**Rendimiento de la API:**

Se realizará un breve análisis de cómo poder escalar nuestra API la cual, si bien funciona para unos pocos usuarios, deberá ser consumida por más y más usuarios a lo largo del tiempo. Por lo tanto, teniendo esto en mente, trataremos el problema de escalabilidad.

La prueba a realizar analizará el impacto de la base de datos sobre la disponibilidad de la API.

**LoadTest en SQLite:**

*Primero vamos a realizar un test de carga sobre la base de datos directamente.*

**Inicio**:

Se cargaron datos aleatorios sobre las tablas de la base de datos para poder estresar a la base de datos con los planes de prueba.

Las tablas fueron cargadas con la siguiente cantidad de registros:

* Cancion: 10000
* Fragmento: 40000
* Juego: 40000
* Tag: 10000
* Usuario: 14000

Se harán tanto consultas como escrituras para observar el porcentaje de error a causa de que la base de datos esté ocupada.

**Hipótesis**:

Debido a la cantidad de registros elegida se esperará que la BD presente problemas a la hora de la escritura (lockeo) afectando a las próximas consultas (al ser casi simultáneas, el error podrá ser considerable).

Debe tenerse en cuenta, que nuestra API, necesitará una buena respuesta frente a escrituras, ya que al finalizar un juego, este debe grabarse en el sistema, además, muchos usuarios intentarán subir canciones nuevas que no se encuentren en el sistema.

**Prueba**:

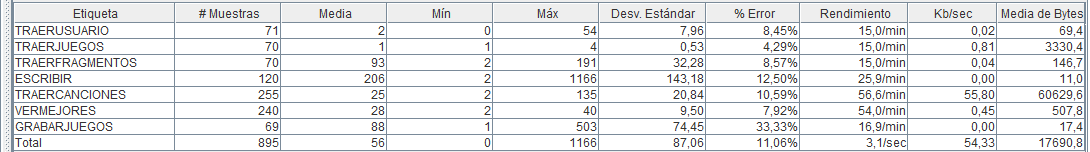
Realizaremos una prueba con Jmeter. La prueba tratará de reflejar el siguiente caso, donde se observan 3 grupos:

* Grupo 1) 10 usuarios ingresan al sistema a lo largo de 5 segundos y realizan consultas sobre su usuario (perfil), ven sus juegos, piden un juego y luego suben ese juego.
* Grupo 2) Cada 15 segundos, ocurren picos de 15 usuarios que ingresan a observar canciones y usuarios.
* Grupo 3) Cada 7 segundos, un grupo de usuarios ingresa a escribir canciones en el sistema (se incluyen los respectivos fragmentos) en el sistema.

Cada acción realizada por los usuarios presenta una pequeña demora para tratar de representar de forma más realista al usuario que ingresa.

La prueba se repite indefinidamente (tratando de dar la apareciencia que los usuarios no se desconectan sino que se mantienen).

A continuación se ingresan los resultados:



* TRAERUSUARIO, TRAERJUEGOS, TRAERFRAGMENTOS Y GRABARJUEGOS pertenecen al grupo 1.
* ESCRIBIR pertenece al grupo 3.
* TRAERCANCIONES y VERMEJORES pertenecen al grupo 2.

Puede observarse en la imagen que podemos mantener 3.1 operaciones/segundo con una tasa de error del 11.05%.

Los principales focos de error son las escrituras (GRABARJUEGOS y ESCRIBIR, las cuales lockean sus tablas).

Además puede observarse que VERMEJORES, TRAERCANCIONES y TRAERFRAGMENTOS tienen un tiempo de ejecución bastante más alto que el de TRAERUSUARIO y TRAERJUEGOS (todas consultas). Traer fragmento es el más importante de estas consultas, ya que es la base del juego.

**Plan de mejora**:

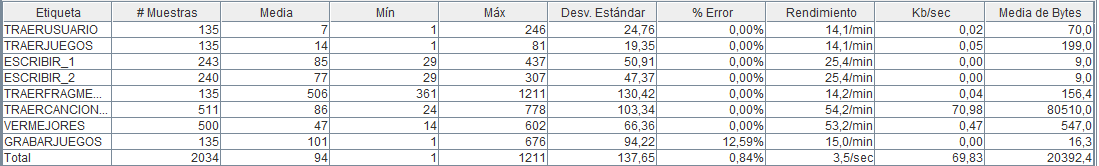
Si bien SQLite es una base de datos simple y útil para iniciar, no es lo suficientemente escalable para esta clase de aplicación (dado que sqlite lockea la base de datos ante cada escritura), por lo que se propone pasar de SQLite a una base de datos que permita concurrencia ante escrituras y lecturas.

El objetivo: Observar si el reemplazo de la base de datos genera un cambio drástico o no, del rendimiento de las consultas y en consecuencia, el de la API, además de ver si el error disminuye.

La base de datos a la cual portar será MariaDB.

**Aplicación de la mejora**:

Luego de portar la base de datos a MariaDB se realizaron las mismas pruebas que antes, dando como resultado:



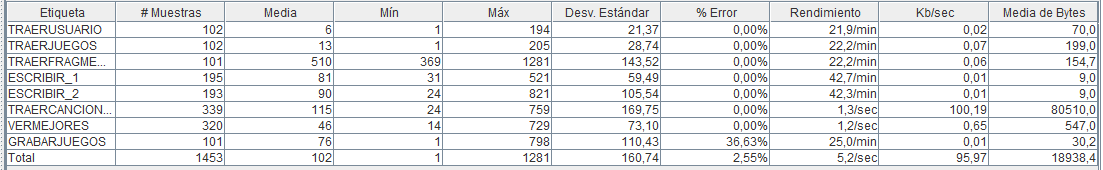
(Escribir fue separado en 2 requests distintos, debido a un error en la llamada desde Jmeter).

Como se puede observar, el error se redujo en casi un 10%, logrando que ahora podamos hacer 0.4 consultas correctas extra por segundo (24 consultas más por minuto).

[GRABARJUEGOS presenta un leve error, debido a un problema con el TIMESTAMP que no es reconocido correctamente para la clave primaria, suponemos, que de no ocurrir este problema, el error sería casi nulo]

Además, ahora que la base no se lockea, las lecturas ya no son fuente de errores (100% éxito).

A continuación, se agrega una prueba con una base de usuarios mayor (casi un 50% más en cada grupo).

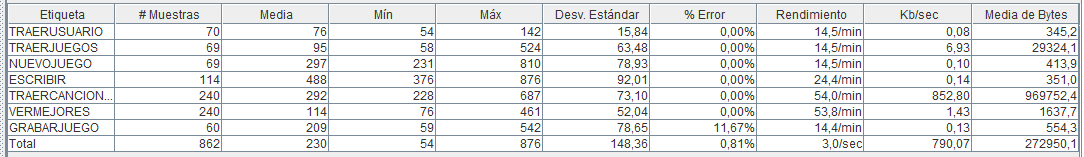


Al igual que antes, el error es despreciable.

**Conclusiones**:

El cambio de una BD que soporta poca concurrencia a una BD más profesional que si la soporta, nos ha permitido reducir considerablemente la tasa de error, por ende, mejorar la disponibilidad de nuestra API al no tener que volver a hacer requests cada vez que la base de datos falla.

**API con sqlite**



El error se redujo debido a que CodeIgniter implementa timeouts para la base de datos, por lo tanto si está lockeada, vuelve a intentarlo, sin embargo esto incrementa en gran medida el tiempo de respuesta a las peticiones.

Puede observarse por ejemplo los requests de TRAERUSUARIO y TRAERJUEGOS comparados con los mismos sin pasar por la API (aproximadamente 70 veces el tiempo de respuesta).

Además podemos ver como la media de bytes se incrementa considerablemente debido a que estas son peticiones HTTP y no consultas directas a la base de datos.

**API con mysql**