

Ejecución de comandos SQL asíncronos en serie y en paralelo

Por: Mauricio Cotes

Junio 16 de 2019

Resumen

Este texto intenta explicar la implementación de una forma simple de ejecutar comandos SQL sin tener que esperar a que estos terminen. Es especialmente útil para comandos de larga duración en su ejecución, donde no es conveniente que el usuario se quede esperando a su finalización, así, el comando es enviado y el control se devuelve al programa que lo invoca de modo que este pueda proseguir con su ejecución. No obstante, en el modelo que aquí se presenta, el programa que los invoca debe eventualmente, preguntar por el estado de las ejecuciones de los comandos para verificar si han concluido. La invocación del comando representa el envío de un mensaje al sistema de ejecución asíncrona, el cual debe ser procesado en algún momento posteriormente. La invocación constituye la primera parte del procesamiento. La segunda parte, consiste en el procesamiento en sí de los comandos, el cual se lleva a cabo en el mismo orden en que fueron invocados, es decir a la manera de una cola.

Asumiendo la metáfora de una cola, si solo hay un punto de atención, los comandos se procesan de forma serial y un segundo comando debe esperar a que finalice la atención del primero. Si hay varios puntos de atención para una cola, varios comandos pueden ser atendidos simultáneamente en paralelo, pero con la condición que estos no pueden ser dependientes entre sí, o sea que el segundo comando no debe necesitar que se ejecute antes el primero. Este modo de atención en paralelo disminuye dramáticamente el tiempo de espera promedio de los elementos en la cola.

La implementación que se describe fue realizada sobre SQL Server 2017, aunque la idea general podría ser implementada en cualquier entorno de programación. SQL Server no incluye esta funcionalidad asíncrona out-of-the-box, aunque proporciona varias posibilidades de infraestructura para ser implementada. Algunas de ellas incluyen: SQL Server Service Broker, SQL Server Agent y quizás, con un poco más de esfuerzo, SQL Server Integration Services. La implementación presentada en el texto actual se basa en SQL Server Agent.

Introducción e instalación

La ejecución de comandos SQL asíncronos puede ser de gran utilidad en ambientes empresariales que involucren bases de datos. El modelo que se presenta es simple, en tanto que puede ser utilizado sin mayores requerimientos. Para su uso, basta con ejecutar el script "[Ejecución de comandos asíncronos en T-SQL.sql](#)" sobre una base de datos SQL Server que se designe para este propósito. En el numeral de **Arquitectura del sistema de ejecución asíncrona**, se revisa con más detalle la forma en que se ha construido el sistema que se presenta, pero se ha preferido explicar antes como puede usarse el sistema.

Antes de empezar es necesario hacer algunas consideraciones en cuanto a los requerimientos del sistema:

- Versiones soportadas: SQL Server 2016 o posterior. Para versiones anteriores es posible usarlos con muy pocos ajustes.
- Se requiere que SQL Server Agent esté activo para la instancia de SQL Server donde se instale el sistema.
- La propiedad *Containment type* de la base de datos donde se instale debe ser 'NONE'. De aquí se deduce que el sistema de ejecución asíncrona tiene dependencias externas y, en efecto, se basa en el subsistema del SQL Server Agent y en la correspondiente base de datos msdb.

El **procedimiento de instalación**, como ya se mencionó, consiste en ejecutar el Script "Ejecución de comandos asíncronos en T-SQL.sql" sobre la base de datos SQL Server que se quiera habilitar para este efecto. Esta acción instalará los siguientes elementos sobre la base de datos¹:

- Esquema: async (todos los objetos del sistema quedan bajo este esquema).
- Tablas: async.SpoolConfiguration y async.CommandsHistory.
- Procedimientos almacenados:
 - DropSpool
 - ExecuteCmd
 - PollPendingCommands
 - ProcessSpooledCmd
 - SetSpoolConfiguration
 - StartAgents
 - StopAgents
- Funciones *Table-valued*:
 - GetExecutedCommands
 - GetCmdExecStatus
 - GetPendingCommands

¹ El sistema ha sido programado en inglés en su totalidad debido a que es la lengua de las ciencias de la computación y el sistema hace uso de varios patrones de arquitectura que son bien conocidos en el ámbito de estas ciencias. No obstante, los mensajes que devuelve el sistema al usuario están escritos en español de Colombia.

- Funciones escalares:
 - NumOfAgentsRunning
 - NumOfPendingCommands

Se ha intentado hacer la explicación comenzando desde lo simple y poco a poco ir incorporando nuevos elementos, como se verá en los siguientes numerales.

Uso simple de comandos asíncronos: serial y en paralelo

Son muchos los escenarios donde puede utilizarse este sistema y solo están limitados por el ingenio del arquitecto o desarrollador que los use.

Ejecución serial

El primer escenario a revisar consiste en la ejecución de un comando² asíncrono de forma serial. Supóngase que existe un procedimiento almacenado que realiza un complejo proceso de cálculo que toma un tiempo considerable (como un proceso de cierre al final de mes, o algo así). Un usuario a través de una aplicación quiere invocar el proceso, pero no esperar a que termine.

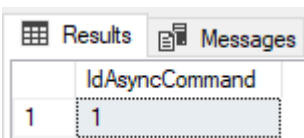
Para efectos de nuestro ejemplo, el proceso de larga duración estará representado por la siguiente sentencia SQL:

```
WAITFOR DELAY '00:00:10' -- Proceso de larga duración.
```

Al ejecutarlo toma diez segundos su ejecución. Pero si en vez de esto encapsulamos la invocación al proceso y usamos:

```
Execute async.ExecuteCmd 'WAITFOR DELAY ''00:00:10'''
```

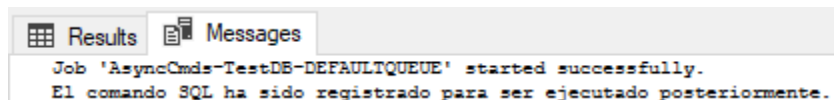
Al ejecutarlo se puede tomar menos de un segundo en contraste con la ejecución del comando anterior, pero, además, el procedimiento **ExecuteCmd** retorna un resultado:



IdAsyncCommand
1

El resultado de la columna **IdAsyncCommand** retorna el número con el cual quedó registrado el comando en el sistema de ejecución asíncrona. Con este número es posible consultar el estado de la ejecución más adelante.

Aquí se separa la invocación de la ejecución, de modo que la invocación solo dura el tiempo necesario para registrar el comando en la cola por defecto que funciona de forma serial. Pero, ¿Qué ocurre tras bambalinas? ¿Cómo se ejecuta finalmente el comando enviado a la cola? Una pista está en la pestaña de mensajes que muestra lo siguiente:



```
Job 'AsyncCmds-TestDB-DEFAULTQUEUE' started successfully.  
El comando SQL ha sido registrado para ser ejecutado posteriormente.
```

² Cuando se ha hecho referencia a un comando se está hablando específicamente del patrón de diseño Command (puede consultarse en https://en.wikipedia.org/wiki/Command_pattern). Aunque, en este sistema puede enviarse una sentencia SQL que retorne registros (como un SELECT), los resultados son ignorados por el sistema.

Antes de la confirmación del registro del comando, aparece una línea notificando el inicio de un Job de SQL Server Agent llamado: **AsyncCmds-TestDB-DEFAULTQUEUE** que representa el agente que atiende la cola por defecto para la base de datos **TestDB**. Dicha notificación solo aparece cuando el agente por algún motivo no está corriendo, pues por lo general, se queda permanentemente en funcionamiento, esperando a que haya elementos en la cola para ser ejecutados.

Para verificar si un comando específico ha sido ejecutado, puede usarse la función **GetCmdExecStatus**, la cual recibe como parámetro el **IdAsyncCommand** que para el ejemplo anterior fue 1, así:

```
SELECT * FROM async.GetCmdExecStatus(1)
```

Obteniendo el siguiente resultado tras su ejecución³:

IdAsyncCommand	Command	RegisteredOn	L...	S...	Cle...	ExecutionSpoolName	StartedOn	FinishedOn	Fa...	ExecutionStatus
1	WAITFOR DELA...	2019-06-14 ...	I...	d...	<lo...	DEFAULTQUEUE	2019-06-14 22:17:16.043	2019-06-14 22:17:26.060	N...	Finished

La columna **ExecutionStatus** permite comprobar el estado en un momento dado junto con otra información de interés.

En caso de que un comando específico produzca errores, estos también pueden ser consultados de la misma manera. En el siguiente ejemplo se introduce un error y además se ilustra la forma como el **IdAsyncCommand** puede ser capturado para luego consultar su estado. Considere el siguiente programa:

```
DECLARE @T TABLE (IdCmd bigint)

INSERT INTO @T
EXECUTE async.ExecuteCmd 'SELECT 1/0'

-- ... WAITFOR DELAY '00:00:01'

SELECT * FROM async.GetCmdExecStatus((SELECT IdCmd FROM @T))
```

El **1/0** debe producir un error de división por cero. La variable **@T** de tipo tabla “recoge” el valor del **IdAsyncCommand** en la columna **IdCmd**, la cual es usada luego para obtener el estado a través de **GetCmdExecStatus**. Si se deja pasar el suficiente tiempo entre la invocación y la consulta (aproximadamente un segundo) podrá verse algo como esto:

IdAsyncCommand	Command	RegisteredOn		ExecutionSpoolName	StartedOn	Finishe...	FailedOn	ExecutionStatus	ErrorMessage
4	SELECT 1/0	2019-06-14 ...		DEFAULTQUEUE	2019-06-14 23:09:54.203	NULL	2019-06-14 23:...	Failed	8134:Divide by zero error encountered. Línea: 1...

En este caso, **ExecutionStatus** muestra el estado de falla, la columna **FailedOn** tiene la fecha y hora del error y la columna **ErrorMessage** la descripción del mismo.

³ Las fechas y horas de este sistema de comandos asincrónicos fueron tratadas usando Tiempo Universal Coordinado (UTC), no mediante el tiempo local. Esto implica que a las fechas y horas hay que adicionarles el desplazamiento correspondiente a la zona horaria donde se consulten los datos.

Un aspecto final a resaltar aquí, es el carácter serial de este tipo de ejecuciones. Lo que quiere decir que, un segundo comando solo se ejecuta cuando finaliza la ejecución del primero. No importa si el primero ha terminado bien o con falla. Esto es importante, porque si el primero falla y el segundo depende de un resultado del primero, tendrá que saber lidiar con esta posible situación anómala, pues la cola no tiene la inteligencia para lidiar con este problema y simplemente continua con la ejecución.

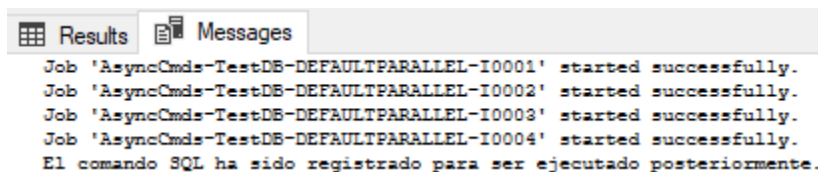
Ejecución en paralelo

En este tipo de ejecución, se usa el mismo procedimiento **ExecuteCmd** para poner comandos en la cola, pero se usa un parámetro adicional llamado **@pIsIndependent**, que cuando es 1 (comando independiente) es atendido por un agente que puede tener más de una instancia para procesar comandos. La connotación de independiente indica que un segundo comando no necesita esperar a que el primero termine para ser ejecutado, sino que puede ser ejecutado antes que el primero termine. En otras palabras, se trata de una cola con múltiples puntos de atención. Por defecto, la cola paralela, llamada **DEFAULTPARALLEL** tiene 4 agentes, pero este valor puede ser modificado usando el procedimiento **SetSpoolConfiguration** que será revisado más adelante.

De forma análoga al caso serial, se muestra el ejemplo registrando un comando en la cola paralela así:

```
Execute async.ExecuteCmd 'WAITFOR DELAY ''00:00:10''', @pIsIndependent = 1
```

En contraste con el caso serial, al revisar los mensajes tras ejecutar la sentencia anterior, puede verse que se inician cuatro instancias del agente paralelo⁴:



```
Results Messages
Job 'AsyncCmds-TestDB-DEFAULTPARALLEL-I0001' started successfully.
Job 'AsyncCmds-TestDB-DEFAULTPARALLEL-I0002' started successfully.
Job 'AsyncCmds-TestDB-DEFAULTPARALLEL-I0003' started successfully.
Job 'AsyncCmds-TestDB-DEFAULTPARALLEL-I0004' started successfully.
El comando SQL ha sido registrado para ser ejecutado posteriormente.
```

La misma funcionalidad demostrada para el caso serial aplica para el paralelo, pero este último, proporciona la posibilidad de consumir la cola a mayor velocidad que en el serial, en tanto que, por premisa, los comandos son independientes entre sí.

Considerando el siguiente ejemplo:

```
Execute async.ExecuteCmd 'WAITFOR DELAY ''00:00:10''', @pIsIndependent = 1
Execute async.ExecuteCmd 'WAITFOR DELAY ''00:00:10''', @pIsIndependent = 1
Execute async.ExecuteCmd 'WAITFOR DELAY ''00:00:10''', @pIsIndependent = 1
Execute async.ExecuteCmd 'WAITFOR DELAY ''00:00:10''', @pIsIndependent = 1
Execute async.ExecuteCmd 'WAITFOR DELAY ''00:00:10''', @pIsIndependent = 1
```

⁴ Estas notificaciones de inicio de los agentes solo aparecen cuando uno o más de ellos no están corriendo, pues por lo general, son Jobs que corren de forma permanente una vez se han iniciado. Estos pueden ser detenidos usando el procedimiento `async.StopAgents`.

```
Execute async.ExecuteCmd 'WAITFOR DELAY ''00:00:10''', @pIsIndependent = 1
```

En este caso, se registran 6 comandos, cada uno dura 10 segundos en la cola paralela por defecto. El registro, tanto en el caso en paralelo como en el serial tomaría aproximadamente el mismo tiempo. Pero la diferencia considerable se ve en la ejecución. En el caso serial tomarían un poco más de un minuto, mientras que en caso paralelo toman 20 segundos puesto que hay cuatro puntos de atención.

Uso de colas personalizadas

Hasta el momento se ha hablado de ejecuciones seriales y paralelas. Internamente, el sistema asincrónico implementa dos colas respectivas por defecto: DEFAULTQUEUE y DEFAULTPARALLEL. En las invocaciones a **ExecuteCmd** la cola usada es determinada por el parámetro **@pIsIndependent**.

Es posible utilizar colas personalizadas adicionales a las dos ya mencionadas. Esto permite implementar múltiples escenarios de negocio y facilita la implementación de pruebas de concurrencia y de estrés sobre código escrito en T-SQL, como scripts o procedimientos almacenados. Una cola personalizada está identificada con un nombre arbitrario que es determinado por el desarrollador y permite establecer una especie de canal que puede ser monitoreado de manera independiente. Es decir, a una de estas colas personalizadas pueden enviarse una multitud de comandos e ir monitoreando su ejecución hasta que la cola se consuma por completo.

Para crear una cola personalizada debe usarse el procedimiento **SetSpoolConfiguration** como se muestra a continuación:

```
EXEC async.SetSpoolConfiguration
    @pExecutionSpoolName = 'Stress Test 1'
    ,@pNumberExecutionAgents = 4
    ,@pDaysToKeepProcessedCommands = 1
    ,@pEnabled = 1
```

Esta sentencia crea una nueva cola llamada **'Stress Test 1'** que representa nuestro canal, al cual se le pueden enviar comandos para ser ejecutados. En este caso la cola ha sido creada con 4 puntos de atención y almacena un día de historia para los comandos procesados. Una vez que la cola ha sido creada se le pueden enviar comandos de la siguiente forma:

```
Execute async.ExecuteCmd 'WAITFOR DELAY ''00:00:03''',
    @pNonDefaultExecutionSpoolName = 'Stress Test 1'
```

Se usa el mismo **ExecuteCmd** con el parámetro adicional **@pNonDefaultExecutionSpoolName** que permite especificar el nombre de la cola específica que se quiere usar. Para monitorear la cola y revisar los comandos pendientes por ejecutar de la cola, puede usarse la siguiente consulta:

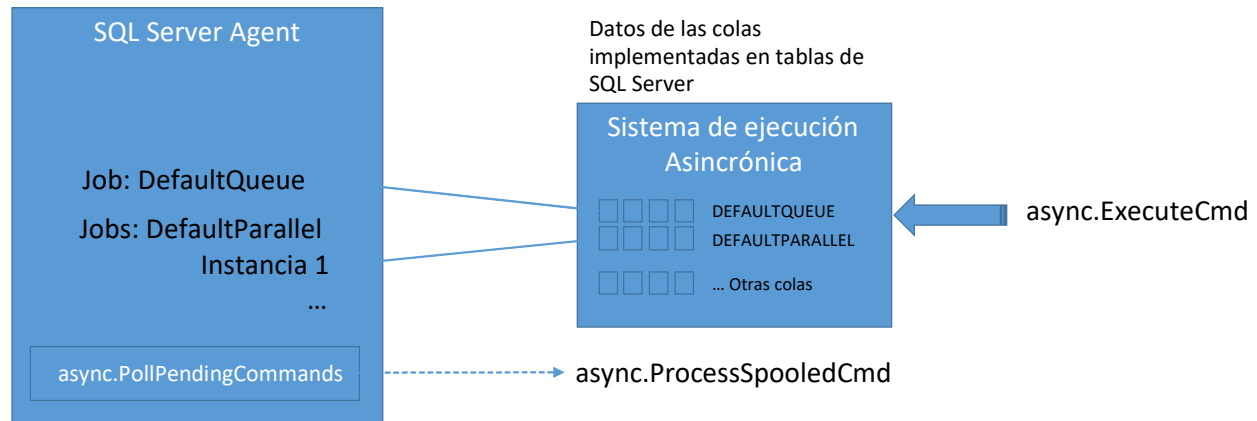
```
SELECT * FROM async.GetPendingCommands('Stress Test 1')
```

Finalmente, una vez que se consuman los comandos de la cola y ya no se requiera más de sus registros históricos ni de sus agentes, puede invocarse el procedimiento **DropSpool** así:

```
EXEC async.DropSpool 'Stress Test 1'
```


Arquitectura del sistema de ejecución asincrónica

Fundamentalmente se trata de una implementación del patrón de integración de aplicaciones conocido como *Fire and Forget*. Los elementos necesarios para el funcionamiento del sistema se ilustran en la siguiente gráfica.



A través del procedimiento almacenado **async.ExecuteCmd** se registran en una cola los comandos a ser ejecutados. Las colas han sido implementadas en tablas estándar de SQL server. A cada una de ellas se le ha denominado **spool**, debido a los modos de acceso usados sobre ellas, donde los comandos inician su procesamiento en estricto orden de llegada (*First-in first-out*).

El procedimiento **ExecuteCmd** se asegura que los agentes necesarios para procesar los comandos enviados estén creados y disponibles. Un **spool** es atendido por uno o más puntos de atención, donde cada punto está representado por un **Job de SQL Server Agent** con un solo paso. Cada paso invoca al procedimiento **async.PollPendingCommands** que corre en el contexto de la base de datos desde donde se invocó **ExecuteCmd** e incluye el nombre del **spool** sobre el cual se está operando. Así pues, un **spool** serial tendrá un único Job y uno con cuatro puntos de atención tendrá cuatro Jobs.

El procedimiento **async.PollPendingCommands** permanece ejecutando hasta que la columna **AgentsStarted** de la tabla **async.SpoolConfiguration** sea igual a 0. Cada 10 segundos verifica si hay comandos pendientes y si así es, invoca al procedimiento almacenado **async.ProcessSpooledCmd**, que es el que finalmente termina ejecutando el próximo comando del **spool**.

Tablas

Son dos las tablas en las que se soporta el sistema: **async.SpoolConfiguration** y **async.CommandsHistory**. La primera contiene el nombre del **spool**, el número de puntos de atención, el número de días en que se conserva la historia después de que termina la ejecución de un comando y si el **spool** está habilitado o no. En la tabla **CommandsHistory** se almacenan los comandos en el orden en que van llegando y cuando se ejecutan, se registra el estado de la ejecución, así como la fecha y hora de terminación, sea esta exitosa o no.

Por defecto, la tabla SpoolConfiguration contiene la configuración para los dos *spools* por defecto:

ExecutionSpoolName	InstanceNameTemplate	NumberExecutionAgents	PendingCommands	DaysToKeepProcessedCommands	AgentsStarted	Enabled
DEFAULTPARALLEL	AsyncCmds-<<Database>>-DEFAULTPARALLEL-I<<Instan...	4	0	1	1	1
DEFAULTQUEUE	AsyncCmds-<<Database>>-DEFAULTQUEUE	1	0	1	1	1

Procedimientos almacenados

Procedimiento async.ExecuteCmd

Parámetros:

- @pCmd: Comando a ejecutar de forma asincrónica.
- Opcional: @pIsIndependent. 0 por defecto.
- Opcional: @pNonDefaultExecutionSpoolName. Nombre del *spool* a usar.

Encola un comando asincrónico especificado en el parámetro @pCmd.

Por defecto, el parámetro @pIsIndependent es 0, ocasionando que los comandos se procesen en el *spool* serial (DEFAULTQUEUE). Si por el contrario se especifica que el comando es independiente, se direcciona al *spool* paralelo (DEFAULTPARALLEL).

Si se especifica un @pNonDefaultExecutionSpoolName, el comando es atendido por el *spool* especificado en dicho parámetro siempre que haya sido creado anteriormente mediante el procedimiento almacenado async.SetSpoolConfiguration.

En cualquier caso, el comando es registrado si el *spool* correspondiente está habilitado y si los agentes pudieron ser iniciados, de lo contrario se produce una excepción.

Procedimiento async.PollPendingCommands

Parámetro opcional: @pExecutionSpoolName: nombre del *spool*. Por defecto asume DEFAULTQUEUE.

Este procedimiento está diseñado para ser ejecutado desde un Job de SQL Server Agent, aunque puede ser ejecutado desde otra interfaz. Se encarga de monitorear el *spool* especificado en el parámetro @pExecutionSpoolName buscando comandos pendientes cada 10 segundos. Cuando encuentra uno o más comandos a ser ejecutados en dicho *spool*, los va consumiendo mediante la invocación a async.ProcessSpooledCmd.

Procedimiento async.ProcessSpooledCmd

Parámetro opcional: @pExecutionSpoolName: nombre del *spool*. Por defecto asume DEFAULTQUEUE.

Este es el procedimiento que realmente ejecuta los comandos enviados al *spool* especificado en el parámetro @pExecutionSpoolName. Dado que el sistema está diseñado para ejecutar comandos que no retornan conjuntos de resultados, si acaso alguno de ellos lo hace, son ignorados por ProcessSpooledCmd.

En caso de que se produzca una excepción no controlada dentro de la ejecución del comando, el mensaje de error queda registrado en la tabla CommandsHistory en la columna ErrorMessage y se actualiza la columna FailedOn con la fecha y la hora de la ocurrencia del error.

Procedimiento `async.SetSpoolConfiguration`

Parámetros:

- @pExecutionSpoolName: nombre del *spool*.
- Opcional: @pNumberExecutionAgents: indica el número de agentes que atienden el *spool*. Si no se especifica usa el número de procesadores lógicos disponibles para la instancia de SQL Server.
- Opcional: @pDaysToKeepProcessedCommands: indica el número de días a conservar la historia después que termina la ejecución de un comando en este *spool*. Por defecto asume 1 día.
- Opcional: @pEnabled: indica si el *spool* está habilitado. Por defecto asume que sí.

Permite crear o modificar la configuración de un *spool*. Si el *spool* no existe lo crea con los parámetros de configuración especificados y si existe, actualiza la configuración.

Es posible que cambie el número de agentes que atienden el *spool*. Este cambio solo se verá reflejado con la siguiente llamada a ExecuteCmd o al procedimiento StartAgents.

Procedimiento `async.DropSpool`

Parámetro opcional: @pExecutionSpoolName: nombre del *spool*. Por defecto asume DEFAULTQUEUE.

Borra un *spool* personalizado, no es posible borrar un *spool* por defecto ni uno que tenga comandos pendientes por ser ejecutados.

Al borrar el *spool* se apagan y se eliminan los agentes que lo atienden y se borran todos los registros correspondientes de las tablas CommandsHistory y SpoolConfiguration.

Procedimiento `async.StartAgents`

Parámetro opcional: @pExecutionSpoolName: nombre del *spool*. Por defecto asume DEFAULTQUEUE.

Inicia los agentes que atienden el *spool* especificado en el parámetro, de acuerdo con la configuración especificada para el mismo. Dado que estos agentes se basan en SQL Server Agent, el procedimiento revisa que este esté disponible y corriendo para la instancia correspondiente de SQL Server. Por lo general no es necesario llamar explícitamente a este procedimiento, ya que este es invocado por ExecuteCmd.

Procedimiento `async.StopAgents`

Parámetro opcional: `@pExecutionSpoolName`: nombre del *spool*. Por defecto asume `DEFAULTQUEUE`.

Detiene la ejecución de los agentes que atienden el *spool* especificado en el parámetro poniendo la columna `async.SpoolConfiguration.AgentsStarted` en cero.

Si el *spool* tiene comandos pendientes por ejecutar, los ejecuta y para cuando todos los comandos sean consumidos.

Funciones

Hay una serie de funciones que retornan tablas (TVF) y otra que retorna valores escalares.

TVF `async.GetCmdExecStatus`

Parámetro: `@pIdAsyncCommand`: identificador de un comando registrado.

Obtiene el registro correspondiente a un comando registrado en `CommandsHistory` a partir del identificador del comando especificado en el parámetro.

TVF `async.GetExecutedCommands`

Parámetro: `@pExecutionSpoolName`: nombre del *spool*.

Obtiene los comandos ejecutados para el *spool* especificado en el parámetro a partir de la tabla `CommandsHistory`.

TVF `async.GetPendingCommands`

Parámetro: `@pExecutionSpoolName`: nombre del *spool*.

Obtiene los comandos pendientes por ejecución para el *spool* especificado en el parámetro a partir de la tabla `CommandsHistory`.

Función escalar `async.NumOfAgentsRunning`

Parámetro: `@pExecutionSpoolName`: nombre del *spool*.

Obtiene el número de agentes de SQL Server Agent (puntos de atención) que están siendo ejecutados en un momento dado para el *spool* especificado en el parámetro.

Función escalar `async.NumOfPendingCommands`

Parámetro: `@pExecutionSpoolName`: nombre del *spool*.

Obtiene el número de comandos pendientes en un momento dado para el *spool* especificado en el parámetro.