How to Use SQL Server's Extended Events and Notifications to Proactively Resolve Performance Issues

Written By Jason Strate

QUEST SOFTWARE®
Simplicity At Work™

Contents

Abstract

SQL Server comes with a wide array of tools for monitoring your environment. There are logs and traces that provide information when errors occur, but these are often used passively to react to events that have already occurred. As DBAs, we need to monitor our environments proactively and create solutions as issues arise. In this white paper, we will look at a couple technologies – event notifications and extended events – that can help you achieve these goals. With these two features, we'll look at the error log and deadlocks, and demonstrate how you can get relevant information delivered as it occurs. We'll also look at ways that run-time errors can be captured and used to help reduce the amount of time required to investigate issues.

Introduction

One of the core duties for all DBAs is to monitor the environments for which they are responsible. This includes tasks that range from designing and deploying databases to managing database backups. We must also monitor performance and troubleshoot problems as they arise. These job functions require us to obtain timely information on issues that we deal with, and that information must be detailed enough so it can be acted upon to resolve issues effectively and efficiently.

Throughout the last few years, the growth in the number of databases that SQL Server DBAs must manage has outpaced the increase in DBAs to manage those databases. We must begin leveraging what we have available to manage the hundreds and thousands of databases we have now, instead of the dozens of databases that we worked with a few years back. To succeed, we must learn to do more with less and improve our capacity to manage and monitor our SQL Server environments.

There are countless ways to accomplish the monitoring that is required. Many of these ways, such as SQL Profiler and Performance Monitor, can provide a lot of general information when problems arise, but they are often unable to provide the timely and detailed information that is necessary to immediately begin troubleshooting issues. In this white paper, we will look at two technologies that are already available within SQL Server 2005 and SQL Server 2008: event notifications and extended events. Each of these provides a method to garner timely and detailed information on issues as they arise. We will also look at some common monitoring the DBAs should be doing, and how we can automate and delve deeper into that monitoring. These solutions will provide a better method of leveraging SQL Server to accomplish what's needed to help us stay on top of our SQL Server environments.

Using SQL Server Features

Two SQL Server features will be discussed in this white paper: event notifications and extended events. Both of these features are already a part of SQL Server and can be used in your environment today. With just a little configuration, you can raise the bar on your monitoring and delve deeper into the problems that you face.

An Introduction to Event Notifications

Event notifications is an event-triggering platform that was introduced with SQL Server 2005. It utilizes events based on data-definition data (DDL) statements and SQL Trace events to feed event data into Service Broker tables. Through the use of Service Broker, the data from these events can be processed in whatever manner is needed based on the event. For instance, the event data can be inserted into monitoring tables or sent to DBAs via email.

Event notifications differ from DDL triggers in that they execute asynchronously. Due to this difference, event notifications can be triggered without affecting the transaction that generated the event. This benefit is realized in two ways: First, this removes the risk that DDL-centric event notifications will cause the triggering transactions to roll back. Second, any performance or resource costs associated with the DDL trigger will be offloaded from the triggering transaction.

When looking at SQL Trace events, made available through event notifications, the platform allows the DBA to perform actions on SQL Trace events and process them, instead of just storing the information in a file or table. With SQL Trace and profiler, these events could be collected, but there wasn't a method to process the data. The bulk of the effort was spent building a process to make the SQL Trace data available to monitor. With event notifications, the data is available as part of the feature. The DBA can concentrate on how to handle the information, instead of how to make it available.

Using event notifications to process and handle information from the DDL trigger and SQL Trace events that are exposed allows these events to be monitored in ways that improve the ability to scale the DBA across the SQL Server environment.

An Introduction to Extended Events

Extended events are a lightweight event-handling platform introduced in SQL Server 2008. This platform captures events, such as wait events, user errors, and deadlocks, as they occur. After the events have been triggered, additional information that may be useful in troubleshooting the event can be appended to it before it is sent to a destination, or target, that can be easily queried. Extended events are very similar to what is available through SQL Profiler or SQL Trace. In fact, it is intended as a replacement for both technologies. With the next release of SQL Server, code-named "Denali," SQL Profiler will be deprecated for the SQL Server engine.

A key benefit to extended events is its performance when compared to other monitoring options. With SQL Trace, there is often the concern that SQL Trace sessions can add to a performance issue. Extended events were designed with performance in mind, and tests have shown that the impact of extended events on transactions is about two microseconds per triggered event. Additionally, while events are triggered synchronously, many destinations for extended events are populated asynchronously, further reducing the likelihood that the monitoring will have an impact on performance.

In order to be as flexible as possible, extended events were designed so that a triggering event can be sent to any of the available destinations. This means that any destination, or target, can receive any single event or multiple events without needing special configuration to handle the various events. Also, any group of events can be bundled together in extended event sessions. If the properties of an event change over time, the session information is as valid with the first version of the event collected as it is with the future versions.

Extended events provide a depth of information about events that far exceeds what is available through SQL Trace, SQL Profiler, Performance Monitor, or many of the DMVs within SQL Server. These other tools can provide high-level information on page splits, latches, and wait stats. Only extended events can look at when these events occur and provide information such as the plan handle, T-SQL stack, and much more, as they are triggered within the database.

Extended events provide a high-performance doorway to look into SQL Server to monitor and provide valuable troubleshooting information. Most of this information is available in other tools and features, but not in a manner that performs as well as extended events.

Starter Solutions

In this paper, we'll focus on a few real-world scenarios for using event notifications and extended events. For each of these scenarios, we'll walk through a method for addressing each of them using both event notifications and extended events. The scenarios we will cover include:

- Deadlocks
- Blocking
- Error messages

For each of these scenarios, we'll go through ways that you can apply event notifications and extended events to the monitoring that you need to accomplish.

Deadlocks

Deadlocks are an unfortunate reality in many environments. As DBAs, we are not necessarily responsible for the occurrence of the deadlocks, but we are responsible for detecting and attempting to prevent them from reoccurring. What do we do as DBAs to monitor for and prevent deadlocks?

The common scenario that we face goes a little something like this. You, the DBA, are sitting at your desk minding your own business. In walks the developer or business user, asking about their failed transaction. Unless you are actively looking at transactions on the server, you may not be able to help the developer or business user at all. This time, let's pretend that they brought the error message and it mentions that there was a deadlock. If you have the deadlock trace flags turned on, you could open the error log and look for information on the deadlock. While it is useful to enable the trace flags, which places the deadlocks graph in the error log, it isn't very useful if you aren't actively looking for deadlocks. Also, reading the deadlock graph through the error log is a bit of a pain.

In a variation on this scenario, suppose you're a proactive DBA who actively monitors performance counters. You may be monitoring the perfmon counter deadlocks/sec. so that an alert will be received whenever a deadlock occurs. This is a valid method for monitoring deadlocks because it tells you how many deadlocks are occurring and the scope of the problem. But it does nothing about what transactions have deadlocked. Again, you would need to have the trace flags in place to push deadlocks into the error log; which brings you back to the issues with the first scenario.

Both of these scenarios for monitoring and resolving deadlocks lack one key element: These methods do not provide timely information on the deadlocks. When the deadlocks occur, you aren't prompted to act or don't have the information on hand necessary to act, and must look to other means to provide this.

Resolving Deadlocks with Event Notifications

Fortunately, event notifications provide rich and timely information on deadlocks. The deadlock event can occur, and a process can be built to automatically research and provide information on the deadlock shortly after it occurs.

There are a few steps involved in configuring deadlock monitoring through event notifications:

1. Enable service broker on the database.
2. Create a service broker queue to receive the event notification messages.
3. Create a service broker service to deliver event notification messages.
4. Create a service broker route to route the event notification message to the service broker queue.
5. Create event notification on deadlock event to create messages and send them to the service broker service.

These steps are implemented through the following script:

```
USE master
GO

IF EXISTS (SELECT 1 FROM sys.databases WHERE [name] = 'AdventureWorks' AND is_broker_enabled = 0)
        BEGIN
          ALTER DATABASE AdventureWorks SET ENABLE_BROKER
        END
GO

USE AdventureWorks
GO

CREATE QUEUE DeadlockNotificationQueue;
GO

CREATE SERVICE DeadlockNotificationService
ON QUEUE DeadlockNotificationQueue
([http://schemas.microsoft.com/SQL/Notifications/PostEventNotification]);
GO

CREATE ROUTE DeadlockNotificationRoute
WITH SERVICE_NAME = 'DeadlockNotificationService',
ADDRESS = 'LOCAL' ;
GO

CREATE EVENT NOTIFICATION DeadlockNotification
ON SERVER
WITH FAN_IN
FOR DEADLOCK_GRAPH
TO SERVICE 'DeadlockNotificationService','current database'
GO
```

Once the event notification is created, deadlock event notifications will start to populate the service broker queue. Depending on your system, these messages could be rare or frequent. In any case, you will need to process the items in the service broker queue.

This is where the power of service broker comes into play. With performance counter alerts or trace flags, there isn't any method to transform the information as it is collected into something timely and useful. Through service broker, a stored procedure can be written that responds to deadlock events. Event notifications allow deadlock graphs to be transformed, stored, and sent wherever they need to go.

To illustrate some of the potential, let's say we want to do the following whenever a deadlock is encountered:

1. Store the deadlock graph in a table.
2. Retrieve the cached plans associated with the deadlock in another table.
3. Email the deadlock graph to the DBA team.

All of these can be easily accomplished through a stored procedure on the event that was created. To start, we'll need a couple tables to store the deadlock graphs and the cached plans.

```
USE AdventureWorks
GO

IF NOT EXISTS ( SELECT  *
                FROM    sys.tables
                WHERE   OBJECT_ID = OBJECT_ID('dbo.DeadlocksReports') )
    BEGIN
        CREATE TABLE dbo.DeadlocksReports
        (
         deadlock_id INT IDENTITY(1,1)
        ,deadlock_graph XML
        ,create_date DATETIME
            CONSTRAINT DF_MonitorDeadlocksReports_CreateDate
            DEFAULT (GETDATE())
        ,CONSTRAINT DF_MonitorDeadlocksReports PRIMARY KEY CLUSTERED
            (deadlock_id)
        ) ;
    END
GO

IF NOT EXISTS ( SELECT  *
                FROM    sys.tables
                WHERE   OBJECT_ID = OBJECT_ID('dbo.DeadlocksPlans') )
    BEGIN
        CREATE TABLE dbo.DeadlocksPlans
        (
         deadlock_plan_id INT IDENTITY(1,1)
        ,deadlock_id INT
        ,sql_handle VARBINARY(64)
        ,plan_handle VARBINARY(64)
        ,query_plan XML
        ,CONSTRAINT PK_DeadlocksPlans PRIMARY KEY NONCLUSTERED
            (deadlock_plan_id)
        ) ;

        CREATE CLUSTERED INDEX [CIX_DeadlocksPlans] ON dbo.[DeadlocksPlans] (deadlock_id,
deadlock_plan_id) ;

        ALTER TABLE dbo.[DeadlocksPlans]  WITH CHECK
        ADD CONSTRAINT [FK_DeadlocksPlans_DeadlocksReports] FOREIGN KEY(deadlock_id)
        REFERENCES dbo.[DeadlocksReports] (deadlock_id) ;
    END;
GO
```

Once the tables are created to store the event notification messages, the stored procedure can be created. Since we are working with service broker, the stored procedure will be a bit cumbersome since it is based on a stylized template for retrieving messages from service broker queues.

The stored procedure is going to go through a few steps for each of the deadlock notification messages. These steps are:

1. Retrieve deadlock message from DeadlockNotificationQueue queue.
2. Insert deadlock graph into dbo.DeadlocksReports.
3. Insert compiled plans in plan cache into dbo.DeadlocksPlans.
4. Email deadlock graph to DBAs.

See the code below for accomplishing these steps. One item to note: at the end of the stored procedure is the addition of a signature to the stored procedure. This signature is required to allow SQL Server to have the proper permissions to use sp_send_dbmail in the stored procedure. More about signing and the certificate can be found in the appendix of this white paper.

```
USE AdventureWorks
GO

IF EXISTS(SELECT * FROM sys.procedures WHERE object_id =
OBJECT_ID('[dbo].[ProcessDeadlocksReports]'))
    DROP PROCEDURE dbo.[ProcessDeadlocksReports];
GO

CREATE PROCEDURE [dbo].[ProcessDeadlocksReports]
WITH EXECUTE AS OWNER
AS
SET NOCOUNT ON

DECLARE @message_body XML
   ,@message_type INT
   ,@dialog UNIQUEIDENTIFIER
   ,@subject VARCHAR(MAX)
   ,@body VARCHAR(MAX)

DECLARE @deadlock TABLE
    (
     deadlock_id INT
    ,deadlock_graph XML
    )
DECLARE @handles TABLE
    (
     deadlock_id INT
    ,plan_handle VARBINARY(64)
    ,sql_handle VARBINARY(64)
    )

WHILE (1 = 1)
    BEGIN
        BEGIN TRANSACTION

-- Receive the next available message from the queue
        WAITFOR (
                RECEIVE TOP(1) -- just handle one message at a time
                        @message_type=message_type_id, --the type of message received
                        @message_body=CAST(message_body AS XML),      -- the message contents
```

```
                        @dialog = conversation_handle    -- the identifier of the dialog this
message was received on
                        FROM dbo.DeadlockNotificationQueue
       ), TIMEOUT 1000  -- if the queue is empty for one second, give UPDATE and go away

-- If we didn't get anything, bail out
        IF (@@ROWCOUNT = 0)
            BEGIN
                ROLLBACK TRANSACTION
                BREAK
            END

        SELECT  @message_body

        INSERT  INTO dbo.DeadlocksReports
                (
                 deadlock_graph
                )
        OUTPUT  INSERTED.deadlock_id
               ,INSERTED.deadlock_graph
                INTO @deadlock
                SELECT  @message_body.query('(/EVENT_INSTANCE/TextData/deadlock-list/.)[1]') ;

        INSERT  INTO @handles
                (
                 deadlock_id
                ,plan_handle
                ,sql_handle
                )
                SELECT  d.deadlock_id
                       ,qs.plan_handle
                       ,qs.sql_handle
                FROM    @deadlock d
                        CROSS APPLY deadlock_graph.nodes('//frame') x (frame)
                        INNER JOIN sys.dm_exec_query_stats qs
                            ON qs.[sql_handle] =
frame.value('xs:hexBinary(substring((@sqlhandle)[1],3))', 'varbinary(max)')


        INSERT  INTO dbo.DeadlocksPlans
                (
                 deadlock_id
                ,plan_handle
                ,sql_handle
                ,query_plan
                )
                SELECT  c.deadlock_id
                       ,c.plan_handle
                       ,c.sql_handle
                       ,qp.query_plan
                FROM    @handles c
                        CROSS APPLY sys.dm_exec_query_plan(c.plan_handle) qp

        SET @subject = @@SERVERNAME + ' - Deadlock Notification'
        SELECT  @body = CONVERT(NVARCHAR(MAX),deadlock_graph) + CHAR(13)
                + CHAR(13)
                + '!! Automatically generated by [dbo].[ProcessDeadlocksReports] !!'
        FROM    @deadlock

        EXEC msdb.dbo.sp_send_dbmail @recipients = 'production_dba@your_company' -- your email
            ,@subject = @subject -- Subject defined above
            ,@body = @body ; -- Body defined above

        DELETE  FROM @deadlock
        DELETE  FROM @handles
```

```
--  Commit the transaction.  At any point before this, we could roll
--  back - the received message would be back on the queue and the response
--  wouldn't be sent.
        COMMIT TRANSACTION
    END;
GO

ADD SIGNATURE TO OBJECT::[ProcessDeadlocksReports]
BY CERTIFICATE [DBMailCertificate]
WITH PASSWORD = 'P@$$w0rd';
GO
```

One piece to mention in the stored procedure above is the inclusion of a signature on the stored procedure. This provides the stored procedure permission to execute sp_send_mail without enabling the TRUSTWORTHY database option. Every time the stored procedure is changed, the stored procedure will need to be signed. Additional information on signing stored procedures is found in the appendix.

At this point, the event notification queues and the stored procedure for processing the deadlock messages have been created. The last step is to assign the stored procedure to the service broker queue for the deadlock events. This is completed by altering the queue to assign the stored procedure to the queue. By doing this, when deadlock event messages are received, the stored procedure will get called to process them until there are no more messages.

```
USE AdventureWorks
GO

ALTER QUEUE DeadlockNotificationQueue
WITH
 ACTIVATION
        (STATUS=ON,
         PROCEDURE_NAME = [ProcessDeadlocksReports],
         MAX_QUEUE_READERS = 1,

         EXECUTE AS OWNER);
```

Enabling the capture of deadlocks with event notifications takes a bit more to set up than enabling a trace flag. But in the end, doing this will allow you to be alerted to deadlock events as they happen, providing the opportunity to troubleshoot issues immediately, rather than when end users start to notice them.

Resolving Deadlocks with Extended Events

Another way to monitor deadlocks is through extended events. In SQL Server 2008, an extended event session named system_health is automatically created and started. One of the events that this session captures is the deadlock graph.

Without any setup or configuration, deadlocks in the system_health session can be accessed through the use of the query below:

```
;WITH SystemHealth
AS (
    SELECT CAST(target_data AS xml) AS SessionXML
    FROM sys.dm_xe_session_targets st
        INNER JOIN sys.dm_xe_sessions s ON s.address = st.event_session_address
    WHERE name = 'system_health'
)
SELECT Deadlock.value('@timestamp', 'datetime') AS DeadlockDateTime
    ,CAST(Deadlock.value('(data/value)[1]', 'varchar(max)') as xml) as DeadlockGraph
FROM SystemHealth s
    CROSS APPLY SessionXML.nodes ('//RingBufferTarget/event') AS t (Deadlock)
WHERE Deadlock.value('@name', 'nvarchar(128)') = 'xml_deadlock_report';
```

Running this query returns the deadlock graph and the time in which the deadlock occurred. These can be used to start digging into the causes of the deadlock. In situations where you haven't set up deadlock monitoring, information necessary to troubleshoot recent deadlocks is already being stored.

	DeadlockDateTime	DeadlockGraph
1	2011-04-20 03:27:16.657	<deadlock-list><deadlock><victim-list><victimPro...
2	2011-04-20 04:09:19.143	<deadlock-list><deadlock><victim-list><victimPro...
3	2011-04-20 04:09:24.143	<deadlock-list><deadlock><victim-list><victimPro...
4	2011-04-20 04:09:30.393	<deadlock-list><deadlock><victim-list><victimPro...
5	2011-04-20 04:09:35.393	<deadlock-list><deadlock><victim-list><victimPro...

Figure 1. Extended Events Results from System Health

If your SQL Server environment contains SQL Server with versions lower than SQL Server 2008 Service Pack 1 Cumulative Update Package 6 or SQL Server 2008 R2 Cumulative Update Package 1, you will encounter a bug converting the deadlock graphs above into XML. The issue is detailed in the Knowledge Base article 978629. To resolve the issue, use the query below instead:

```
;WITH SystemHealth
AS (
    SELECT CAST(target_data AS xml) AS SessionXML
    FROM sys.dm_xe_session_targets st
        INNER JOIN sys.dm_xe_sessions s ON s.address = st.event_session_address
    WHERE name = 'system_health'
)
SELECT Deadlock.value('@timestamp', 'datetime') AS DeadlockDateTime
    ,CAST(REPLACE(REPLACE(Deadlock.value('(data/value)[1]', 'varchar(max)'),
        '<victim-list>', '<deadlock><victim-list>'),
        '<process-list>','</victim-list><process-list>')
              as xml) as DeadlockGraph
FROM SystemHealth s
    CROSS APPLY SessionXML.nodes ('//RingBufferTarget/event') AS t (Deadlock)
WHERE Deadlock.value('@name', 'nvarchar(128)') = 'xml_deadlock_report';
```

This solution can be found in Jonathan Kehayias' article Retrieving Deadlock Graphs with SQL Server 2008 Extended Events. Also, more information on parsing the deadlock graph is available from Michael Zilberstein in he post Parsing Extended Events xml_deadlock_report.

Blocked Process

Along the same vein as deadlocks are blocked processes. Blocked processes occur to some degree in many SQL Server environments. A blocking event is when one session locks a resource and makes it unavailable to other sessions needing the same resource, thus blocking the other session from proceeding. It can also occur more frequently as transaction volume increases.

The key is for DBAs to understand the types of blocking that are occurring, and to know when blocking has exceeded the environment's acceptable threshold. For some environments, such as where orders are being taken over the Internet, it may be critical to get in and take a look at any blocking that occurs for more than two seconds. In reporting environments, where users are churning out several large reports, the bar can be much higher. In that case, blocking may matter only after a couple of minutes. There is no single answer that is perfect for all organizations. The key is to determine what is necessary for your applications and environment to make certain that blocking isn't leading to performance issues and application timeouts.

If you don't already know, find out the threshold on blocking for your environment. Once that is determined, you need to actively monitor blocked processes to know when they are becoming an issue, before it becomes apparent to the end user.

To find the blocked process threshold, run the following:

```
sp_configure 'blocked process threshold';
GO
```

To reconfigure the blocked process threshold, run the following:

```
--Change the blocked process threshold to 2 seconds
sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
sp_configure 'blocked process threshold', 2;
GO
RECONFIGURE;
GO
```

Monitoring Blocked Processes with Event Notifications

Event notifications can provide assistance with monitoring blocked processes. Configuring blocked process monitoring is very similar to setting up deadlock monitoring. The chief differences are the event that is sent to the service broker queue and the procedure used to process the event notification message.

The blocked process event notification can be set up through the following script:

```
USE AdventureWorks
GO

CREATE QUEUE BlockedProcessReportQueue;
GO

CREATE SERVICE BlockedProcessReportService
ON QUEUE BlockedProcessReportQueue
([http://schemas.microsoft.com/SQL/Notifications/PostEventNotification]);
GO

CREATE ROUTE BlockedProcessReportRoute
WITH SERVICE_NAME = 'BlockedProcessReportService',
ADDRESS = 'LOCAL';
GO

CREATE EVENT NOTIFICATION BlockedProcessReport
ON SERVER
WITH FAN_IN
FOR BLOCKED_PROCESS_REPORT
TO SERVICE 'BlockedProcessReportService','current database';
GO
```

As with deadlock monitoring, once the service broker queue is created, blocked process information will begin to be collected. There are a few more steps that need to happen before we will be able to start collecting blocked process reports. Namely, we need to build a stored procedure to process the blocked process reports and configure the blocked process threshold within SQL Server to enable the blocked process report.

For the processing of blocked process reports, the procedure will be similar to the deadlock monitoring process. We'll want a table blocked process reports, for historical purposes.

```
USE AdventureWorks
GO

IF OBJECT_ID('[dbo].[BlockedProcessReports]') IS NULL
        CREATE TABLE [dbo].[BlockedProcessReports]
                (
                blocked_process_id int IDENTITY(1,1) PRIMARY KEY,
                database_name sysname,
                post_time datetime,
                blocked_process_report xml
                );
GO
```

Next, we want to build a stored procedure that can do the following actions to each blocked process report message that is received through event notifications:

1. Store the blocked process report in a table.
2. Email the blocked process report to the DBA team.

The stored procedure will have a similar layout to the one for the deadlock monitoring. This is mainly because stored procedures for service broker follow a stylized format.

```
USE AdventureWorks
GO

CREATE PROCEDURE [dbo].[ProcessBlockProcessReports]
WITH EXECUTE AS OWNER

AS

SET NOCOUNT ON

DECLARE @message_body XML
   ,@message_type INT
   ,@dialog UNIQUEIDENTIFIER
   ,@subject VARCHAR(MAX)
   ,@body VARCHAR(MAX)

WHILE (1 = 1)
    BEGIN
        BEGIN
            BEGIN TRANSACTION

-- Receive the next available message from the queue

            WAITFOR (
               RECEIVE TOP(1) -- just handle one message at a time
                       @message_type=message_type_id, --the type of message received
                       @message_body=CAST(message_body AS XML),      -- the message contents
                       @dialog = conversation_handle    -- the identifier of the dialog this
message was received on
                       FROM dbo.BlockedProcessReportQueue
       ), TIMEOUT 1000  -- if the queue is empty for one second, give UPDATE and go away

-- If we didn't get anything, bail out
            IF (@@ROWCOUNT = 0)
                BEGIN
                    ROLLBACK TRANSACTION
                    BREAK
                END

            INSERT  INTO [dbo].[BlockedProcessReports]
                    (
                     database_name
                    ,post_time
                    ,blocked_process_report
                    )
                    SELECT  DB_NAME(CAST(@message_body AS
XML).value('(/EVENT_INSTANCE/DatabaseID)[1]',
                                                            'int'))
                           ,CAST(@message_body AS XML).value('(/EVENT_INSTANCE/PostTime)[1]',
                                                            'datetime')
                           ,CAST(@message_body AS XML).query('(/EVENT_INSTANCE/TextData/blocked-
process-report/.)[1]')
```

```
            SET @subject = @@SERVERNAME + ' - Deadlock Notification'
            SELECT  @body = CONVERT(NVARCHAR(MAX),CAST(@message_body AS
XML).query('(/EVENT_INSTANCE/TextData/blocked-process-report/.)[1]')) + CHAR(13)
                    + CHAR(13)
                    + '!! Automatically generated by [Monitor].[ProcessBlockProcessReports] !!'

            EXEC msdb.dbo.sp_send_dbmail @recipients = 'production_dba@your_company' -- your
email
                ,@subject = @subject -- Subject defined above
                ,@body = @body ; -- Body defined above
        END

--  Commit the transaction.  At any point before this, we could roll
--  back - the received message would be back on the queue AND the response
--  wouldn't be sent.
        COMMIT TRANSACTION
    END;
GO

ADD SIGNATURE TO OBJECT::[dbo].[ProcessBlockProcessReports]
BY CERTIFICATE [DBMailCertificate]
WITH PASSWORD = 'P@$$w0rd';
GO
```

With this created, the procedure can be added to the blocked process report service broker queue. Adding this can be done with the following SQL statement:

```
USE AdventureWorks
GO

ALTER QUEUE BlockedProcessReportQueue
WITH
 ACTIVATION
    (STATUS=ON,
     PROCEDURE_NAME = [dbo].[ProcessBlockProcessReports],
     MAX_QUEUE_READERS = 1,
     EXECUTE AS OWNER);
GO
```

At this point, the only thing left to configure is the threshold for creating blocked process reports. This is done as a configuration value at the SQL Server instance level. The value provided to the configuration determines how many seconds a blocked process will wait before creating a report. Configuring the blocked process report to be collected after five seconds is done with the statements below:

```
EXEC sp_configure 'show advanced options',1 ;
GO

RECONFIGURE;
GO

EXEC sp_configure 'blocked process threshold',2 ;
GO

RECONFIGURE;
GO
```

At this point, the blocked process event notification is configured. When blocked processes occur in your environment, an email will be received. You have the ability to deal with blocking as it occurs to determine the root cause and mitigate application interference. Instead of developers and users coming to you with questions on why the applications don't seem to be doing anything, you'll go to them with solutions and resolutions in hand.

One key thing to remember is that the report is created every time a query is blocked for the interval that is configured. For instance, if it is set to five seconds then every five seconds that a process is blocked, the blocked process report will be created. This may appear to be overkill but it provides a measure of pressure to enforce dealing with blocked processes. As a result, a query can't be left for hours without drawing attention if the first alert is overlooked.

The stored procedure for the blocked process event notification comes from Jonathan Kehayias' SQL Saturday Session Real Time Problem Identification with Event Notifications.

Monitoring Blocked Processes with Extended Events

Like deadlocks, blocked processes can also be monitored with extended events, though blocked processes aren't already configured like the deadlock monitoring is above. To do this monitoring, you need to create an extended event session.

There are two events that will be used with extended events:

- sqlserver.locks_lock_timeouts: Number of times SQL Server waited on a row lock
- sqlserver.locks_lock_waits: Total number of milliseconds SQL Server waited on a row lock

The extended event session below can be built to capture blocking. This session will capture both events, along with the sql_text and tsql_stack actions.

```
CREATE EVENT SESSION BlockingTransactions ON SERVER
    ADD EVENT sqlserver. locks_lock_timeouts (
    ACTION (sqlserver.sql_text, sqlserver.tsql_stack)
    )
    ,ADD EVENT sqlserver. locks_lock_waits (
    ACTION (sqlserver.sql_text, sqlserver.tsql_stack)
    )
    ADD TARGET package0.ring_buffer
    WITH (MAX_DISPATCH_LATENCY = 30 SECONDS);
GO

ALTER EVENT SESSION BlockingTransactions ON SERVER STATE = START;
GO
```

When blocking occurs, the information can be extracted from the session with the following query:

```
WITH BlockingTransactions
AS (
    SELECT CAST(target_data AS xml) AS SessionXML
    FROM sys.dm_xe_session_targets st
        INNER JOIN sys.dm_xe_sessions s ON s.address = st.event_session_address
    WHERE name = 'BlockingTransactions'
)
SELECT
block.value('@timestamp', 'datetime') AS event_timestamp
,block.value('@name', 'nvarchar(128)') AS event_name
,block.value('(data/value)[1]', 'nvarchar(128)') AS event_count
,block.value('(data/value)[1]', 'nvarchar(128)') AS increment
,mv.map_value AS lock_type
,block.value('(action/value)[1]', 'nvarchar(max)') AS sql_text
,block.value('(action/value)[2]', 'nvarchar(255)') AS tsql_stack
FROM BlockingTransactions b
    CROSS APPLY SessionXML.nodes ('//RingBufferTarget/event') AS t(block)
    INNER JOIN sys.dm_xe_map_values mv ON block.value('(data/value)[3]', 'nvarchar(128)') =
mv.map_key AND name = 'lock_mode'
WHERE block.value('@name', 'nvarchar(128)') = 'locks_lock_waits'
UNION ALL
SELECT
block.value('@timestamp', 'datetime') AS event_timestamp
,block.value('@name', 'nvarchar(128)') AS event_name
```

```
,block.value('(data/value)[1]', 'nvarchar(128)') AS event_count
,NULL
,mv.map_value AS lock_type
,block.value('(action/value)[1]', 'nvarchar(max)') AS sql_text
,block.value('(action/value)[2]', 'nvarchar(255)') AS tsql_stack
FROM BlockingTransactions b
    CROSS APPLY SessionXML.nodes ('//RingBufferTarget/event') AS t(block)
    INNER JOIN sys.dm_xe_map_values mv ON block.value('(data/value)[2]', 'nvarchar(128)') =
mv.map_key AND name = 'lock_mode'
WHERE block.value('@name', 'nvarchar(128)') = 'locks_lock_timeouts';
```

An example output from this would look like the results show below.

	event_timestamp	event_name	event_count	increment	lock_type	sql_text	tsql_stack
1	2011-04-26 05:28:30.347	locks_lock_waits	17987	17987	IU	BEGIN TRAN UPDATE Produ...	<frame level='1' handle='0x02000000...
2	2011-04-26 05:28:27.007	locks_lock_timeouts	2	NULL	IU	BEGIN TRAN UPDATE Produ...	<frame level='1' handle='0x02000000...

In the sample output above, a single lock occurred. The event had approximately 18K milliseconds of waiting. The locking was on an IU or intent to update. Added to that are the actions for sql_text and tsql_stack. The sql_text provides information on the SQL that was submitted in the transaction. Then the tsql_stack provides the stack of calls leading to the statement that encountered the blocking.

This information could probably be collected and reported on in other ways. What makes this different from using event notifications is that it provides a lot of control over the details returned. The extended event session can be filtered to look at only certain databases, servers, users, etc. After a specified number of events, the event collection can be capped. If the actions included are not enough, more details can be added, such as session id, host name, application name, etc.

Another advantage to using extended events is the opportunity to use multiple targets and collect information over time. For example, instead of the ring_buffer target, the sessions could be configured to use the asynchronous_bucketizer or synchronous_bucketizer target. With those targets, the transactions causing blocking can be grouped together and blocking patterns on specific statements can be seen. This flexibility allows for deep and meaningful blocking troubleshooting with minimal effort on the DBA's part.

Monitoring Errors

A key challenge we have as DBAs is finding the time to adequately monitor the SQL Server errors and error logs in our environments. In an ideal world, a DBA would be on top of all errors in an environment by reviewing error logs on a daily basis, making certain that nothing unexpected appears within them. Unfortunately, monitoring error logs doesn't scale well. As a DBA becomes responsible for more and more instances, the number of logs to review increases. It doesn't take many instances before the DBA has to choose between reading error logs or doing any other work.

Another part of this challenge is that reading the error logs the day after issues appeared is the wrong time to address the issue. Whether the information in the error log is about corruption or an account lockout, DBAs need to know about the issues that are happening in real time. Along these same lines, if there isn't time to check all of the error logs on a daily basis, there certainly isn't time to continuously monitor error logs throughout the day looking for new events.

On top of both of these challenges is the amount of clutter that shows up in the error log. We get notices of backups completed, DBCC messages, user errors, applications errors, etc. Some have high-severity urgency and need to be addressed right away. Others are informational and should be "seen but not heard." As we look through the error log, we need to ignore some of the statistics to find the messages of real value – otherwise we risk missing messages that are important or even critical to address.

Monitoring Errors with Event Notifications

Fortunately, event notifications offer a solution to the dilemma of reading through the error log. One of the events provided in event notifications is the ERRORLOG event. With this event notification, messages sent to the error log can also be sent to a service broker queue for further processing.

The error log event notification can be set up through the following script:

```
USE AdventureWorks
GO

CREATE QUEUE ErrorLogNotificationQueue ;
GO

CREATE SERVICE ErrorLogNotificationService
ON QUEUE ErrorLogNotificationQueue
([http://schemas.microsoft.com/SQL/Notifications/PostEventNotification]) ;
GO

CREATE ROUTE ErrorLogNotificationRoute
WITH SERVICE_NAME = 'ErrorLogNotificationService',
ADDRESS = 'LOCAL';
GO

CREATE EVENT NOTIFICATION ErrorLogNotification
ON SERVER
WITH FAN_IN
FOR ERRORLOG
TO SERVICE 'ErrorLogNotificationService','current database';
GO

ALTER EVENT SESSION exErrors ON SERVER STATE = START;
GO
```

As with the other event notifications, we'll need to build a stored procedure to add to the service broker queue to process the error log messages. One downside to this event notification is that all messages sent to the error log will also go to the error log service-broker queue. All of the messages won't be needed; otherwise, the error log event notification would lose its usefulness as an alternative to picking through the error log.

For simplicity, the stored procedure will retain all error log messages with a severity of 19 and higher. It will also retain all error messages that are logged for error 825.

The error log store procedure will have the following steps:

1. Filter error log messages for those of value.
2. Email the error log message to the DBA team.

The stored procedure will look like the following:

```
USE AdventureWorks
GO

IF OBJECT_ID('[dbo].[ProcessErrorLog]') IS NOT NULL
```

```
    DROP PROCEDURE [dbo].[ProcessErrorLog]
GO

CREATE PROCEDURE [dbo].[ProcessErrorLog]
WITH EXECUTE AS OWNER
AS
SET NOCOUNT ON

DECLARE @message_body XML
   ,@message_type INT
   ,@dialog UNIQUEIDENTIFIER
   ,@subject VARCHAR(MAX)
   ,@body VARCHAR(MAX)

WHILE (1 = 1)
    BEGIN
        BEGIN
            BEGIN TRANSACTION

-- Receive the next available message from the queue
            WAITFOR (
               RECEIVE TOP(1) -- just handle one message at a time
                      @message_type=message_type_id, --the type of message received
                      @message_body=CAST(message_body AS XML),      -- the message contents
                      @dialog = conversation_handle    -- the identifier of the dialog this
message was received on
                      FROM dbo.ErrorLogNotificationQueue
       ), TIMEOUT 1000  -- if the queue is empty for one second, give UPDATE and go away

-- If we didn't get anything, bail out
            IF (@@ROWCOUNT = 0)
                BEGIN
                    ROLLBACK TRANSACTION
                    BREAK
                END

            IF (
                (@message_body.value('(/EVENT_INSTANCE/Severity)[1]','int') > 10) -- Error is not
Informational
                OR (@message_body.value('(/EVENT_INSTANCE/Error)[1]','int') = 825) -- Indicates
possible failing drives
                )
                AND (@message_body.value('(/EVENT_INSTANCE/Error)[1]','int') <> 18456) -- Ignore
Logon failures
                AND (@message_body.value('(/EVENT_INSTANCE/Error)[1]','int') <> 18452) -- Ignore
Logon failures
                BEGIN

                    SET @subject = @@SERVERNAME + ' - SQL Server Error Log'

                    SET @body = 'The following ErrorLog was logged in the SQL Server ErrorLog:'
                        + CHAR(10) + '<P /><B>PostTime:</B> ' +
@message_body.value('(/EVENT_INSTANCE/PostTime)[1]',
                                                        'varchar(128)')
                        + '<BR><B>Error:</B> ' +
@message_body.value('(/EVENT_INSTANCE/Error)[1]',
                                                        'varchar(20)')
                        + '<BR><B>Severity:</B> ' +
@message_body.value('(/EVENT_INSTANCE/Severity)[1]',
                                                        'varchar(20)')
                        + '<BR><B>HostName:</B> ' +
@message_body.value('(/EVENT_INSTANCE/HostName)[1]',
                                                        'varchar(128)')
                        + '<BR><B>ApplicationName:</B> ' +
@message_body.value('(/EVENT_INSTANCE/ApplicationName)[1]',
```

```
                                                        'varchar(128)')
                        + '<BR><B>DatabaseName:</B> ' +
@message_body.value('(/EVENT_INSTANCE/DatabaseName)[1]',
                                                        'varchar(128)')
                        + '<BR><B>SessionLoginName:</B> ' +
@message_body.value('(/EVENT_INSTANCE/SessionLoginName)[1]',
                                                        'varchar(128)')
                        + '<BR><B>TextData:</B><BR> ' +
REPLACE(@message_body.value('(/EVENT_INSTANCE/TextData)[1]',
                                                        'varchar(4000)'),
                                                        CHAR(10),'<BR>')
                        + '<P><I>!! Automatically generated by [Monitor].[ProcessErrorLog]
!!</I></P>'

                    EXEC msdb.dbo.sp_send_dbmail @recipients = 'production_dba@your_company' --
your email
                        ,@subject = @subject -- Subject defined above
                        ,@body = @body  -- Body defined above
                        ,@body_format = 'HTML' ;
                END
        END

--  Commit the transaction.  At any point before this, we could roll
--  back - the received message would be back on the queue AND the response
--  wouldn't be sent.
        COMMIT TRANSACTION
    END;
GO

ADD SIGNATURE TO OBJECT::[dbo].[ProcessErrorLog]
BY CERTIFICATE [DBMailCertificate]
WITH PASSWORD = 'P@$$w0rd';
GO
```

With this created, the procedure can be added to the error log service-broker queue. Adding this can be done with the following SQL statement:

```
USE AdventureWorks
GO

ALTER QUEUE ErrorLogNotificationQueue
WITH
 ACTIVATION
    (STATUS=ON,
     PROCEDURE_NAME = [dbo].[ProcessErrorLog],
     MAX_QUEUE_READERS = 1,
     EXECUTE AS OWNER);
GO
```

At this point, SQL Server will begin to send emails when these errors occur in your environment, freeing you from having to regularly review the error logs to detect errors. The email will include basic information on the error to get you started reviewing the issue.

Monitoring Errors with Extended Events

The errors that appear in the error log aren't the only ones you will care about as a DBA. Although we aren't always the ones writing applications in our environments, we do bear the responsibility of knowing when applications fail and why. For these responsibilities, the extended events platform provides the flexibility and depth needed to gather information on errors in SQL Server. In the context of errors, we are talking about errors that users will often see but that don't end up in the error log, such as the error message "could not find store procedure":

```
Msg 2812, Level 16, State 62, Line 2
Could not find stored procedure 'asdfasdf'.
```

While this and similar errors are part of the applications that use SQL Server, some applications lack the error handling necessary to properly communicate the issue to the end-user or developer. Sometimes, when the errors bubble up, the context of the error message is lost along with important details regarding the error.

Extended events can provide a lightweight tracing platform to collect errors as they occur, providing the DBA with a chance to review recent error messages and assist developers with troubleshooting issues. It's also a way to keep a pulse on issues that could become larger in the future.

For this example, we are going to create an extended event session that captures three error messages. The errors are: 208, 2812, and 4121. These translate to: *Invalid object name*, *Could not find stored procedure*, and *Cannot find either column or the user-defined function or aggregate*, respectively. They can be captured through the event sqlserver.error_reported (this is the event triggered any time an error is encountered). Along with this event, the extended event session will capture the following details:

- sqlserver.session_id
- sqlserver.sql_text
- sqlserver.client_app_name
- sqlserver.client_hostname
- sqlserver.database_id
- sqlserver.username

These details will provide the information needed to take an error and track it down to an application to fix the issues that are being encountered. The extended event session for this can be created with the following script:

```
CREATE EVENT SESSION exErrors ON SERVER -- Session Name
ADD EVENT sqlserver.error_reported -- Event we want to capture
(
    ACTION -- What contents to capture
        (
        sqlserver.session_id
        ,sqlserver.sql_text
        ,sqlserver.client_app_name
        ,sqlserver.client_hostname
        ,sqlserver.database_id
        ,sqlserver.username
        )
```

```
    -- Some predicate or filter (here it is object not found error number)
    WHERE (error = 208
    OR error = 2812
    OR error = 4121
    )
)
ADD TARGET package0.ring_buffer
WITH (max_dispatch_latency = 5 seconds); -- The target
GO

ALTER EVENT SESSION exErrors ON SERVER STATE = START
GO
```

When errors have occurred that the event session is capturing, they can be read through a query similar to this:

```
WITH exErrors
AS (
    SELECT CAST(target_data AS xml) AS SessionData
    FROM sys.dm_xe_session_targets st
        INNER JOIN sys.dm_xe_sessions s ON s.address = st.event_session_address
    WHERE name = 'exErrors'
)
SELECT
    error.value('(@timestamp)[1]', 'datetime') as event_timestamp
    ,error.value('(data/value)[5]', 'varchar(max)') as [error_message]
    ,error.value('(data/value)[1]', 'int') as error
    ,error.value('(action/value)[3]', 'nvarchar(255)') AS client_app_name
    ,error.value('(action/value)[4]', 'nvarchar(255)') as client_hostname
    ,DB_NAME(error.value('(action/value)[5]', 'int')) AS database_name
    ,error.value('(action/value)[6]', 'nvarchar(128)') AS username
    ,error.value('(action/value)[2]', 'varchar(max)') as sql_text
    ,error.value('(action/value)[1]', 'int') as session_id
    ,error.value('(data/value)[4]', 'bit') as user_defined
FROM exErrors d
    CROSS APPLY SessionData.nodes ('//RingBufferTarget/event') AS t(error)

WHERE error.value('@name', 'nvarchar(128)') = 'error_reported';
```

This query will provide a result set similar to the following:

	event_timestamp	error_message	error	client_app_name	client_hostname	database_name	username	sql_text	tsql_stack	session_id	user_defined
1	2011-04-28 03:22:02.377	Could not find stored proc...	2812	Microsoft SQL Server ...	SERVER2008	master	SERVER2008\jstrate	EXEC asdfasdf	<frame level='1' handle='0...	56	0
2	2011-04-28 03:22:02.380	Could not find stored proc...	2812	Microsoft SQL Server ...	SERVER2008	master	SERVER2008\jstrate	EXEC asdfasdf	<frame level='1' handle='0...	56	0
3	2011-04-28 03:22:02.440	Invalid object name 'asdf'.	208	Microsoft SQL Server ...	SERVER2008	master	SERVER2008\jstrate	SELECT * FROM a...	<frame level='1' handle='0...	56	0
4	2011-04-28 03:22:02.477	Invalid object name 'asdf'.	208	Microsoft SQL Server ...	SERVER2008	master	SERVER2008\jstrate	SELECT * FROM a...	<frame level='1' handle='0...	56	0
5	2011-04-28 03:22:02.530	Cannot find either column ...	4121	Microsoft SQL Server ...	SERVER2008	master	SERVER2008\jstrate	SELECT dbo.asdfa...	<frame level='1' handle='0...	56	0
6	2011-04-28 03:22:02.567	Cannot find either column ...	4121	Microsoft SQL Server ...	SERVER2008	master	SERVER2008\jstrate	SELECT dbo.asdfa...	<frame level='1' handle='0...	56	0
7	2011-04-28 03:25:50.473	Cannot find either column ...	4121	Microsoft SQL Server ...	SERVER2008	master	SERVER2008\jstrate	WITH MissingObje...	<frame level='1' handle='0...	56	0

From here, a DBA can easily begin tracking down issues and informing developers of where errors are occurring, the frequency in which they are occurring, and what T-SQL statements are causing them to occur. This example only includes a few error messages, but it can easily be expanded to cover many more messages, helping to resolve other error messages you may encounter such as truncation errors.

Conclusion

We've examined ways to address common issues DBAs deal with on a daily basis. Each of these issues presents its own challenges. When issues arise, DBAs need access to timely, detailed information in order to act, rather than information that requires further investigating.

In this white paper, we looked at:

• **Deadlocks**: With event notifications, we are able to capture and send deadlock graphs to the appropriate individuals. In extended events, we reviewed how SQL Server is already capturing deadlocks and storing them in the system_health extended event session.

• **Blocking**: Through event notifications, we monitored blocked processes and were able to send emails when the events were occurring, rather than wait for users to complain about performance. Then with extended events, we looked at how the information for the blocking can be gathered as you would like and captured in a format that is easy to query and act upon.

• **Error Messages**: In the last scenario, we reviewed how event notifications can free the DBA from monitoring error logs and receiving information on issues as they happen, rather than when the DBA happens to review the error log. As for extended events, the example demonstrated how extended events can be used to monitor for user errors that DBAs don't often deal with.

There are a couple restrictions to be aware of before using event notifications and extended events. If you are using SQL Server 2005 or newer, the event notifications feature is available for you to start using today. With extended events, you will need to be on at least SQL Server 2008. Fortunately, though, there are no edition restrictions on either of these technologies.

To resolve the scenarios in this white paper, we looked at ways to apply both event notifications and extended events. We saw that both technologies provide powerful ways in which DBAs can work within their environment. Each technology plays a different role in helping to monitor SQL Server. Event notifications provides method to automate responses to issues and provide immediate feedback. Extended events provides the means to dig deep into SQL Server to get the details you need to resolve any issues. With these methods properly applied, DBAs will be more informed about the state of their environments and better able to act on issues before users have to complain about outages.

Appendix: Database Mail Certificate

In this white paper, I referenced a certificate that's needed for signing stored procedures. This certificate provides those stored procedures with the permissions necessary to use sp_send_mail. The certificate used in the demonstrations was created using the following script:

```
USE AdventureWorks

CREATE CERTIFICATE [DBMailCertificate]
ENCRYPTION BY PASSWORD = 'P@$$w0rd'
WITH SUBJECT = 'Certificate for signing Stored Procedures that utilize DBMail' ;
GO

BACKUP CERTIFICATE [DBMailCertificate]
TO FILE = 'C:\temp\DBMailCertificate.CER' ;
GO

USE master
GO

CREATE CERTIFICATE [DBMailCertificate]
FROM FILE = 'C:\temp\DBMailCertificate.CER' ;
GO

CREATE LOGIN [DBMailLogin]
FROM CERTIFICATE [DBMailCertificate] ;
GO

GRANT AUTHENTICATE SERVER TO [DBMailLogin]
GO

USE [msdb]
GO

CREATE USER [DBMailLogin] FROM LOGIN [DBMailLogin]
GO

EXEC msdb.dbo.sp_addrolemember @rolename='DatabaseMailUserRole',
    @membername='DBMailLogin' ;
GO
```

More information on using signed certifications can be found on MSDN in the article Tutorial: Signing Stored Procedures with a Certificate. Additionally, Jonathan Kehayias discusses using signed certificates to execute sp_send_dbmail in the article Using a Certificate Signed Stored Procedure to Execute sp_send_dbmail.

About the Author

Jason Strate, Digineer Inc., is a database architect and administrator with more than 15 years of experience. He has been a recipient of the Microsoft Most Valuable Professional (MVP) for SQL Server since July 2009.

His experience includes design and implementation of both OLTP and OLAP solutions, as well as assessment and implementation of SQL Server environments for best practices, performance, and high-availability solutions. Jason is a SQL Server MCITP who participated in the development of Microsoft Certification exams for SQL Server 2008.

He is actively involved with the local PASS chapter (SQL Server User Group) and currently serves as Director of Program Development. Over the past year, Jason worked with the current board to organize the PASSMN SQL Summit 2009 in September for the local community.

Jason enjoys helping others in the SQL Server community, and does so by speaking at technical conferences and user group meetings. Most recently, Jason presented at the SSWUG Virtual Conferences, TechFuse, numerous SQL Saturdays, and PASSMN user group meetings. A contributing author for the Microsoft white paper "Empowering Enterprise Solutions with SQL Server 2008 Enterprise Edition," Jason is also an active blogger with a focus on SQL Server and related technologies.

If you have any questions regarding your potential use of this material, contact:

Quest Software World Headquarters
LEGAL Dept
5 Polaris Way
Aliso Viejo, CA 92656
www.quest.com
email: legal@quest.com

Refer to our Web site for regional and international office information.

Updated— January 2012

About Quest Software, Inc.

Quest Software (Nasdaq: QSFT) simplifies and reduces the cost of managing IT for more than 100,000 customers worldwide. Our innovative solutions make solving the toughest IT management problems easier, enabling customers to save time and money across physical, virtual and cloud environments. For more information about Quest solutions for administration and automation, data protection, development and optimization, identity and access management, migration and consolidation, and performance monitoring, go to www.quest.com.

Contacting Quest Software

PHONE 800.306.9329 (United States and Canada)
If you are located outside North America, you can find your local office information on our Web site.

EMAIL sales@quest.com

MAIL Quest Software, Inc.
World Headquarters
5 Polaris Way
Aliso Viejo, CA 92656
USA

Contacting Quest Support

Quest Support is available to customers who have a trial version of a Quest product or who have purchased a commercial version and have a valid maintenance contract.

Quest Support provides around-the-clock coverage with SupportLink, our Web self-service.
Visit SupportLink at https://support.quest.com.

SupportLink gives users of Quest Software products the ability to:

- Search Quest's online Knowledgebase
- Download the latest releases, documentation and patches for Quest products
- Log support cases
- Manage existing support cases

View the Global Support Guide for a detailed explanation of support programs, online services, contact information and policies and procedures.

WPD-SQL-ExtEvtAndNotif-US-SW-01112012