estar trabajando a la vez por lo que tendremos que compartir cpu, entradas y salidas a disco, incluso bloqueos.

Es por ello que para comprobar que nuestros ajustes han surtido efecto, nos tenemos que apoyar en otras herramientas que nos ofrece MySQL, entre las que se encuentran "Show STATUS" y "mysqlslap" que vamos a utilizar para verificar los resultados.

8.1 HERRAMIENTA SHOW STATUS

Cuando ejecutamos una consulta en MySQL, uno de los factores más decisivos en el tiempo que tarda en ejecutarse tiene que ver con el número de lecturas físicas a disco que el motor de base de datos tiene que hacer para devolver el resultado.

Está claro, que todo lo que podamos sacar de memoria (lecturas lógicas) y no tengamos que acceder a disco para resolver la consulta (lecturas físicas), mejorara el tiempo de respuesta.

Para comprobar que cuando ejecutamos una consulta repetidas veces, la base de datos es capaz de resolverla sin acceder a disco, MySQL nos da una herramienta que calcula para una SQL dada, cuantas lecturas físicas y lecturas lógicas realiza.

Ésta herramienta se ejecuta de la siguiente manera:

```
SHOW STATUS LIKE 'Innodb_buffer_pool_read_requests'
```

Éste comando nos devolverá el número de lecturas lógicas, y si queremos saber el número de lecturas físicas utilizaremos la siguiente sentencia:

```
SHOW STATUS LIKE 'Innodb_data_reads';
```

En nuestro caso de estudio, vamos a aplicarlo a una de las consultas en las que veíamos que el uso de índices optimizaba el resultado, para ello vamos a simular que lanzamos dicha consulta 4 veces, en éste primer caso vamos a ejecutar la consulta sin usar ningún índice y el resultado que nos da es el siguiente:

Lanzamos la consulta que queremos analizar:

```
Select count(distinct userid) from oj_onlinejudge_submissions where problemid=36
```

Veamos las lecturas lógicas, utilizando el comando anteriormente nombrado:

```
SHOW STATUS LIKE 'Innodb_buffer_pool_read_requests';
```

Y el resultado que obtenemos es el siguiente:

```
'Innodb_buffer_pool_read_requests', '3051813'
```

Ahora comprobemos las lecturas físicas:

```
SHOW STATUS LIKE 'Innodb_data_reads';
```

Y obtenemos:

```
'Innodb_data_reads', '1494'
```

Si repetimos el proceso hasta en cuatro ocasiones, es decir vamos a simular que se lanza la misma consulta 4 veces, vamos a analizar lo que nos dicen los resultados y que conclusiones podemos sacar.

Lecturas Lógicas:

Innodb_buffer_pool_read_requests	VALOR	DIFERENCIA
Ejecución 1	3051813	
Ejecución 2	4059132	1007319
Ejecución 3	5066457	1007325
Ejecución 4	6073711	1007254

Lecturas Físicas:

Innodb_data_reads	VALOR	DIFERENCIA
Ejecución 1	1494	
Ejecución 2	1871	377
Ejecución 3	2248	377
Ejecución 4	2625	377

Ahora repetimos el proceso de ejecutar la misma consulta, pero en esta ocasión usando índices:

```
Select count(distinct userid) from oj_onlinejudge_submissions where problemid=36
```

Lecturas Lógicas:

Innodb_buffer_pool_read_requests	VALOR	DIFERENCIA
Ejecución 1	6078015	
Ejecución 2	6082319	4304
Ejecución 3	6086623	4304
Ejecución 4	6090927	4304

Lecturas Físicas:

Innodb_data_reads	VALOR	DIFERENCIA
Ejecución 1	2625	
Ejecución 2	2625	0
Ejecución 3	2625	0
Ejecución 4	2625	0

Como podemos comprobar a la vista de los resultados obtenidos, queda claro que la consulta

ejecutada con los índices creados, obtiene resultados evidentes de mejoría.

Es decir con la consulta ejecutada sin índices, observamos que el número de lecturas lógicas es mucho mayor que en el caso de la consulta indexada, pero lo que es más importante, después de ejecutar la primera vez dicha consulta, la consulta indexada nos muestra que no tiene que realizar ningún acceso a disco para devolver el resultado, y esto marca y mucho el tiempo de respuesta que nos da el servidor a la hora de devolvernos el resultado.

Como prácticamente siempre, no nos debemos quedar con un solo estadístico para verificar el correcto funcionamiento de un hecho, y es por ello que para tener la certeza de que todo lo que hemos hecho ha mejorado objetivamente el rendimiento, vamos a pasar a utilizar otra herramienta que nos cerciore la mejora.

8.2 HERRAMIENTA MYSQLSLAP

Como comentamos en la parte teórica, mysqlslap es una herramienta de benchmarking que nos proporciona MySQL para analizar el rendimiento tanto de consultas como de procedimientos almacenados, y en la que también intervienen valores tan determinantes como la concurrencia, es decir simularemos un entorno real con muchos usuarios distintos conectados a la vez.

En nuestro caso práctico, lo aplicaremos a una de las consultas que nos generaban problemas, y al procedimiento que hacía que no pudiéramos tener en tiempo real los resultados estadísticos de la herramienta del juez online.

Partimos de la siguiente consulta:

```
select count(distinct userid) from oj_onlinejudge_submissions where problemid=36
```

Haremos una simulación de 100 usuarios conectados a la vez, e iteraremos el proceso 4 veces para que nos de una respuesta real, sobre una tabla que no está indexada, para ello utilizaremos el siguiente comando:

```
 ./mysqlslap --query  "  select count(distinct userid) from oj_onlinejudge_submissions where problemid=36" --create-schema "renovado"
-u root -p --iterations 4 --concurrency 100
```

Y nos devolverá el siguiente resultado:

```
Benchmark
        Average number of seconds to run all queries: 10.507 seconds
        Minimum number of seconds to run all queries: 10.383 seconds
        Maximum number of seconds to run all queries: 10.730 seconds
        Number of clients running queries: 100
        Average number of queries per client: 1
```

Vamos a crear un índice, en este caso el índice de una única columna "problemid", que

todavía no sería el óptimo, y lancemos el comando para ver si se ha notado algún tipo de mejoría:

```
 ./mysqlslap --query  "  select count(distinct userid) from oj_onlinejudge_submissions where problemid=36" --create-schema "renovado"
-u root -p --iterations 4 --concurrency 100
```

Y nos devolverá el siguiente resultado:

```
Benchmark
            Average number of seconds to run all queries: 5.203 seconds
            Minimum number of seconds to run all queries: 2.454 seconds
            Maximum number of seconds to run all queries: 10.123 seconds
            Number of clients running queries: 100
            Average number of queries per client: 1
```

Finalmente realizamos el mismo test, pero en éste caso con el índice óptimo para resolver esta consulta, que sería el índice de cobertura problemid2 (problemid,userid).

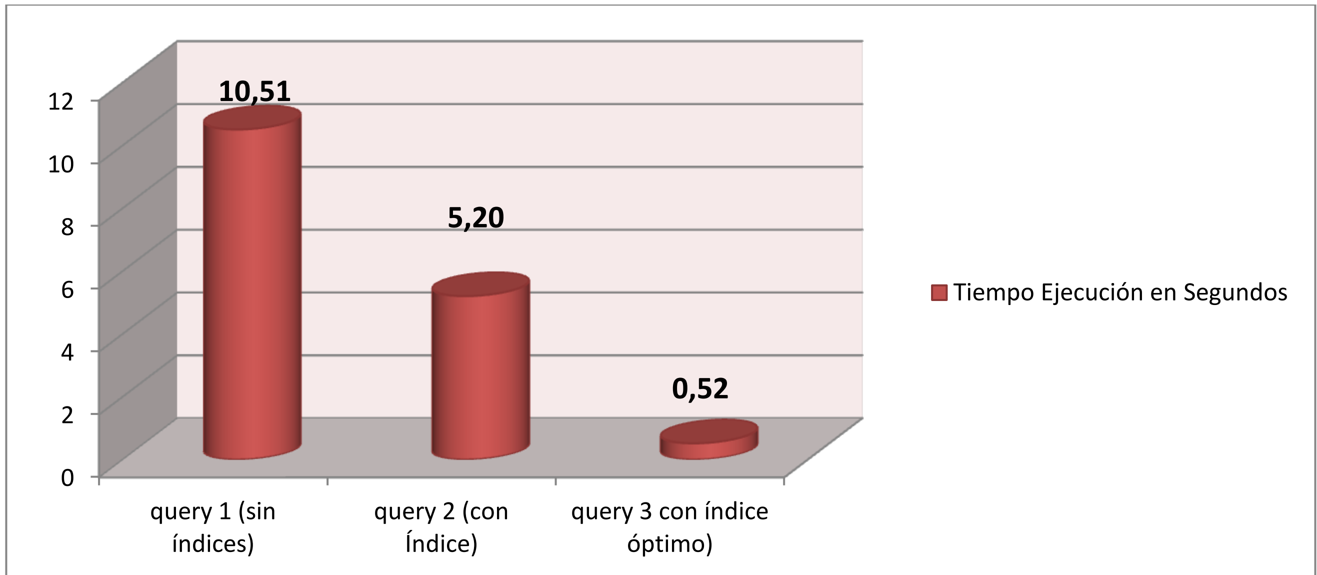
```
 ./mysqlslap --query  "  select count(distinct userid) from oj_onlinejudge_submissions where problemid=36" --create-schema "renovado"
-u root -p --iterations 4 --concurrency 100
```

Y nos devolverá el siguiente resultado:

```
Benchmark
            Average number of seconds to run all queries: 0.516 seconds
            Minimum number of seconds to run all queries: 0.495 seconds
            Maximum number of seconds to run all queries: 0.537 seconds
            Number of clients running queries: 100
            Average number of queries per client: 1
```

A primera vista, ya se observa que el resultado obtenido con la creación del índice apropiado sobre la tabla, mejora espectacularmente los tiempos de respuesta del servidor, lo que hace adivinar que en ésta ocasión la estrategia tomada para la creación del índice ha sido acertada.

Veámoslo gráficamente, para observar la mejora del rendimiento.



Vamos a continuar con el uso de la herramienta mysqlslap, pero en esta ocasión la aplicaremos sobre el trigger que se lanzaba al hacer un update sobre la tabla oj_onlinejudge_submissions.

El comando a utilizar, para lanzar la simulación sobre el trigger original, sería el siguiente:

```
./mysqlslap --query  "  UPDATE oj_onlinejudge_submissions SET verdict=90 where id=2" --create-schema "original" -u root -p --
iterations 4
```

Y el resultado que obtenemos es:

```
Benchmark
            Average number of seconds to run all queries: 0.380 seconds
            Minimum number of seconds to run all queries: 0.043 seconds
            Maximum number of seconds to run all queries: 1.354 seconds
            Number of clients running queries: 1
            Average number of queries per client: 1
```

Esto actualizaría las estadísticas a nivel de usuario y problema, pero el verdadero escollo lo teníamos cuando lanzábamos el procedimiento que calculaba el ranking. Para realizar dicha simulación lanzaríamos el siguiente comando:

```
mysqlslap --query  "call carga_original" --create-schema "original" -u root -p --iterations 4
```

obteniendo

```
Benchmark
            Average number of seconds to run all queries: 116.572 seconds
            Minimum number of seconds to run all queries: 112.468 seconds
            Maximum number of seconds to run all queries: 118.383 seconds
            Number of clients running queries: 1
            Average number of queries per client: 1
```

Como podemos ver, ejecutar este procedimiento, que como vemos tarda casi 2 minutos, nos generaba un verdadero problema, ya que no podíamos tener los rankings actualizados en tiempo real.

Para evitar esto, decidimos recodificar el trigger, y que en este mismo instante lanzara estadísticas de usuario, estadísticas de problema y ranking en tiempo real, para probar el resultado obtenido lanzamos el siguiente comando:

```
./mysqlslap --query  "  UPDATE renovado.oj_onlinejudge_submissions SET verdict=90 where id=2" --create-schema "renovado" -u root -p --iterations 4
```

Y vemos en la siguiente salida:

```
Benchmark
            Average number of seconds to run all queries: 0.097 seconds
            Minimum number of seconds to run all queries: 0.093 seconds
            Maximum number of seconds to run all queries: 0.108 seconds
            Number of clients running queries: 1
            Average number of queries per client: 1
```

La recodificación y en este caso el cambio de planteamiento en el procedimiento, hace que tengamos resultados espectaculares en el rendimiento de la base de datos.

Originalmente teníamos que lanzar un trigger para calcular estadísticas de usuario y problemas, lo cual no era un problema notorio ya que el tiempo de respuesta era de 0.38 segundos, pero para poder disponer del ranking teníamos un tiempo de respuesta de casi 2 minutos y además no disponíamos en tiempo real de esta información.

Con la nueva solución planteada, en un solo trigger lanzar todo, el tiempo de respuesta es de 0,09 segundos, es decir inapreciable para el usuario y esto nos permite disponer de toda la información necesaria en tiempo real.

Es decir, en muchas ocasiones, cambiar la lógica de cómo resolver la información que necesitamos es el arma más poderosa en la optimización de la base de datos.

Pero vamos a imaginar que no quisiéramos cambiar la lógica, y que el ranking lo quisiéramos lanzar fuera del trigger como un procedimiento aparte como originalmente estaba planteado.

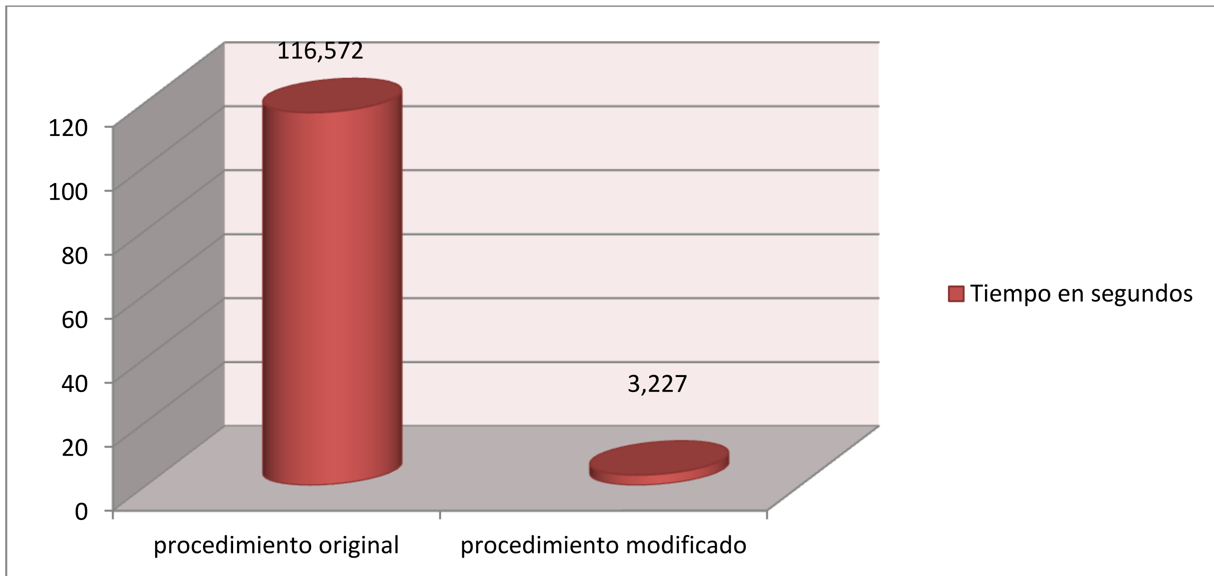
En éste caso, con la misma lógica pero aplicando las buenas prácticas de optimización en procedimientos almacenados, simularíamos el lanzamiento de éste cálculo con el siguiente comando:

```
./mysqlslap --query  "call lanza_ranking" --create-schema "renovado" -u root -p --iterations 4
```

Y el resultado obtenido sería:

```
            Average number of seconds to run all queries: 3.227 seconds
            Minimum number of seconds to run all queries: 3.203 seconds
            Maximum number of seconds to run all queries: 3.266 seconds
            Number of clients running queries: 1
            Average number of queries per client: 1
```

Gráficamente el resultado comparativo es de nuevo sorprendente, y aplicando exclusivamente técnicas de optimización pasamos de los casi 2 minutos original a tan solo 3 segundos.



Como conclusión de éste análisis, nos quedaríamos que la mejor opción es la del trigger que lanza tanto las estadísticas de usuario, de problema y el ranking con el nuevo planteamiento, ya que con esto habremos conseguido el objetivo fundamental, que el usuario de la aplicación disponga de toda la información en tiempo real.

9.- CONCLUSIONES

Cuando nos encontramos con un problema de rendimiento en una aplicación web, y en éste caso concreto en la aplicación del juez online, uno de los puntos más decisivos a abordar, se centra en el ajuste de la base de datos y en la optimización tanto de las consultas, como de los procedimientos almacenados que generan la lógica y los resultados, que se trasladarán a la aplicación web.

Aunque los problemas de rendimiento en las bases de datos, van creciendo según crece el volumen de datos y la concurrencia de usuarios, existen ajustes proactivos que hay que abordar en las etapas de análisis, diseño y codificación de la aplicación.

Por ello es realmente importante, seguir unas pautas en estas etapas, que mostramos a continuación, que nos ayudarán mucho cuando la aplicación esté en un entorno productivo.

- Un modelo de datos sencillo y normalizado que nos permita escribir las consultas de manera sencilla. Si las sentencias SQL son largas y complicadas el optimizador no será capaz de optimizarlas de manera eficaz.
- Creación de claves primarias, únicas y ajenas.
- Apoyarnos en el uso de objetos de base de datos, como son los procedimientos almacenados y los triggers que nos aportan ventajas tales como :
    - romper complejas SQL, en sentencias más sencillas que resultarán más fáciles de optimizar tanto al desarrollador como al optimizador
    - compartir cursores, que minimicen así la fase de análisis de una consulta
    - Evitar el uso de subconsultas correlacionadas, es decir evitar subconsultas que se ejecutan para cada fila que se encuentra en la consulta externa.

En nuestro caso, en la aplicación del juez online, partíamos de una aplicación que además de estar ya en un entorno productivo, se sumaba un gran volumen de usuarios que nos generaba verdaderos cuellos de botella.

Cuando nos encontramos en éste punto, debemos pasar a realizar ajustes reactivos, que podríamos resumir en los siguientes puntos:

- Identificar las consultas que están generando el problema de rendimiento
- Reconstrucción de índices : eliminar índices no selectivos o agregar columnas al índice para mejorar la selectividad
- Actualizar estadísticas periódicamente para que el optimizador tenga la mejor información posible(lanzar un job que las actualice)
- Crear nuevas estrategias de indexación
- Reescribir consultas y procedimientos para conseguir un mejor rendimiento

El grueso de éste proyecto, por tanto, se ha centrado fundamentalmente en los puntos descritos en los ajustes reactivos, aplicando las buenas prácticas que hemos ido describiendo a lo largo del proyecto, hemos llegado a conseguir los objetivos que nos marcamos al principio que recordemos que se centraba en conseguir disponer de la información tanto de los rankings, como de las estadísticas de los problemas y los usuarios en tiempo real.

Los resultados, contrastados con herramientas de simulación y benchmarking, han demostrado que estas técnicas pueden cambiar radicalmente el comportamiento de la base de datos, y por lo tanto de la aplicación web, y conseguir que un proceso que tardaba 116,572 segundos en ejecutarse para conseguir los rankings, planteándolos de otra forma, pase a ejecutarse en 0,097 segundos y cumplir el objetivo de pasar a ser un proceso online.

Por último no olvidemos que el proceso de optimización, siempre acompañara la vida de cualquier aplicación, llegando a formar parte vital del buen funcionamiento de la misma.

10.-BIBLOGRAFIA

- Skiena,Steven S. ;Revilla, Miguel A.Concursos internacionales de informática y Programación.Valladolid,2006 Secretariado de Publicaciones e Intercambio Editorial
- Harrison, Guy ; Feuerstein, Steven.MySql Stored Procedure Programming,2006 O'Reilly
- Schwartz,Baron; Zaitsev,Peter; Tkachenko,Vadim; Zawodny, Jeremy D.;Lentz,Arjen ;Balling,Derek J.High Performance MySQL. 2ª edición,2008 O'Reilly
- Deleglise, Didier MySQL 5 (versiones 5.1 a 5.6): Guía de referencia del desarrollador, 2013 Ediciones ENI

SITIOS WEB

- MySQL 5.6 Reference Manual.<https://dev.mysql.com/doc/refman/5.6/en/index.html> [Última Consulta :20/07/2015
- MySQL tools: benchmarking con mysqlslap.<http://rm-rf.es/mysql-tools-benchmarking-con-mysqlslap/> [Última Consulta : 12/08/2015]
- Advanced MySQL query and Schema Tuning. <https://www.percona.com/live/mysql-conference-2013/sessions/advanced-mysql-query-and-schema-tuning>.[Última Consulta: 21/07/2015]
- UVa Online Judge <https://uva.onlinejudge.org/>[Última Consulta:23/07/2015]
- uHunt:UVa hunting <http://uhunt.felix-halim.net/>[Última Consulta: 15/07/2015]
- uDebug <http://www.udebug.com/>[Última Consulta: 18/07/2015]