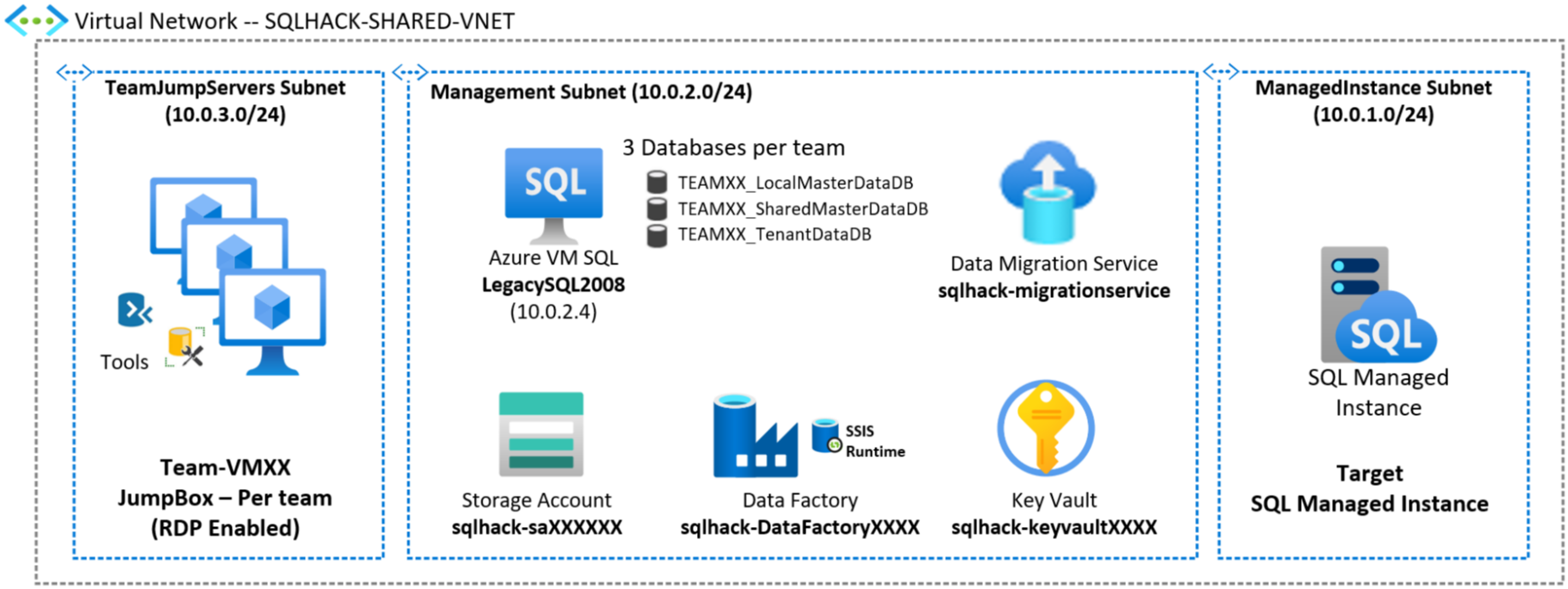
SQL Modernisation Hack
Database Administering and Monitoring Labs Step-by-step

V3.0

Contents

Migration architecture and Azure components



SQLHACK-SHARED-VNET Single Virtual Network containing all workshop resources		
TeamJumpServers Subnet Each team is assigned a Win10 VM that mimics their company desktop	**Management Subnet** Several machines and services are already deployed within a dedicated subnet within the Virtual Network	**ManagedInstance Subnet** The Azure SQL Managed Instance has been deployed into a dedicated Subnet

Lab Overview & Background

In this exercise you will explore how you can monitor and diagnose performance issues with Azure SQL Database Managed Instance using SQL Server Management Studio, DMVs (Dynamic Management Views) and Azure Portal tools.

Background

We have seen that the legacy application is a multi-tenant system with a collection of 3 databases supporting the transactional workload for each customer – these are the 3 databases we have modernised by migrating to the shared Azure SQL Managed Instance.

Apart from the tenant centric transactional databases there are 2 shared databases:

- **2008DW**: A centralised Data Warehouse database that combines data from the various tenant transactional databases and is used for aggregated reporting and analytics.
- **TenantCRM**: A centralised database that is used to manage all customer relationships and order processing.

Recently you have heard complaints from users that the TenantCRM system is running slowly. Recently changes were made to some of the stored procedures in the TenantCRM database and the App dev team believe these are the source of the problem. They've asked for your help to track down the performance issues and its root cause.

Let's follow the steps below to begin to troubleshoot this issue.

The exercises in this lab are conducted against the shared [TenantCRM] database

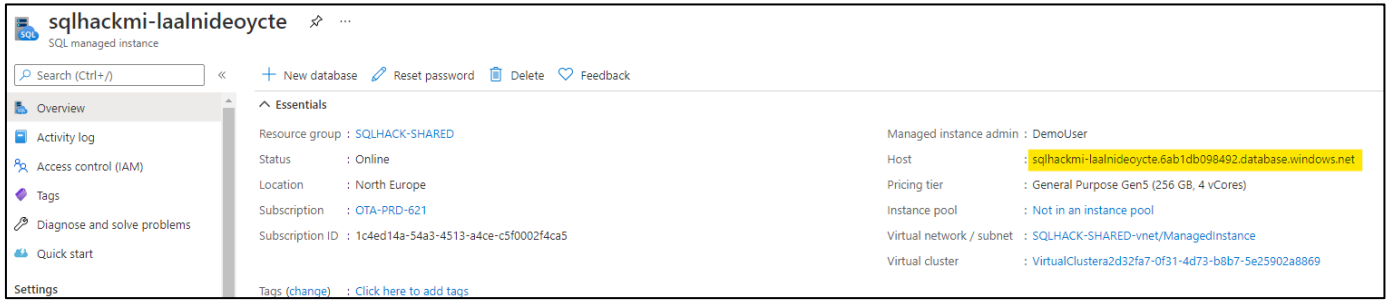
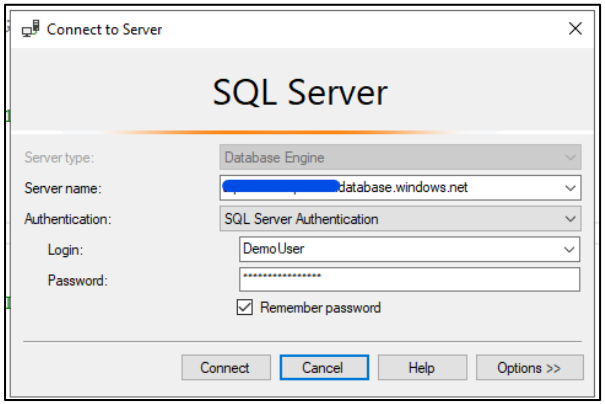
LAB 1: Identifying performance issues, their root causes and fixing the problem

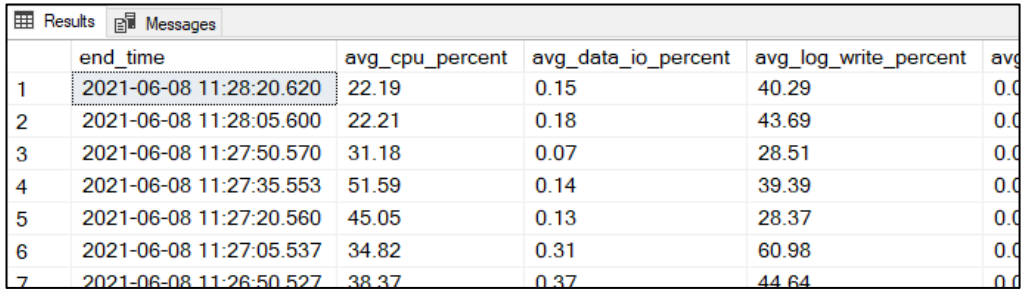
1. Using DMVs and the Query Store to identify performance bottlenecks.

Performance issues in SQL Server can be grouped into 1 of 2 high-level categories: *Running* or *Waiting*.

Running means a query has been allocated CPU resource and is actively processing but is taking a long time to complete. Waiting means a query is not actively being processed by the CPU but is waiting on another resource (such as memory, disk, network, etc) to pass data in so processing can continue.

Finding the bottleneck will help us determine what category issue we have so we can progress to identifying the root cause. Let's see how to find the bottleneck using both SSMS and the Azure portal.

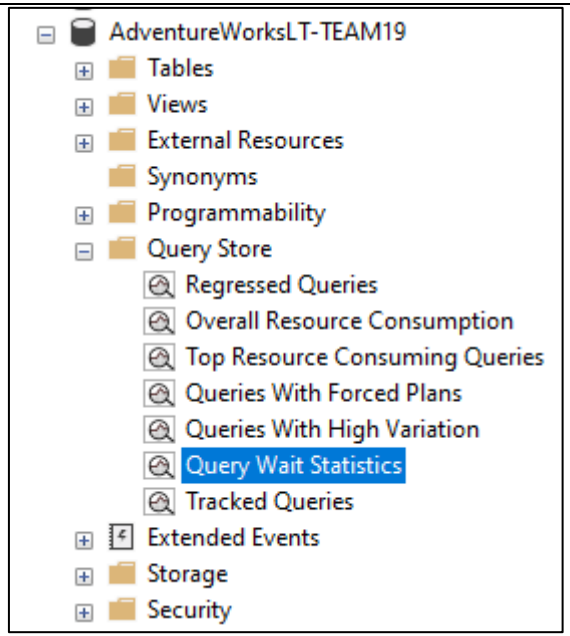
Instructions/Narrative	Screenshot	Notes
On your Win10 Team VM open the Azure Portal and navigate to the shared SQL Managed Instance. Scroll down the Overview screen copy the host name of the Managed Instance and save it somewhere handy.		
On your Team Win10 VM open SQL Server Management Studio and connect to the AdventureWorksLT using these details: **Server name:** <just copied> **Authentication:** SQL Server **Login:** DemoUser **Pword:** Demo@pass1234567		

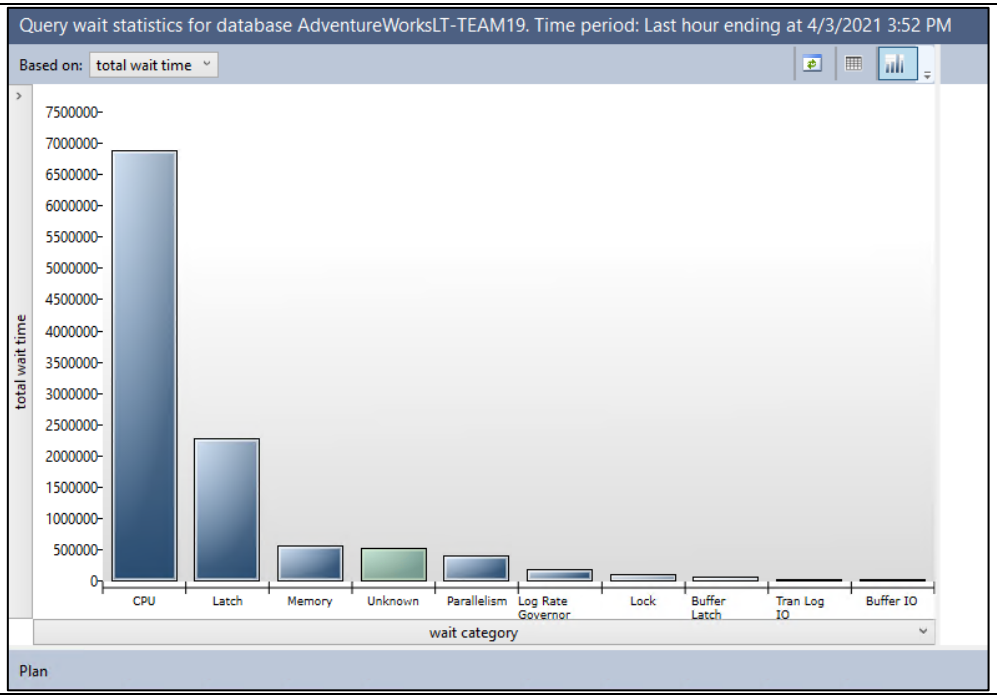
Narrative/Instructions	Screentshot	Notes
In SSMS open SQL script: C:_SQLHACK_\LABS\02-Administering_Monitoring\Part_02_Monitoring_Lab_1.sql **connected to the shared [TenantCRM] database.** Run the **PART 1** query. Look at the results and note the [avg_cpu_percent] value is about 70% indicating that this database is consuming a lot of the hosts CPU resources *(PTO)*		[sys].[dm_db_resource_stats] exposes a number of key database performance metrics including: **avg_cpu_percent** Average compute utilization in percentage of the limit of the service tier. **avg_data_io_percent** Average data I/O utilization as a percentage of the limit of the service tier **avg_log_write_percent** Average transaction log writes (in MBps) as percentage of the service tier limit. **avg_memory_usage_percent** Average memory utilization in percentage of the limit of the service tier.

Results Messages

	end_time	avg_cpu_percent	avg_data_io_percent	avg_log_write_percent	avg
1	2021-06-08 11:28:20.620	22.19	0.15	40.29	0.0
2	2021-06-08 11:28:05.600	22.21	0.18	43.69	0.0
3	2021-06-08 11:27:50.570	31.18	0.07	28.51	0.0
4	2021-06-08 11:27:35.553	51.59	0.14	39.39	0.0
5	2021-06-08 11:27:20.560	45.05	0.13	28.37	0.0
6	2021-06-08 11:27:05.537	34.82	0.31	60.98	0.0
7	2021-06-08 11:26:50.527	38.37	0.37	44.64	0.0

Narrative/Instructions	Screenshot	Notes
Let's analyse the wait stats as you would in an on-premises environment. In SSMS run **PART 2** of Part_04_Monitoring_Lab_1.sql which uses [sys].[dm_db_wait_stats] to report on top wait reasons since the instance was last restarted. View the results. You should see that SOS_SCHEDULER_YIELD is one of the top waits. This wait is reported when there are not enough CPU worker threads to fulfil the query demands. This confirms the results of [sys].[dm_db_resource_stats] DMV in the first query – that this database is putting the CPU under pressure. Now let's see if the Query Store also points to CPU being the issue.	Results Messages |	WaitType | TotalWait_ms | TotalWaitCount | WaitPercentage | |---|---|---|---|---| | 1 | SOS_SCHEDULER_YIELD | 866501442 | 183384051 | 63.64 | | 2 | CXCONSUMER | 184384713 | 1843109 | 13.54 | | 3 | LCK_M_X | 168747070 | 454 | 12.39 | | 4 | LCK_M_IX | 55339875 | 2235 | 4.06 | | 5 | LOGBUFFER | 23412715 | 196674 | 1.72 | | 6 | LCK_M_S | 11354351 | 349416 | 0.83 |	[sys].[dm_db_resource_stats] exposes telemetry captured in the Query Store which you may have used in your on-premise environments. In Azure it is enabled for all Azure SQL Managed Instance and Azure SQL Database databases. The Query Store collects telemetry data every 15 seconds and persists it for about 1hr. *The query used is taken from Glenn Berry's excellent SQL Server performance focused website https://glennsqlperformance.com/* *Glenn's website has a host of SQL version specific queries and is a great resource if you want to really stretch your DMV usage.*

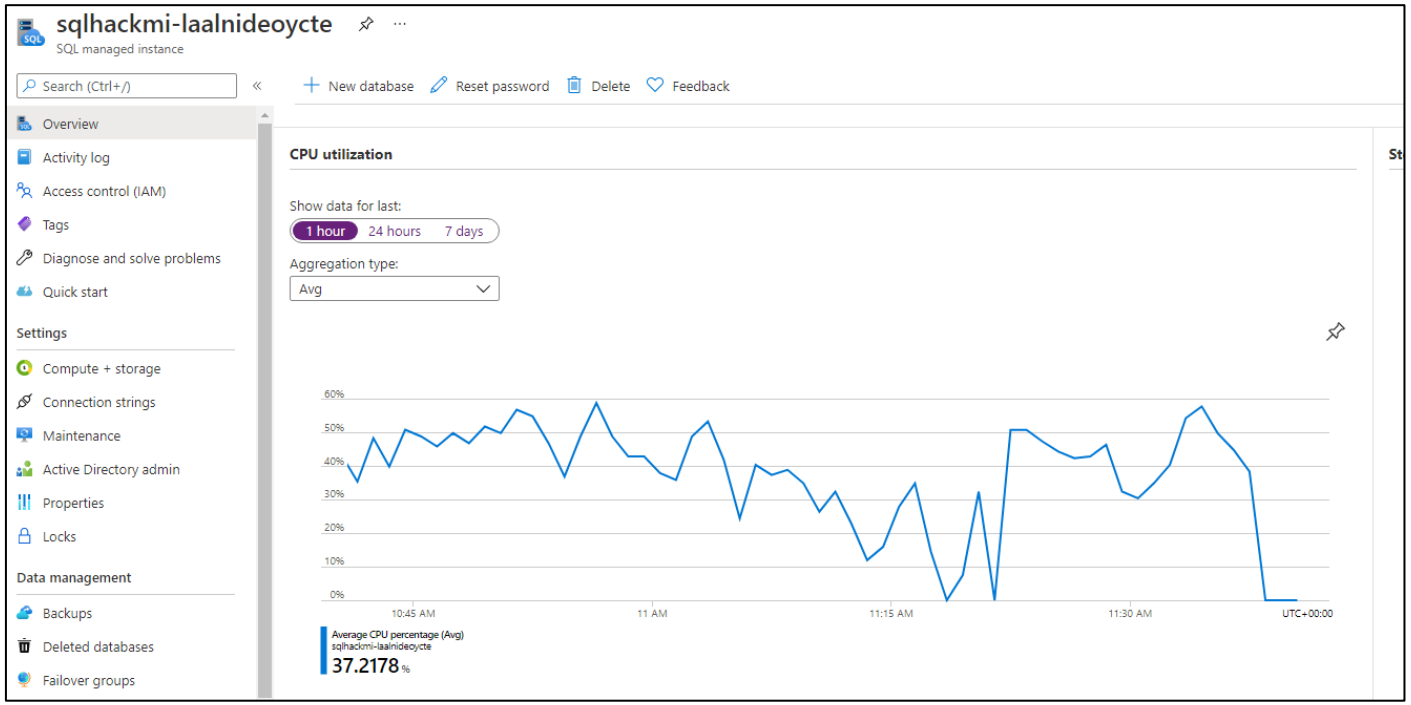
Instructions/Narrative	Screenshot	Notes
In SSMS expand the [TenantCRM] database then expand Query Store. Remember that the Query Store collects telemetry data every 15 seconds and persists it for about 1hr. Double click on **Query Wait Statistics** report to open it. *(PTO)*		

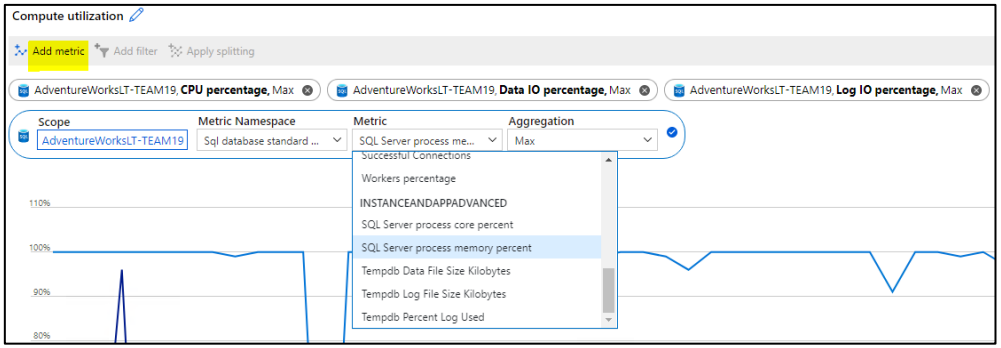
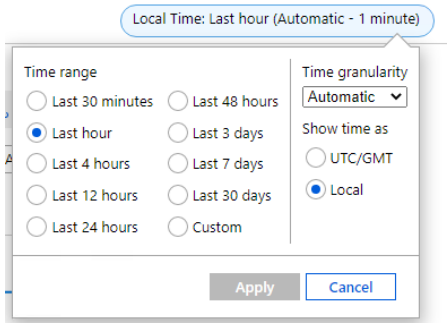
Narrative/Instructions	Screentshot	Notes
You can see that CPU is also reflected here as one of the top waits on the database. Given that SOS_SCHEDULER_YIELD waits are CPU waits the Query Store report confirms CPU is [TenantCRM] database is dominating the CPU resources.		

(PTO)

2. Using the Azure Portal to identify performance bottlenecks

As we are using Azure SQL Managed Instance we can also use the richness of the Azure Portal to help us to troubleshoot performance issues. Let us see how this is done.

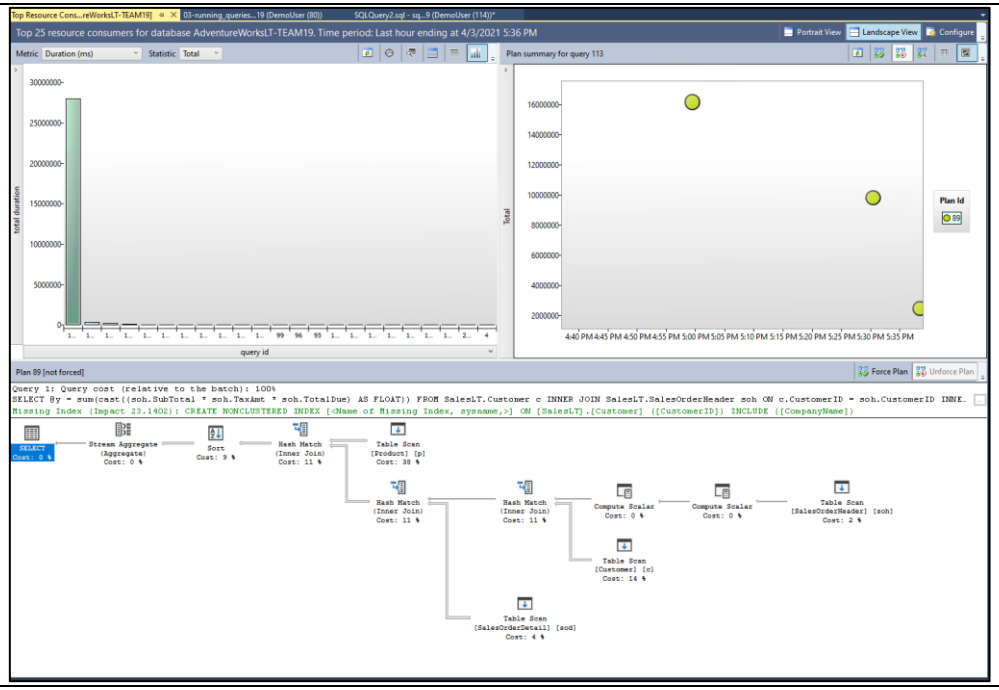
Instructions/Narrative	Screenshot	Notes
Open the Azure Portal (portal.azure.com) and open the shared SQL Managed Instance. Scroll down the **Overview** screen until you see the performance graph showing CPU utilization. Toggle the switches next to view CPU consumption over 1 hour, 24 hours and 7 days. Has the CPU usage pattern changed over time?		

Instructions/Narrative	Screenshot	Notes
Click into the graph to go into Metrics chart screen. Here you have more flexibility over the data being reported and the format of the chart. Let's add IO Request Count the chart. Click **Add Metric** In the Metric drop down select **IO requests count**. Note that because the Average CPU Percentage metric was already on the chart the vertical axis uses percentage which doesn't make much sense when we're looking at the count of IO requests. We will fix this in a later lab.		Notice that from the chart screen you can alter the type of chart (Line, Area, Bar, Scatter to data Grid), set up alerts, and pin the chart to your Azure Portal dashboard.
Click **Local Time** on the top right of the chart to view the data for various time intervals.		

3. Using TSQL and the Query Store to identify the root cause of the CPU strain

Now that we have discovered that the amount of CPU consumed by the [TenantCRM] database is a concern, we need to drill-down and find out what batches are causing the issue. Again we can do this through both SSMS and the Azure portal. Let us begin with Management Studio.

Instructions/Narrative	Screenshot	Notes
In SSMS return to the SQL script: C:_SQLHACK_\LABS\02-Administering_Monitoring\Part_04_Monitoring_Lab_1.sql which should still be connected to your connected to the [TenantCRM] database. Run the **PART 3** query which uses the sys.dm_exec_requests and sys.dm_exec_sql_text DMVs. View the results. Note columns **last_wait_type** and **statement_text** as statement_text reveals the poorly running query. We have already identified that SOS_SCHEDULER_YIELD is the wait type associated with CPU stress and we can now see the query which is causing this bottleneck to occur. Run this query a few times to see if the suspect query and wait type is consistent over multiple runs.	Results Messages session_id | start_time | blocking_session_id | cpu_time_ms | last_wait_type | ObjectName | statement_text 1 | 124 | 2021-06-14 10:27:21.463 | 0 | 3257992 | SOS_SCHEDULER_YIELD | usp_GenerateWorkload_AdventureWorks2017_READ_ONLY | SELECT @y = sum(cast((soh.Sub 2 | 129 | 2021-06-14 09:44:20.367 | 0 | 2964356 | SOS_SCHEDULER_YIELD | usp_GenerateWorkload_AdventureWorks2017_READ_ONLY | SELECT @y = sum(cast((soh.Sub 3 | 131 | 2021-06-14 10:05:59.650 | 0 | 2435650 | SOS_SCHEDULER_YIELD | usp_GenerateWorkload_AdventureWorks2017_READ_ONLY | SELECT @y = sum(cast((soh.Sub 4 | 123 | 2021-06-14 10:11:02.403 | 0 | 2277736 | SOS_SCHEDULER_YIELD | usp_GenerateWorkload_AdventureWorks2017_READ_ONLY | SELECT @y = sum(cast((soh.Sub 5 | 133 | 2021-06-14 11:17:30.083 | 0 | 550302 | SOS_SCHEDULER_YIELD | usp_GenerateWorkload_AdventureWorks2017_READ_ONLY | SELECT @y = sum(cast((soh.Sub 6 | 119 | 2021-06-14 11:25:57.357 | 0 | 318727 | SOS_SCHEDULER_YIELD | usp_GenerateWorkload_AdventureWorks2017_READ_ONLY | SELECT @y = sum(cast((soh.Sub 7 | 118 | 2021-06-14 10:56:41.017 | 0 | 58946 | CXCONSUMER | usp_GenerateWorkload_AdventureWorks2017_READ_ONLY | SELECT pm.[ProductModelID], pr 8 | 112 | 2021-06-11 08:46:13.767 | 0 | 843 | XE_LIVE_TARGET_TVF | NULL | SELECT type, data FROM sys.fn_ 9 | 128 | 2021-06-14 11:17:22.500 | 118 | 124 | LCK_M_X | usp_GenerateWorkload_AdventureWorks2017_READ_ONLY | DROP TABLE IF EXISTS t1 10 | 105 | 2021-06-10 22:00:56.880 | 0 | 97 | XE_LIVE_TARGET_TVF | NULL | SELECT type, data FROM sys.fn_ 11 | 78 | 2021-06-14 11:37:45.603 | 0 | 44 | RESERVED_MEMORY_ALLOCATION_EXT | NULL | SELECT req.session_id , r ***CROSS APPLY and Table-valued Functions*** The PART 3 query joins the [sys].[dm_exec_requests] and [sys].[dm_exec_sql_text] DMVs to obtain long running batches and critically their offending SQL. Note that [dm_exec_sql_text] is actually a table-valued function (TVF) hence the use of the CROSS APPLY as TVFs can't be used in a normal join operation. The CROSS APPLY therefore produces the required inner-join between [dm_exec_requests] and [dm_exec_sql_text]. See APPLY documentation here: Using APPLY | Microsoft Docs and this excellent explanation on MSSQLTips: SQL Server CROSS APPLY and OUTER APPLY (mssqltips.com)	

Instructions/Narrative	Screenshot	Notes
In SSMS expand your database and open the Query Store. Double click **Top Resource Consuming Queries** report to open it. Click on the tallest bar which represents the query with the longest duration. The query text and execution plan will be shown below the charts allowing us to identify the offending query text. Note that it is the same as you saw in the previous DMV output.		The graph on the right shows all the different execution plans for the same query (there will probably only be 1 plan) and their execution duration over time.

We've now seen how the various performance monitoring and diagnosis tools – the Azure Portal, DMVs and the Query Store reports all revealed that the CPU was under pressure. The DMVs and Query Store reports also allowed us to drill-down to root cause and identify the worst offending query. In the real world you would now progress to tune the offending query to reduce the overall load on the Managed Instances CPUs and thereby improve the databases and environment performance.