Daniel Granados Retana, carné 2022104692
Diego Granados Retana, carné 2022158363
Bases de Datos I
15-5-2023

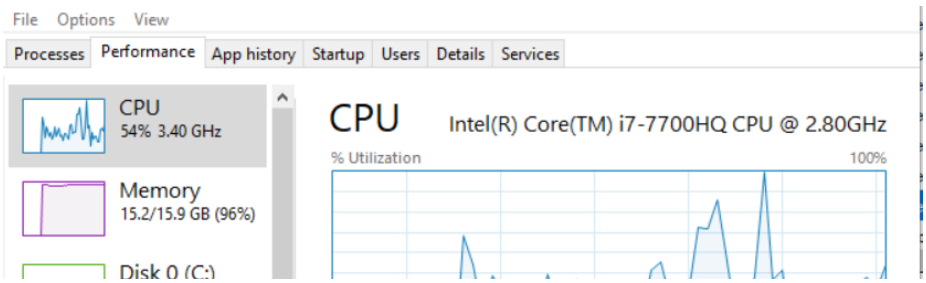
Caso #3 Preliminar #4: Resultados de medición

Para estas pruebas, se ejecutaron requests en 20 hilos donde cada request tenía un intervalo de 30ms. Para implementar el uso de un connection pool usamos el API de mssql y para no usar pool, usamos Tedious. Para implementar el ORM, utilizamos Sequelize.

Pooling: 10 min, 20 max

Primero, probamos tener un pool de 10 a 20 conexiones. La idea de esto era probar el rendimiento donde cada usuario tenía su propia conexión para trabajar. Sin embargo, vimos que esta cantidad de pools era excesiva para lo que de verdad se necesitaba. Esto generó un consumo mayor de recursos de la computadora, por lo que las consultas tenían que más bien esperar a que se liberaran recursos de hardware para poder trabajar. Así, quedaban suspendidas hasta que hubiera suficiente memoria para poder correr.

55	1	sa	ev34	SUSPENDED	SELECT	node-mssql	4990	RESOURCE_SEMAPHORE	resourceWait
56	1	sa	ev34	SUSPENDED	SELECT	node-mssql	5304	RESOURCE_SEMAPHORE	resourceWait
57	1	sa	ev34	SUSPENDED	SELECT	node-mssql	5640	RESOURCE_SEMAPHORE	resourceWait
58	1	sa	ev34	SUSPENDED	SELECT	node-mssql	4799	RESOURCE_SEMAPHORE	resourceWait
60	1	sa	ev34	SUSPENDED	SELECT	node-mssql	5356	RESOURCE_SEMAPHORE	resourceWait
61	1	sa	ev34	SUSPENDED	SELECT	node-mssql	5055	RESOURCE_SEMAPHORE	resourceWait
62	1	sa	ev34	SUSPENDED	SELECT	node-mssql	4789	RESOURCE_SEMAPHORE	resourceWait
63	1	sa	ev34	SUSPENDED	SELECT	node-mssql	4739	RESOURCE_SEMAPHORE	resourceWait
64	1	sa	ev34	SUSPENDED	SELECT	node-mssql	4651	RESOURCE_SEMAPHORE	resourceWait



Notamos que la memoria que estaba siendo utilizada era casi la totalidad disponible. Luego de probar varias veces, nos dimos cuenta que esto ocurría porque el JMeter guardaba todos los resultados de las corridas, y esto consume mucha memoria.

Esto explicaba por qué en ocasiones se suspendían las conexiones hasta conseguir los recursos necesarios para seguir. No obstante, esto está más relacionado con el hardware de la computadora o servidor.

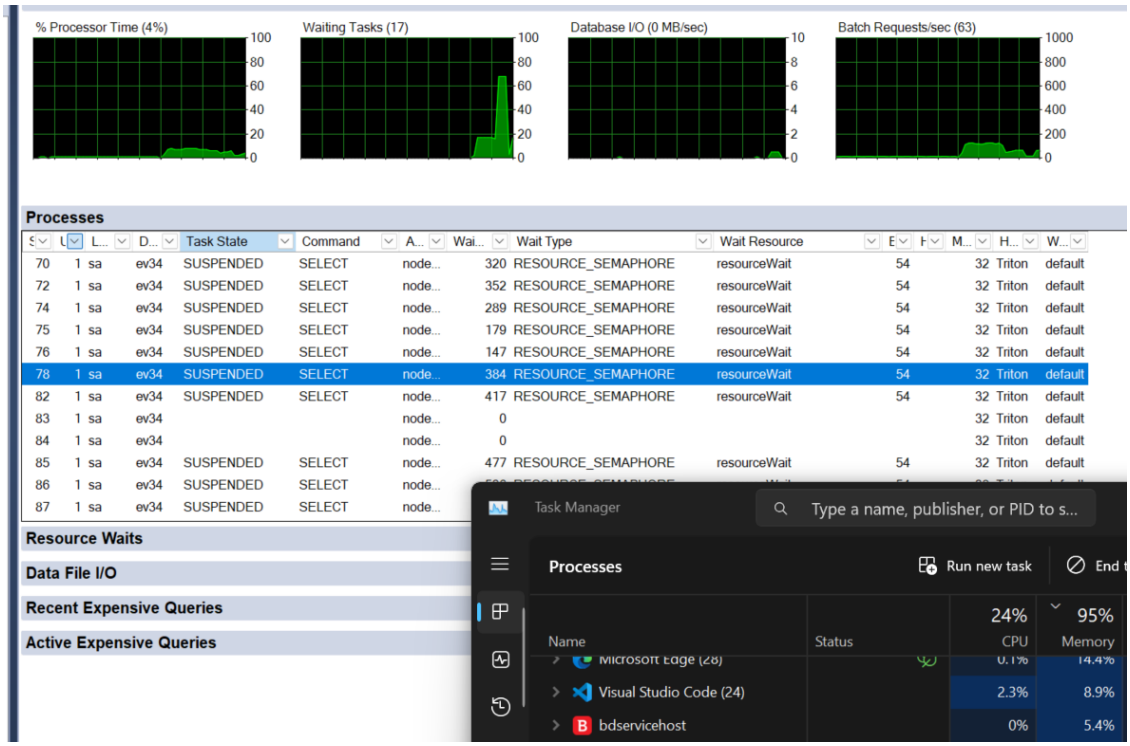
Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
1	15:57:14.522	Thread Group 1-1	HTTP Request	67		831226	137	64	2
2	15:57:14.567	Thread Group 1-2	HTTP Request	118		831226	137	117	0
3	15:57:14.619	Thread Group 1-3	HTTP Request	99		831226	137	98	1
4	15:57:14.716	Thread Group 1-5	HTTP Request	127		831226	137	126	0
5	15:57:14.666	Thread Group 1-4	HTTP Request	193		831226	137	193	1
6	15:57:14.765	Thread Group 1-6	HTTP Request	125		831226	137	123	0
7	15:57:14.815	Thread Group 1-7	HTTP Request	184		831226	137	183	1
8	15:57:14.866	Thread Group 1-8	HTTP Request	156		831226	137	155	1
9	15:57:14.914	Thread Group 1-9	HTTP Request	147		831226	137	146	1
10	15:57:14.966	Thread Group 1-10	HTTP Request	133		831226	137	133	0
11	15:57:15.018	Thread Group 1-11	HTTP Request	143		831226	137	142	1
12	15:57:15.065	Thread Group 1-12	HTTP Request	126		831226	137	125	1
13	15:57:15.116	Thread Group 1-13	HTTP Request	103		831226	137	102	1
14	15:57:15.166	Thread Group 1-14	HTTP Request	103		831226	137	102	0
15	15:57:15.217	Thread Group 1-15	HTTP Request	81		831226	137	81	1
16	15:57:15.267	Thread Group 1-16	HTTP Request	83		831226	137	82	2
17	15:57:15.315	Thread Group 1-17	HTTP Request	74		831226	137	74	1
18	15:57:15.365	Thread Group 1-18	HTTP Request	79		831226	137	78	1
19	15:57:15.417	Thread Group 1-19	HTTP Request	83		831226	137	82	1
20	15:57:15.474	Thread Group 1-20	HTTP Request	78		831226	137	77	1
21	15:57:24.594	Thread Group 1-1	HTTP Request	56		831226	137	55	1
22	15:57:24.688	Thread Group 1-2	HTTP Request	54		831226	137	52	1
23	15:57:24.719	Thread Group 1-3	HTTP Request	90		831226	137	89	1
24	15:57:24.862	Thread Group 1-4	HTTP Request	88		831226	137	87	1
25	15:57:24.846	Thread Group 1-5	HTTP Request	134		831226	137	133	1
26	15:57:24.894	Thread Group 1-6	HTTP Request	122		831226	137	116	1

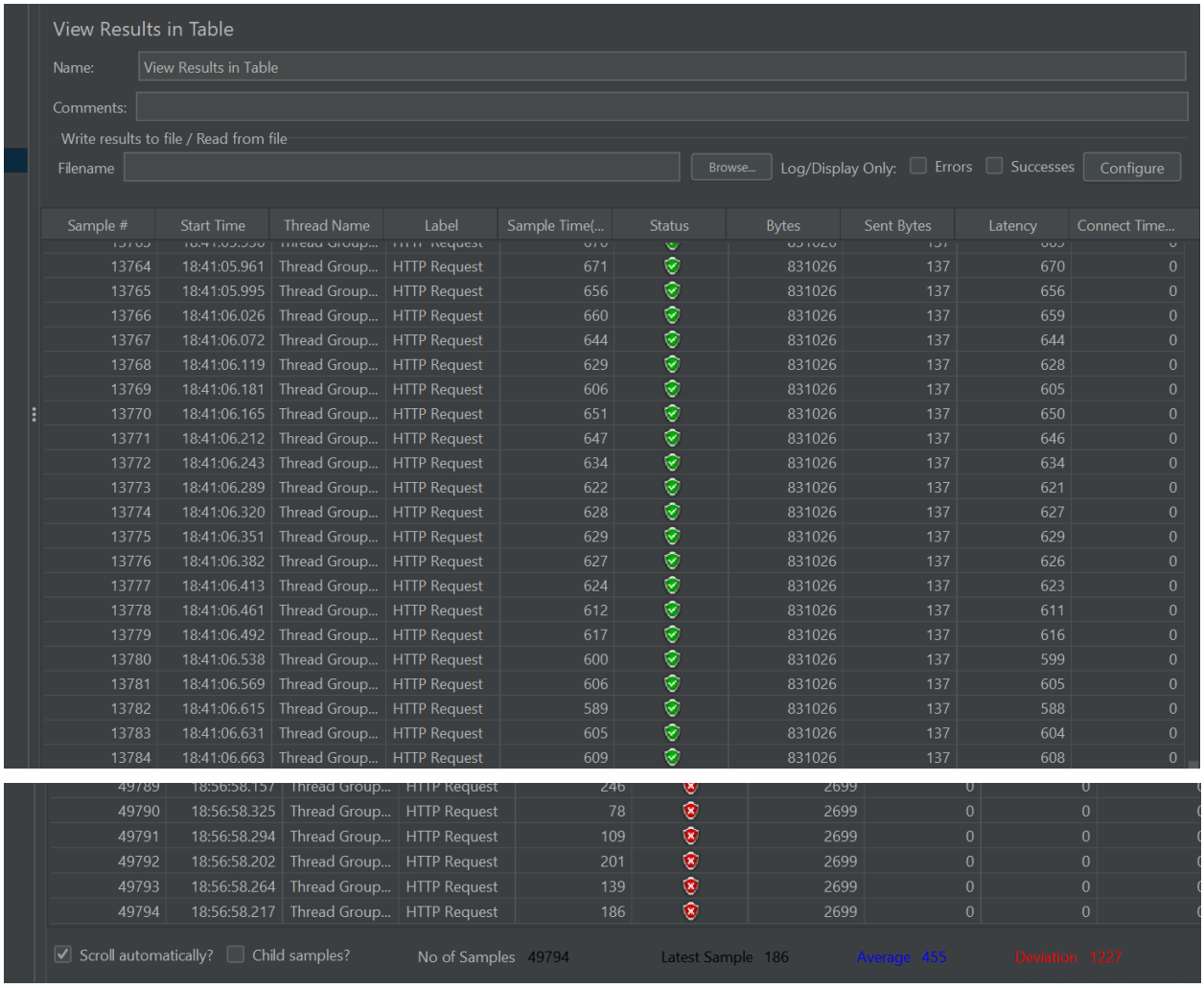
Promedio: 4750 ms

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
736	16:01:00.123	Thread Group 1-17	HTTP Request	89		831226	137	88	0
737	16:01:00.171	Thread Group 1-16	HTTP Request	70		831226	137	69	0
738	16:01:00.235	Thread Group 1-18	HTTP Request	59		831226	137	58	0
739	16:01:00.313	Thread Group 1-19	HTTP Request	54		831226	137	53	0
740	16:01:00.362	Thread Group 1-20	HTTP Request	54		831226	137	54	0
741	16:01:02.199	Thread Group 1-1	HTTP Request	25160		831226	137	25159	0
742	16:01:02.923	Thread Group 1-11	HTTP Request	24503		831226	137	24502	0
743	16:01:02.861	Thread Group 1-10	HTTP Request	24615		831226	137	24614	0
744	16:01:03.160	Thread Group 1-15	HTTP Request	24345		831226	137	24344	0
745	16:01:03.224	Thread Group 1-17	HTTP Request	24312		831226	137	24311	0
746	16:01:02.734	Thread Group 1-8	HTTP Request	24843		831226	137	24842	0
747	16:01:02.970	Thread Group 1-12	HTTP Request	24637		831226	137	24636	0
748	16:01:03.255	Thread Group 1-16	HTTP Request	24381		831226	137	24380	0
749	16:01:03.081	Thread Group 1-14	HTTP Request	24581		831226	137	24580	0
750	16:01:02.813	Thread Group 1-9	HTTP Request	24877		831226	137	24876	0
751	16:01:02.577	Thread Group 1-6	HTTP Request	25139		831226	137	25139	0
752	16:01:02.530	Thread Group 1-5	HTTP Request	25221		831226	137	25220	0
753	16:01:02.388	Thread Group 1-3	HTTP Request	25410		831226	137	25409	0
754	16:01:03.381	Thread Group 1-19	HTTP Request	24449		831226	137	24448	0
755	16:01:03.018	Thread Group 1-13	HTTP Request	24838		831226	137	24837	0
756	16:01:03.302	Thread Group 1-18	HTTP Request	24581		831226	137	24580	0
757	16:01:02.656	Thread Group 1-7	HTTP Request	25262		831226	137	25262	0
758	16:01:02.342	Thread Group 1-2	HTTP Request	25604		831226	137	25603	0
759	16:01:03.428	Thread Group 1-20	HTTP Request	24546		831226	137	24545	0
760	16:01:02.467	Thread Group 1-4	HTTP Request	25533		831226	137	25533	0
761	16:01:30.367	Thread Group 1-1	HTTP Request	71		831226	137	70	0

Cuando el tiempo de los queries subía a 25000 es cuando ocurría el error de RESOURCE SEMAPHORE y se tenía que esperar a que hubiera recursos de memoria.

Al probar en una computadora más poderosa, notamos la importancia del hardware, ya que duraba más requests en llegar al punto donde se agotaban los recursos y tenían que esperar.
Pool con mínimo 10 y máximo 20:

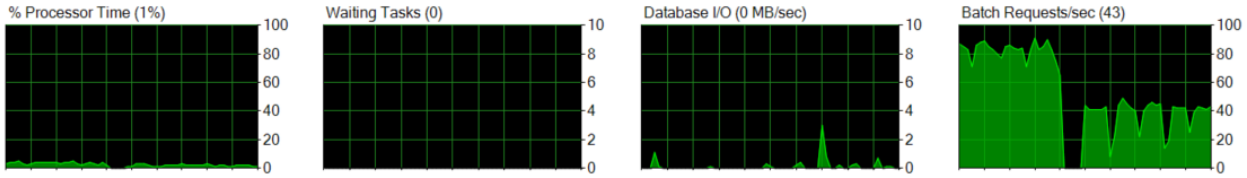


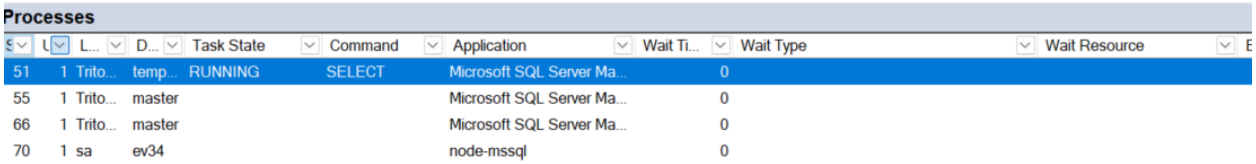


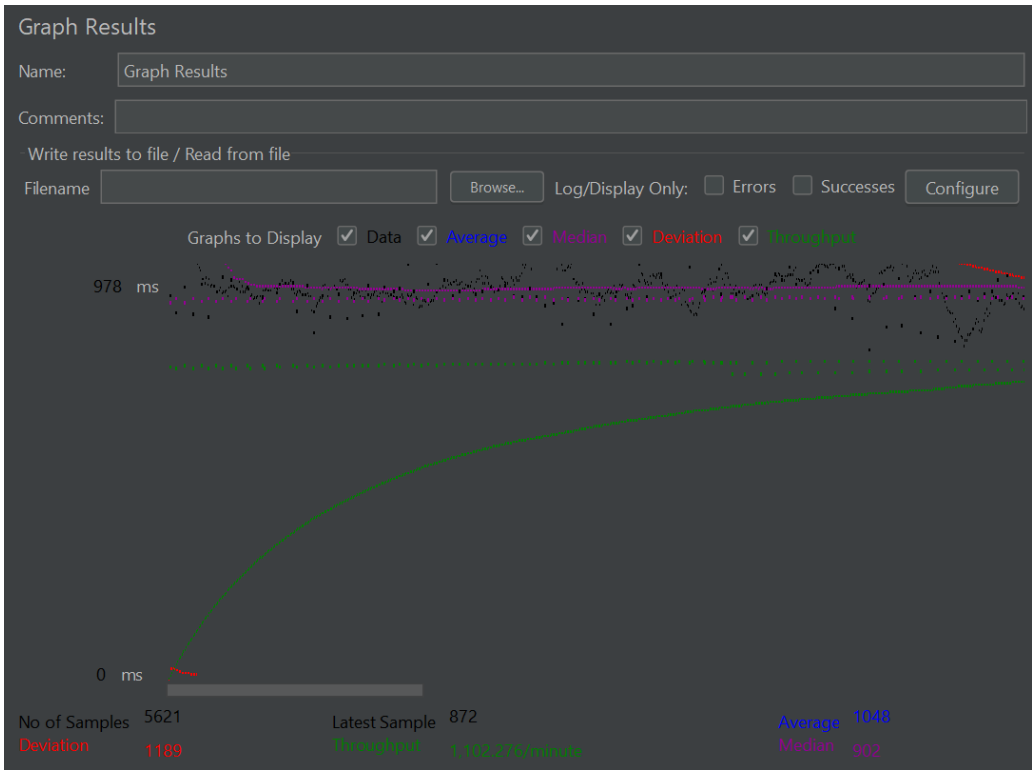
Aquí, tuvo un promedio de 455 ms y se empezó a bloquear llegando a los 50000 requests.

Pruebas con pools de diferente tamaño:

- Min 1 Max 1

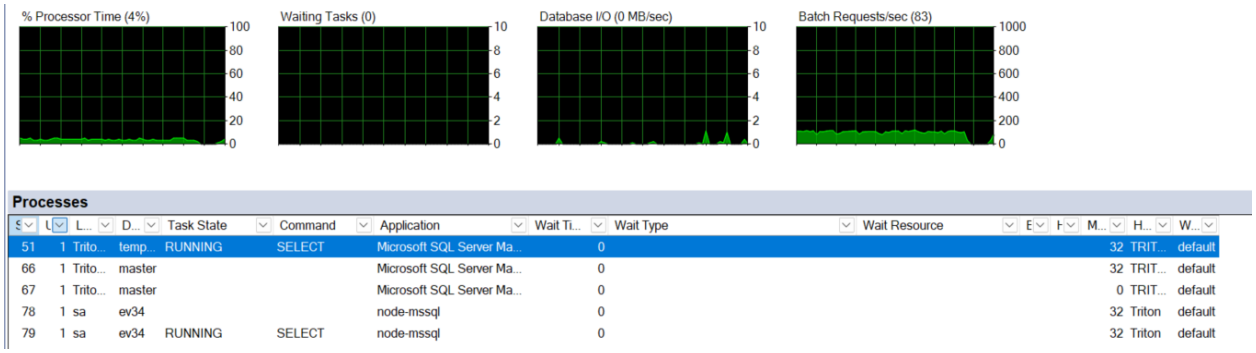


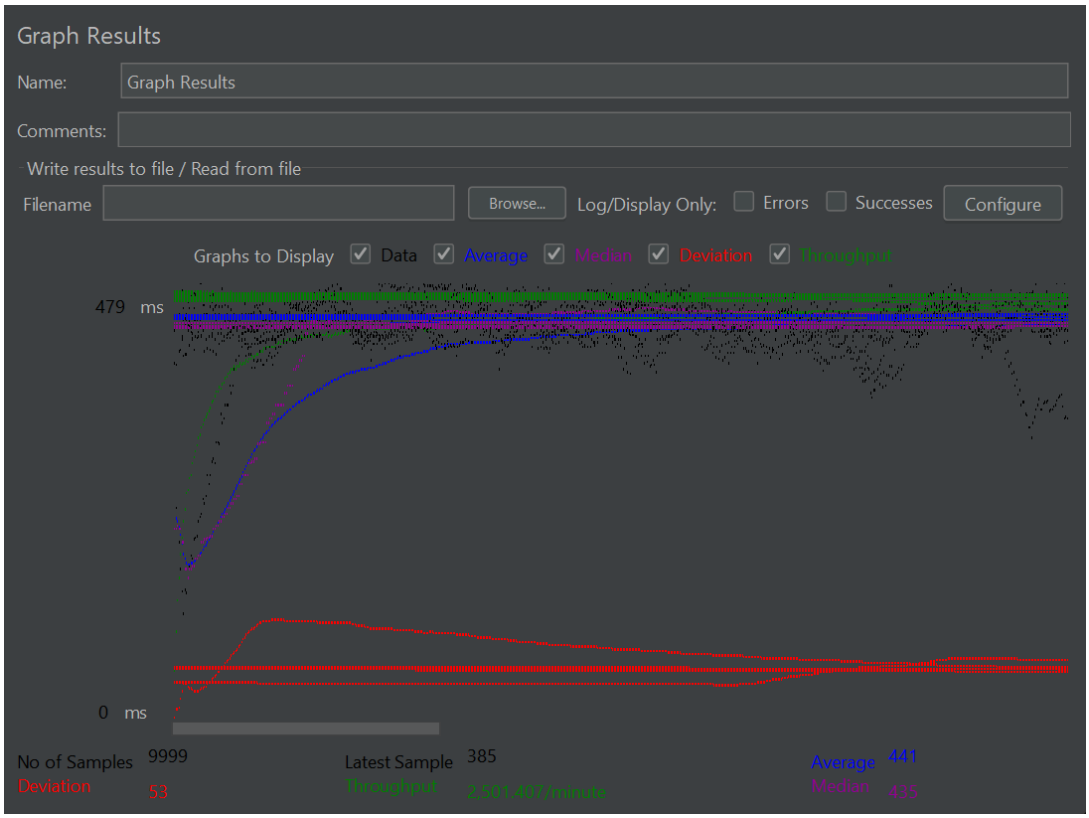




Promedio: 1048 ms

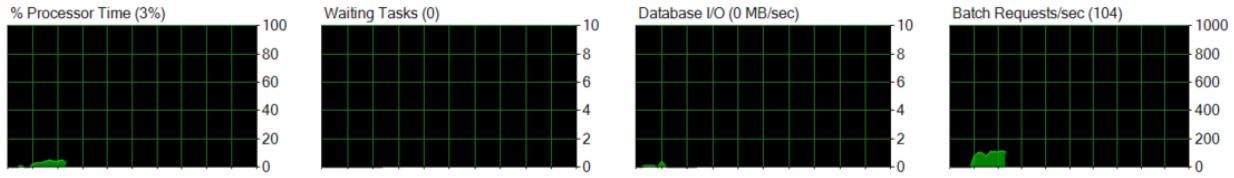
Min 1 Max 2





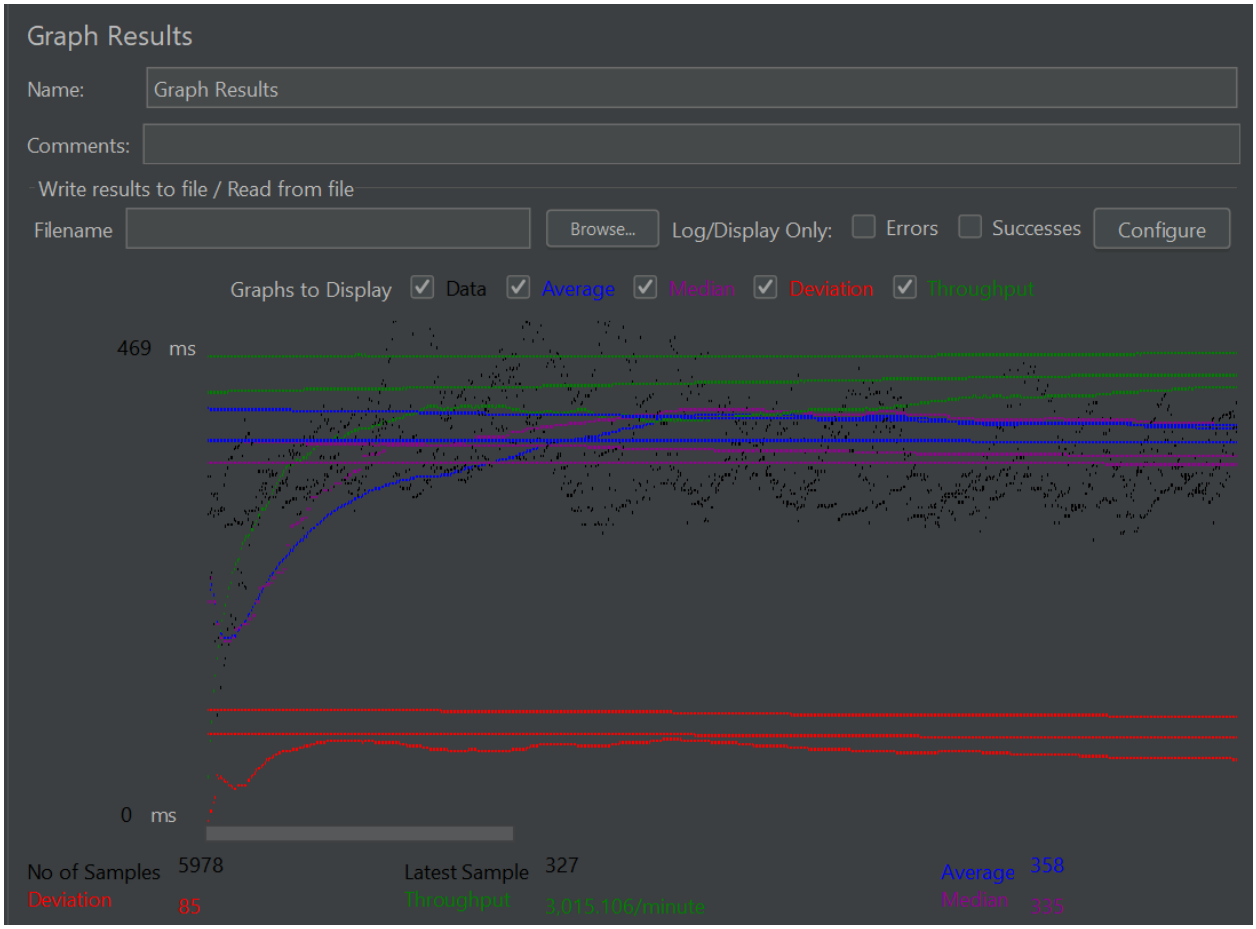
Promedio: 441 ms

Min 1 max 3



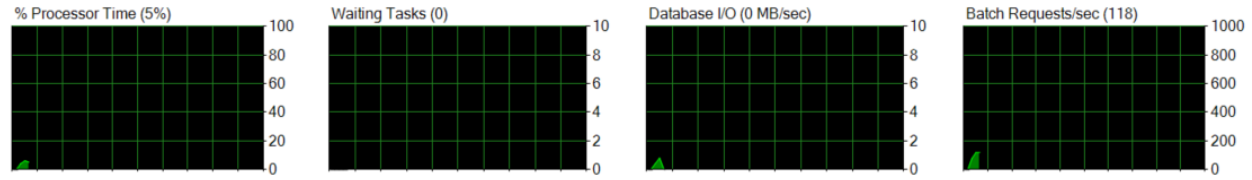
Processes

S	U	L...	D...	Task State	Command	Application	Wait Ti...	Wait Type	Wait Resource	E
51	1	Trito...	temp...	RUNNING	SELECT	Microsoft SQL Server Ma...	0			
66	1	Trito...	master			Microsoft SQL Server Ma...	0			
67	1	Trito...	master			Microsoft SQL Server Ma...	0			
78	1	sa	ev34			node-mssql	0			
79	1	sa	ev34			node-mssql	0			
80	1	sa	ev34	RUNNING	SELECT	node-mssql	0			



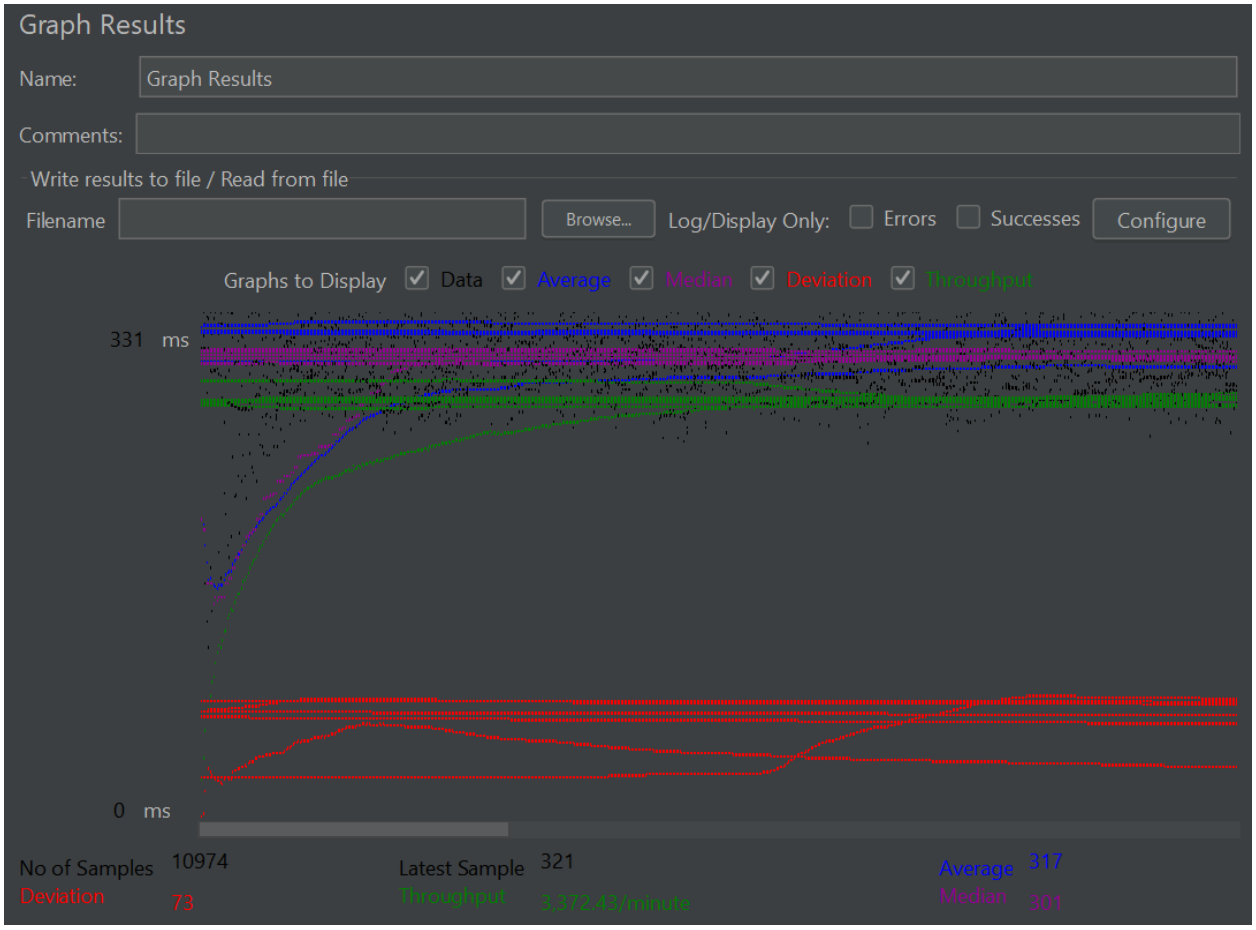
Promedio: 358 ms

Min 1 max 4



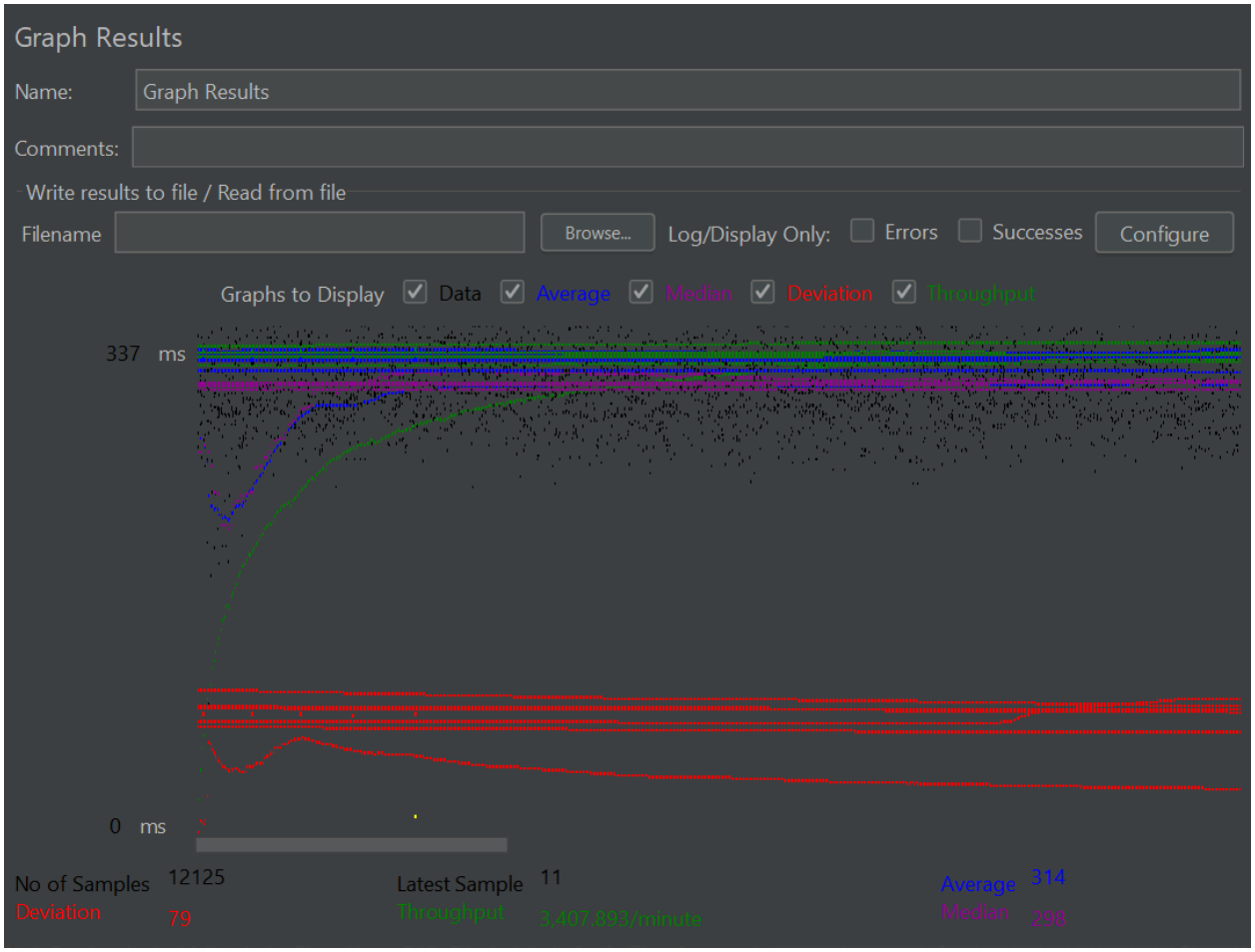
Processes

S	U	L...	D...	Task State	Command	Application	Wait Ti...	Wait Type	Wait Resource	E
51	1	Trito...	temp...	RUNNING	SELECT	Microsoft SQL Server Ma...	0			
54	1	sa	ev34			node-mssql	0			
55	1	sa	ev34	RUNNING	SELECT	node-mssql	0			
64	1	sa	ev34	RUNNING	SELECT	node-mssql	0			
65	1	sa	ev34			node-mssql	0			

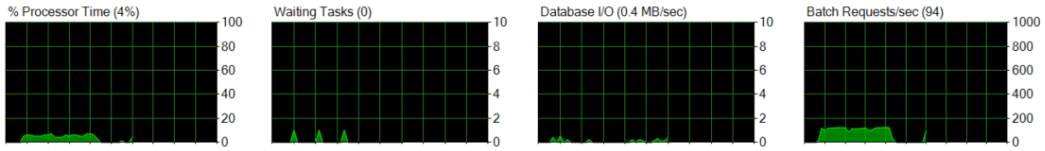


Promedio: 317 ms

Min 1 Max 5



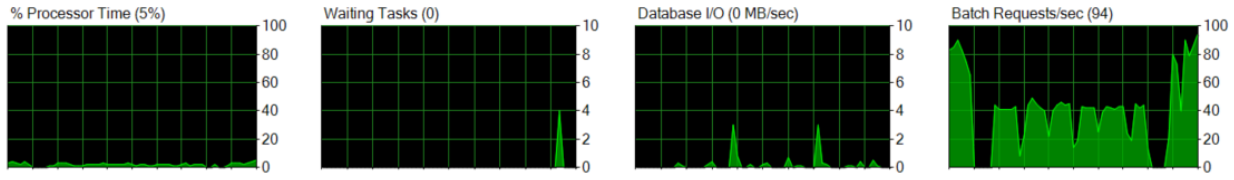
Promedio: 314 ms



Processes

S	U	L...	D...	Task State	Command	Application	Wait Ti...	Wait Type	Wait Resource	E	H	M...	H...	W...
51	1	Trito...	master			Microsoft SQL Server Ma...	0					0	TRIT...	default
54	1	sa	ev34			node-mssql	0					32	Triton	default
55	1	sa	ev34			node-mssql	0					32	Triton	default
64	1	sa	ev34			node-mssql	0					32	Triton	default
65	1	sa	ev34			node-mssql	0					32	Triton	default
66	1	Trito...	master			Microsoft SQL Server Ma...	0					32	TRIT...	default
67	1	sa	ev34			node-mssql	0					32	Triton	default

Min 1 Max 8

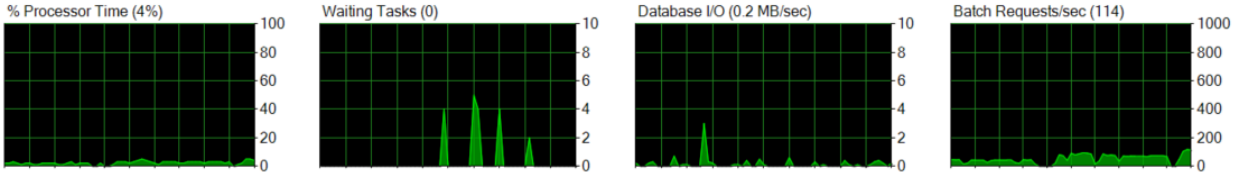


Processes

S	U	L...	D...	Task State	Command	Application	Wait Ti...	Wait Type	Wait Resource	E
51	1	Trito...	temp...	RUNNING	SELECT	Microsoft SQL Server Ma...	0			
54	1	sa	ev34			node-mssql	0			
55	1	Trito...	master			Microsoft SQL Server Ma...	0			
64	1	sa	ev34			node-mssql	0			
65	1	sa	ev34			node-mssql	0			
66	1	Trito...	master			Microsoft SQL Server Ma...	0			
68	1	sa	ev34			node-mssql	0			
69	1	sa	ev34	RUNNING	SELECT	node-mssql	0			
70	1	sa	ev34	RUNNING	SELECT	node-mssql	0			
72	1	sa	ev34			node-mssql	0			
74	1	NT S...	master			SQLServerCEIP	0			
75	1	sa	ev34			node-mssql	0			

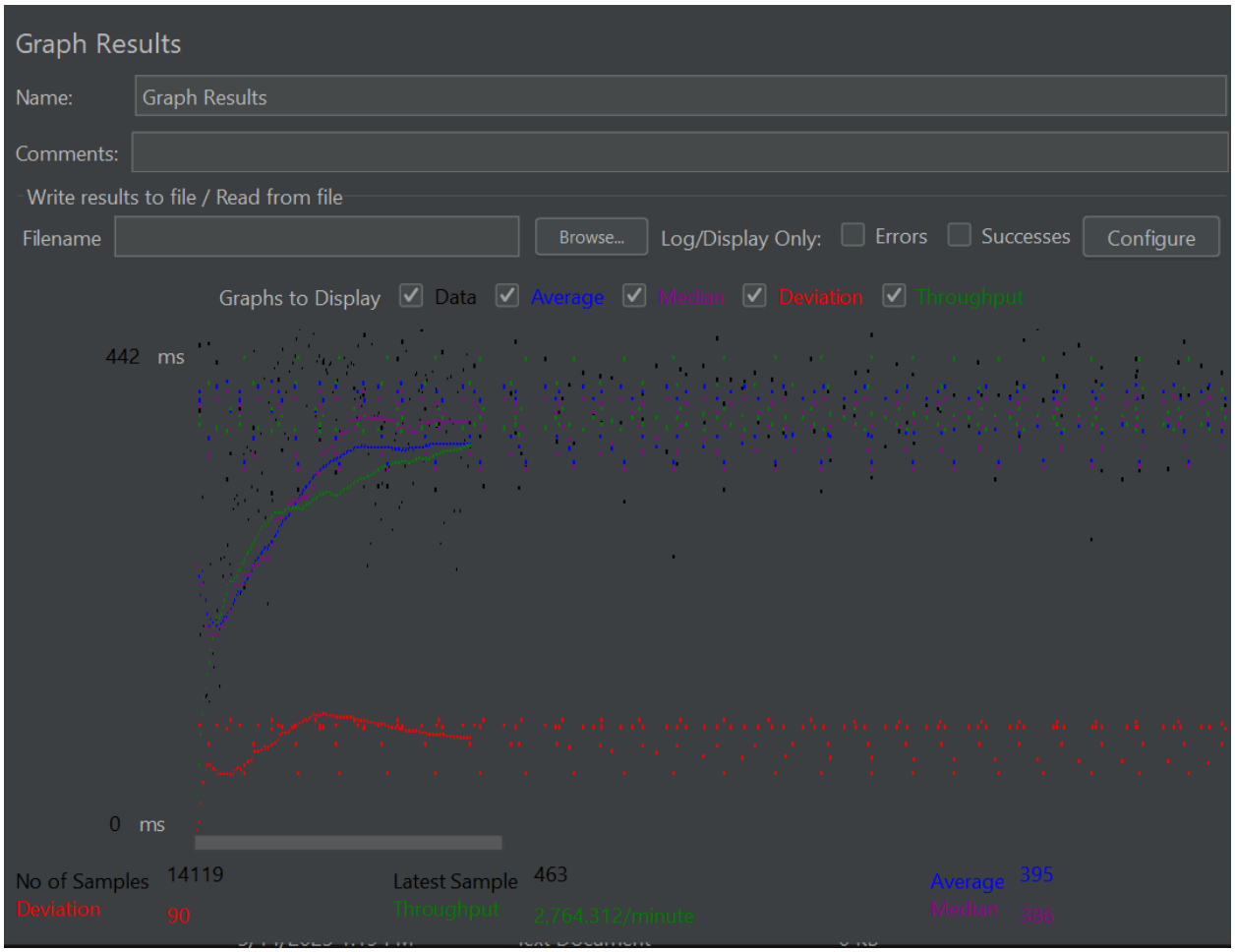
Promedio: 550 ms

Min 1 Max 10:



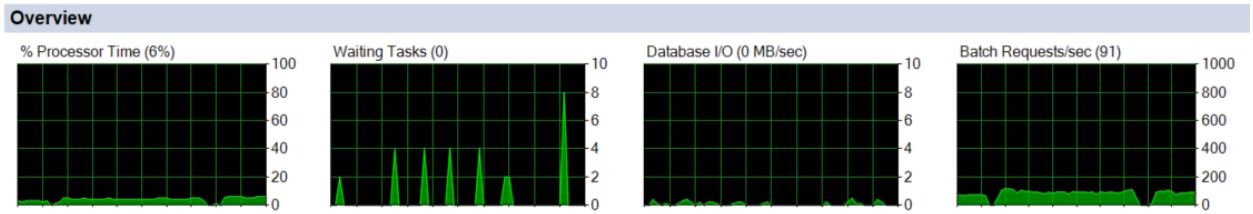
Processes

S	User Pr...	L...	D...	Task State	Command	Application	Wait Ti...	Wait Type	Wait Resource
54	1	sa	ev34			node-mssql	0		
64	1	sa	ev34			node-mssql	0		
65	1	sa	ev34	RUNNING	SELECT	node-mssql	0		
68	1	sa	ev34	RUNNING	SELECT	node-mssql	0		
69	1	sa	ev34			node-mssql	0		
70	1	sa	ev34			node-mssql	0		
72	1	sa	ev34			node-mssql	0		
75	1	sa	ev34			node-mssql	0		
76	1	sa	ev34	RUNNING	SELECT	node-mssql	0		
77	1	sa	ev34			node-mssql	0		



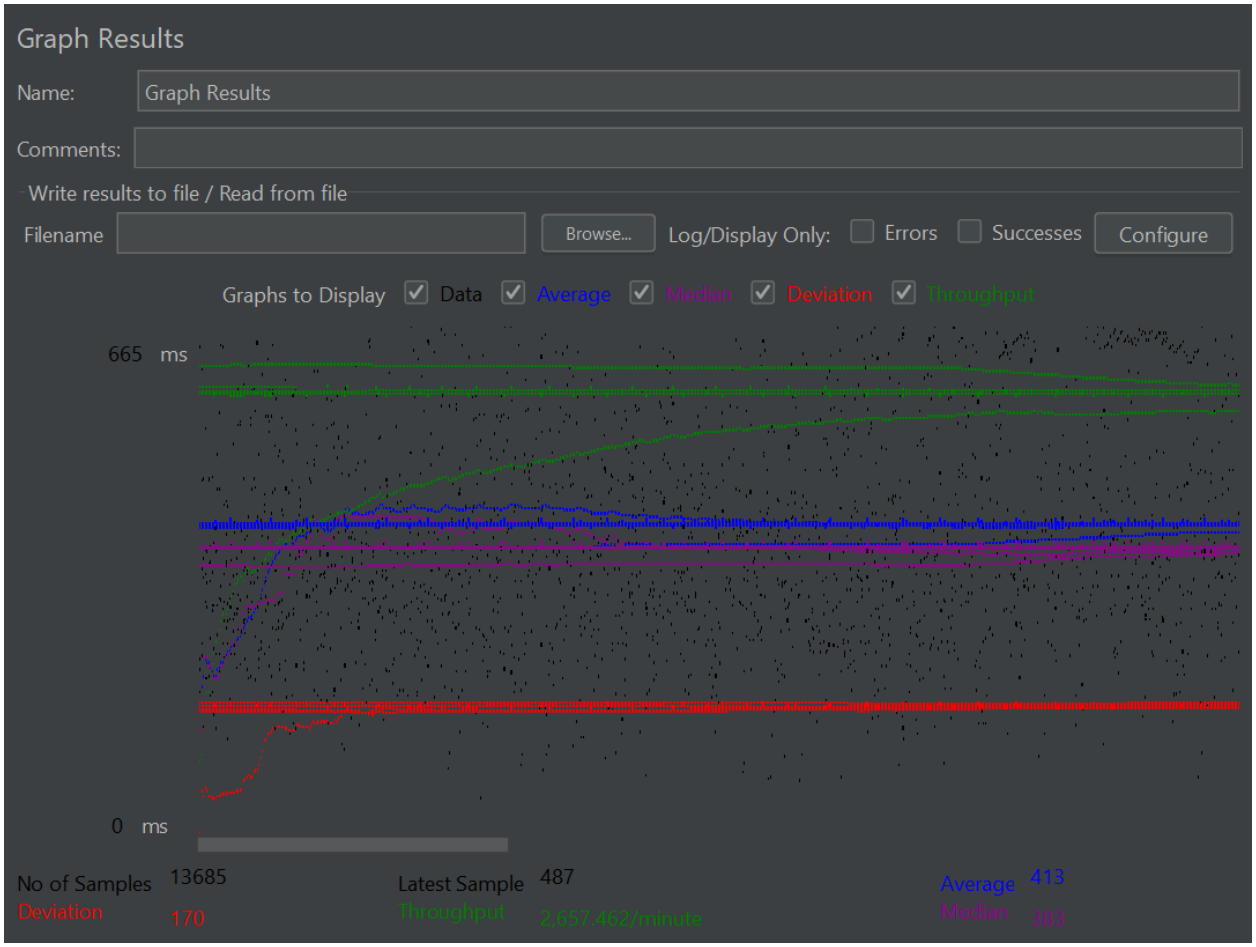
Promedio: 395 ms

Min 10 Max 20:



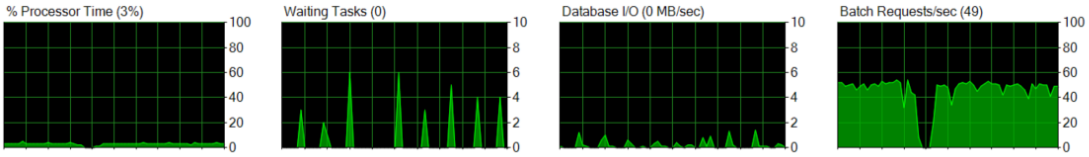
Processes

S	User Pr...	L...	D...	Task State	Command	Application	Wait Ti...	Wait Type	Wait Resource
54	1	sa	ev34	RUNNING	SELECT	node-mssql	0		
64	1	sa	ev34	RUNNING	SELECT	node-mssql	0		
65	1	sa	ev34	RUNNABLE	SELECT	node-mssql	0		
68	1	sa	ev34	RUNNING	SELECT	node-mssql	0		
69	1	sa	ev34	RUNNING	SELECT	node-mssql	0		
70	1	sa	ev34	RUNNING	SELECT	node-mssql	0		
72	1	sa	ev34			node-mssql	0		
75	1	sa	ev34	RUNNING	SELECT	node-mssql	0		
76	1	sa	ev34	RUNNING	SELECT	node-mssql	0		
77	1	sa	ev34	RUNNING	SELECT	node-mssql	0		
79	1	sa	ev34			node-mssql	0		
84	1	sa	ev34	RUNNING	SELECT	node-mssql	0		



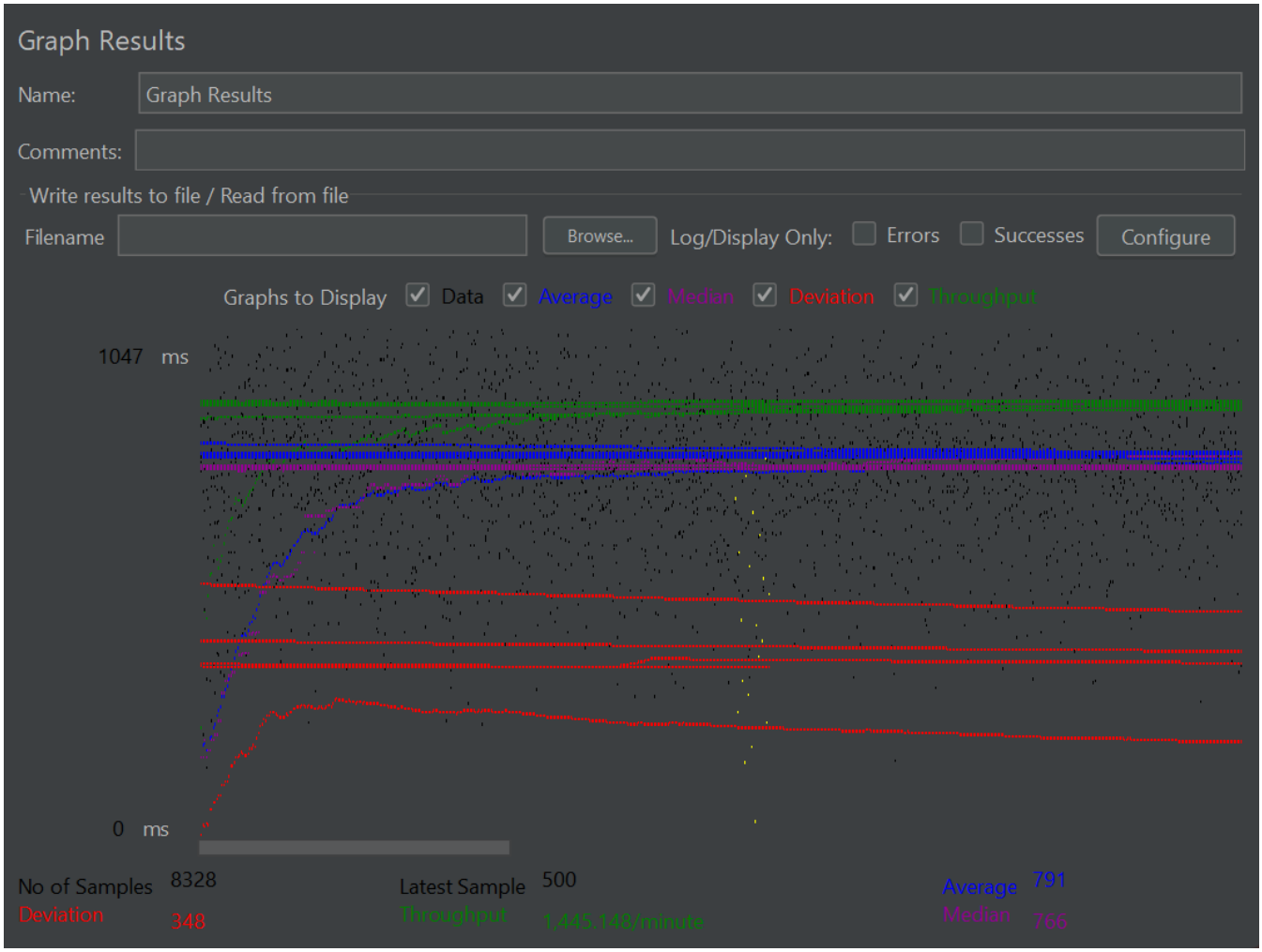
Promedio: 413 ms

Pruebas sin pool de conexiones:



Processes

User Pr...	L...	D...	Task State	Command	Application	Wait Ti...	Wait Type	Wait Resource	E	H	M...	H...	W...
64	1	sa	ev34		Tedious	0					32	Triton	default
67	1	sa	ev34		Tedious	0					32	Triton	default
68	1	sa	ev34		Tedious	0					32	Triton	default
69	1	sa	ev34		Tedious	0					32	Triton	default
70	1	sa	ev34		Tedious	0					32	Triton	default
72	1	sa	ev34		Tedious	0					32	Triton	default
74	1	sa	ev34		Tedious	0					32	Triton	default
76	1	sa	ev34		Tedious	0					32	Triton	default
77	1	sa	ev34		Tedious	0					32	Triton	default
78	1	sa	ev34		Tedious	0					32	Triton	default
79	1	sa	ev34		Tedious	0					32	Triton	default
81	1	sa	ev34		Tedious	0					32	Triton	default



Promedio: 791 ms

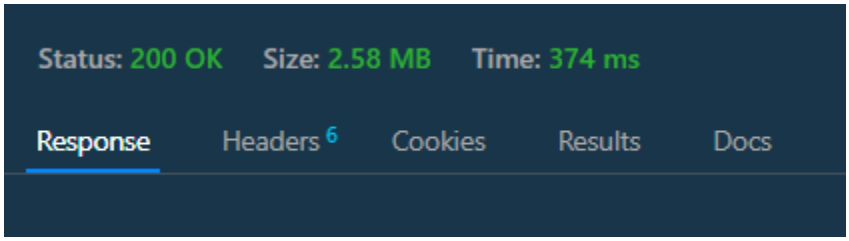
Conclusiones

Con base en estos resultados, podemos observar que con 5 conexiones nos da los mejores resultados, con un promedio de 314 ms por request. Con 5 conexiones, dura menos que cuando hay menos conexiones, como 1 o 2, porque es capaz de resolver más requests a la vez. Menos requests quedan en espera en una cola. No obstante, cuando hay más de 5 conexiones, dura más porque se pueden estar gastando recursos de una forma no tan óptima. El hecho de levantar más conexiones y mantenerlas activas implica un overhead mayor. Adicionalmente, tener menos conexiones hace que el pool se enfoque en reutilizar las que ya tiene abiertas en lugar de crear nuevas. Esto también ayuda a reducir este procesamiento adicional. Otra razón puede ser que hay un número óptimo de hilos que se pueden correr al mismo tiempo. Por ejemplo, puede ser que la base de datos solo pueda correr 5 hilos simultáneamente en diferentes cores del procesador, entonces cuando se suman más hilos, puede ser que se tenga que hacer un procedimiento de concurrencia e ir intercambiando el hilo que se ejecuta. Esto causa que ambos hilos duren más tiempo en terminar. También notamos que se gastaba el RAM de la computadora más rápido, pero esto puede ser porque seguíamos almacenando los resultados de cada request. Por lo tanto, encontramos que el tamaño de 5 conexiones resultó óptimo para recibir los requests entrantes y procesarlos en un menor tiempo.

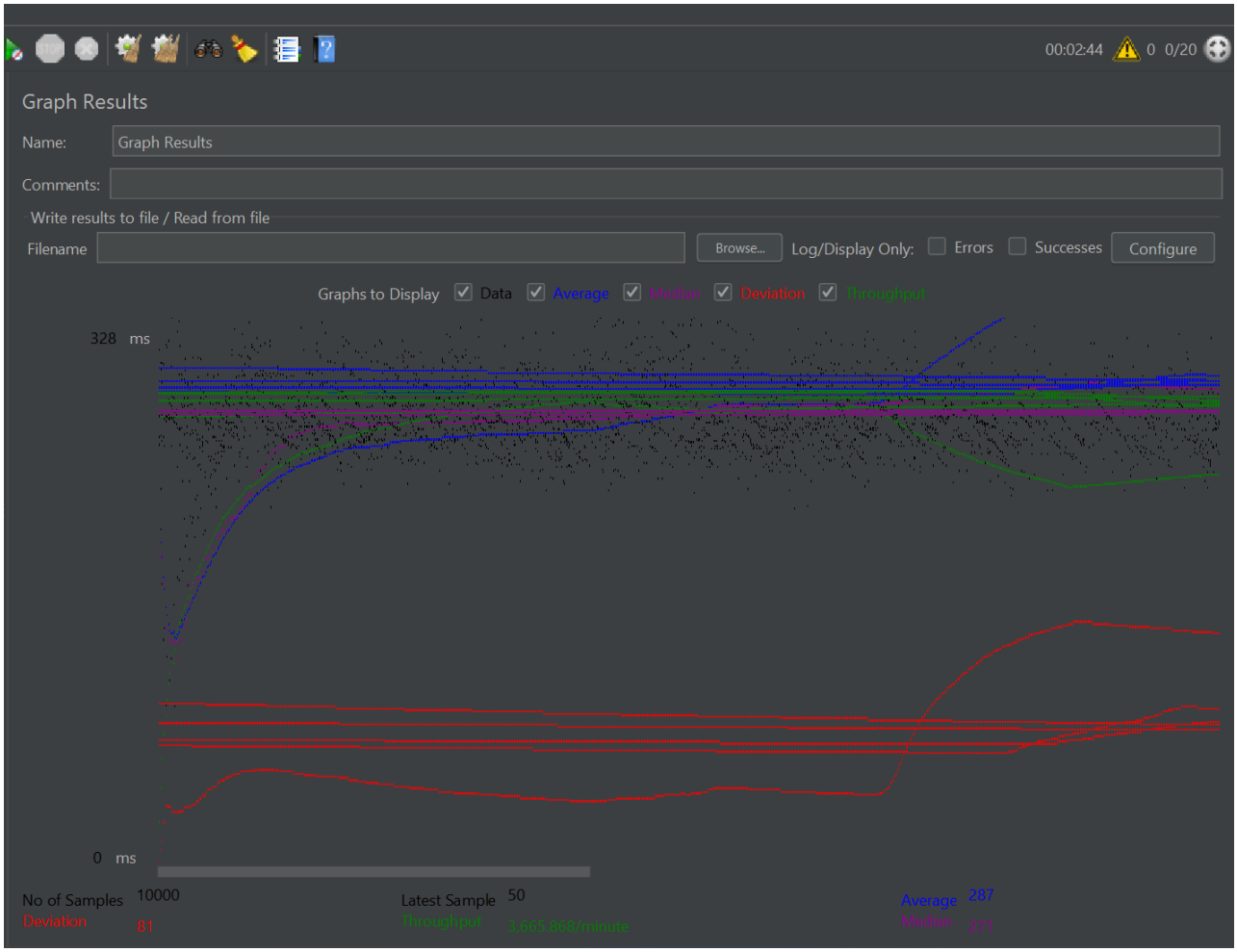
Al comparar los resultados de usar un pool versus no usar uno, obtuvimos que para esta situación, es más eficiente utilizar un pool. Esto puede ser porque usar un connection pool reduce el overhead de levantar una conexión y cerrarla cada vez que se hace un request. Más bien, reutiliza las conexiones que ya tiene abiertas. Esto representa un mejor uso de los recursos, ya que se puede tener un mayor control de las conexiones que están abiertas y de los recursos que la aplicación usa. Adicionalmente, no usar un pool abre muchas conexiones a la vez, cada una con un hilo de ejecución. Esto puede representar una mayor carga para el procesador, por lo que tiene que realizar la ejecución de forma intercalada. Estas podrían ser razones por las que no usar pool tuvo un peor rendimiento que usar un pool.

Otro factor que pudo tener incidencia entre usar pool y no usar pool es el tamaño de lo que retornaba cada resultado. Ambos retornan el mismo conjunto de registros de la base de datos, pero el formato en que lo almacenan y las cosas que guardan son diferentes. Para el connection pool, usamos mssql, y el tamaño de lo que retornaba eran 811.51KB:

Sin el connection pool, usamos Tedious: el cual retorna un json con más información de metadatos, por lo que el tamaño es mayor, de 2.58MB:

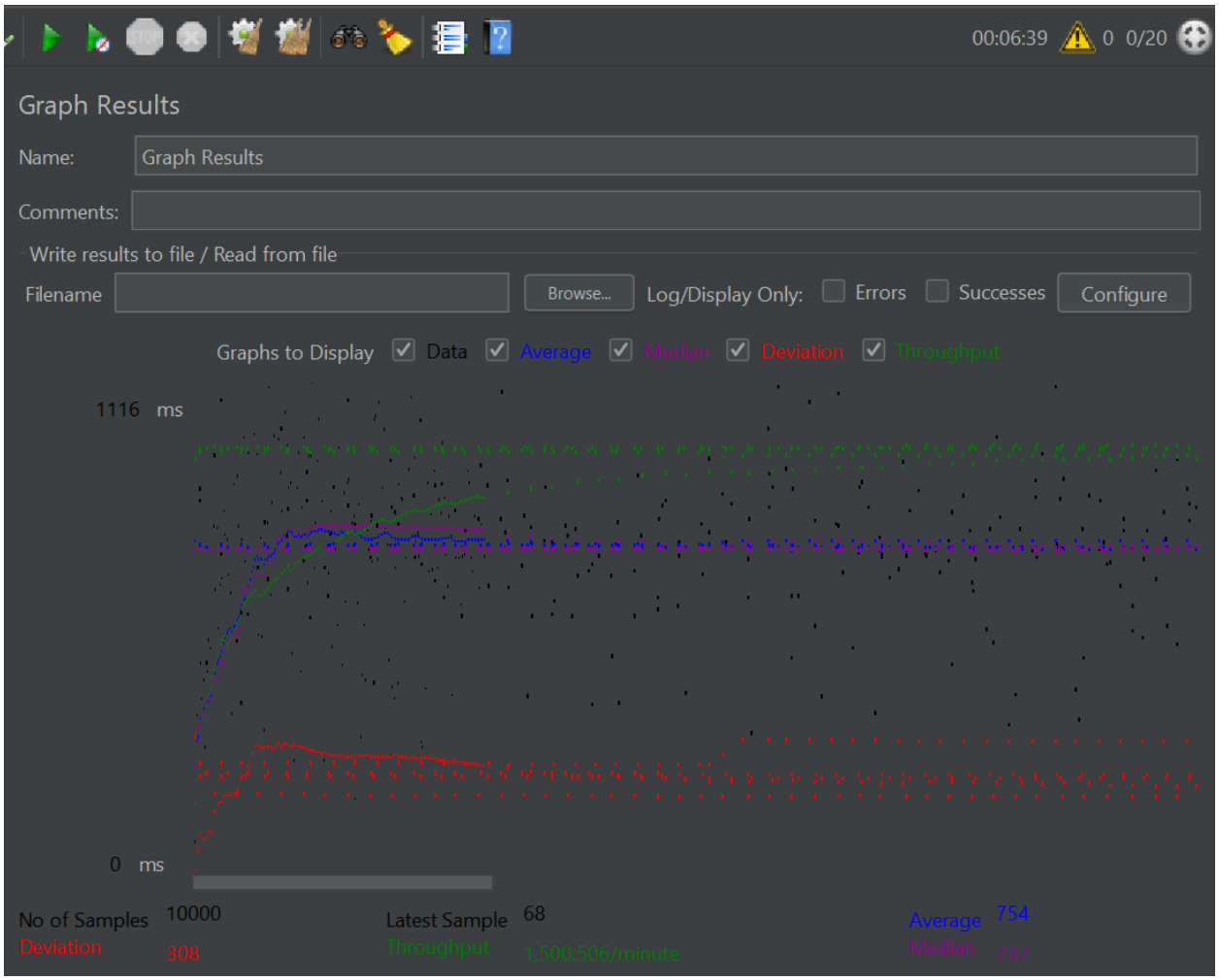


Comparación de ejecutar 500 iteraciones en 20 hilos:
Con pool (1 min / 5 max):



Promedio: 287 ms y duración de 2:44 minutos

Sin pool:



Promedio: 754ms y duración de 6:39 minutos.

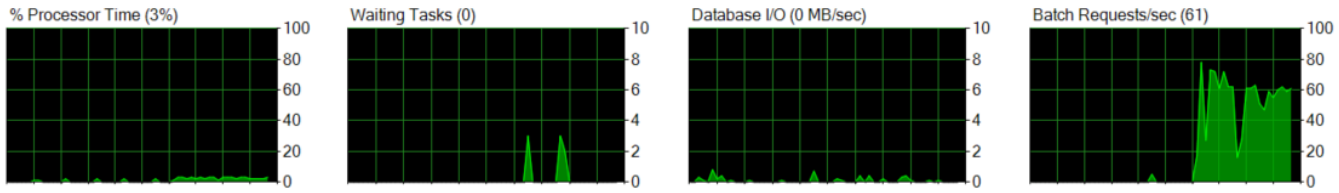
ORM

El ORM fue implementado en Node.js con la herramienta de Sequelize.

Los resultados de implementar el endpoint por medio de ORM son los mismos.

```
Connection has been established successfully.
Executing (default): SELECT 1+1 AS result
Executing (default): SELECT [lotesProduccionLogs].[loteId], [lotesProduccionLogs].[fecha], [producto->nombre].[nombreBase] AS [productoNombre], [lotesProduccionLogs].[prodContratoId], [lotesProduccionLogs].[plantaId], lotesProduccionLogs.cantidad - COALESCE(sum(itemsProductos.cantidadProductos), 0) AS [cantidad], lotesProduccionLogs.costoProduccion / costoMonedas.conversion AS [costoProduccion], preciosProductosContrato.precio / [preciosProductosContrato->precioMonedas].conversion AS [precio] FROM [lotesProduccionLogs] AS [lotesProduccionLogs] LEFT OUTER JOIN [itemsProductos] AS [itemsProductos] ON [lotesProduccionLogs].[loteId] = [itemsProductos].[loteId] INNER JOIN [productos] AS [producto] ON [lotesProduccionLogs].[productoId] = [producto].[productoId] INNER JOIN [nombres] AS [producto->nombre] ON [producto].[nombreId] = [producto->nombre].[nombreId] INNER JOIN [tiposDeCambio] AS [costoMonedas] ON [lotesProduccionLogs].[monedaId] = [costoMonedas].[monedaCambioId] INNER JOIN [preciosProductosContrato] AS [preciosProductosContrato] ON [lotesProduccionLogs].[prodContratoId] = [preciosProductosContrato].[prodContratoId] AND [preciosProductosContrato].[productoId] = [lotesProduccionLogs].[productoId] INNER JOIN [tiposDeCambio] AS [preciosProductosContrato->precioMonedas] ON [preciosProductosContrato].[monedaId] = [preciosProductosContrato->precioMonedas].[monedaCambioId] GROUP BY [lotesProduccionLogs].[loteId], [lotesProduccionLogs].[fecha], [producto->nombre].[nombreBase], [lotesProduccionLogs].[cantidad], [lotesProduccionLogs].[prodContratoId], [lotesProduccionLogs].[plantaId], [lotesProduccionLogs].[costoProduccion], [costoMonedas].[conversion], [preciosProductosContrato].[precio], [preciosProductosContrato->precioMonedas].[conversion] ORDER BY [precio] ASC, [lotesProduccionLogs].[fecha] ASC;
```

Este es el query que genera el ORM.

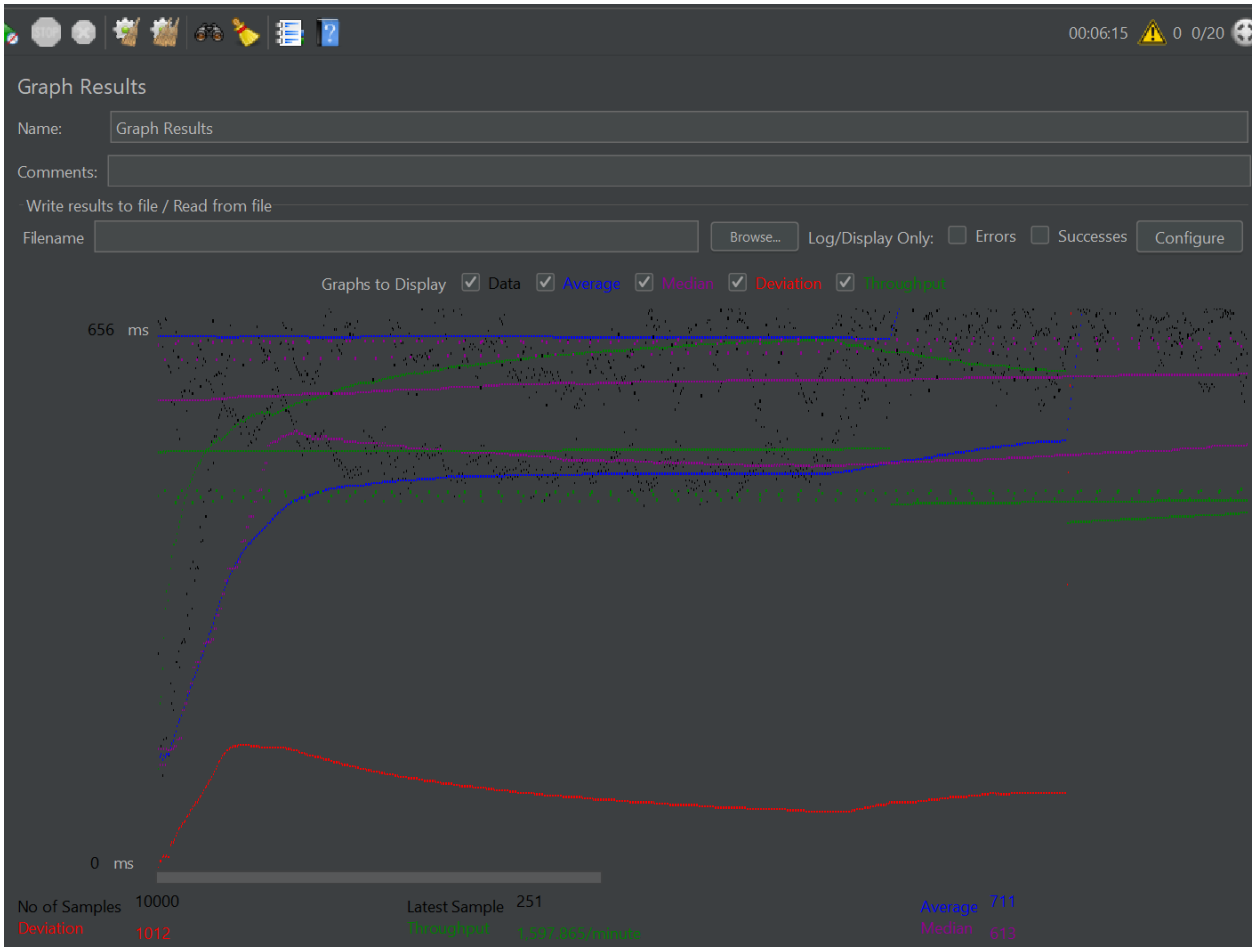


Processes

S	User Pr...	Login	Database	Task State	Command	Application	Wait Ti...	Wait Type
76	1	sa	ev34			Tedious	0	
77	1	sa	ev34			Tedious	0	
78	1	sa	ev34			Tedious	0	
79	1	sa	ev34			Tedious	0	
80	1	sa	ev34			Tedious	0	

Aquí se muestran las conexiones que están abiertas y los requests que le están llegando al servidor de SQL Server.

Al realizar la misma prueba, con 500 iteraciones en 20 hilos, obtuvimos los siguientes resultados:



Promedio: 711ms y 6:15 minutos.

A partir de esto, obtenemos que la conexión con pool sigue siendo la más rápida, seguida de la conexión con ORM, y finalmente, la conexión sin pool. Los resultados de hacerlo por medio un

ORM y hacerlo sin pool son muy similares, probablemente porque ambos están basados sobre el API de Tedious. Como la implementación por medio de ORM fue ligeramente más rápida, es posible que para este caso, el uso de ORM no haya representado un gasto mayor de recursos y procesamiento. Probablemente, Sequelize ya tiene una implementación lo más óptima posible de la conexión a la base de datos, ya que notamos que solo abría cinco conexiones, como en la prueba con pool que nos dio mejores resultados.