Monitoring Denodo with Azure Monitor

Revision 20200803

Denodo North America & APAC: 525 University Avenue, Suite 31, Palo Alto, CA 94301. USA
Denodo Iberia & Latino América: Montalbán 5, 28014 Madrid, Spain
Denodo EMEA: 21st Floor, Portland House, Bressenden Place, London SW1E 5RS. UK
Denodo DACH: Karlstraße 10, 80333 München. Germany
www.denodo.com

CONTENTS

1 INTRODUCTION

Azure Monitor is a Microsoft solution for collecting, analyzing, and acting on telemetry from cloud and on-premises environments. It helps understand how applications are performing and proactively identifies issues affecting them and the resources they depend on.

Data can be collected from a variety of sources, one of them is the Custom Log data source that allows collect events from text files. This allows us to monitor Denodo Log files using Azure Monitor.

In this document, we will learn how to configure Azure Monitor to consume Denodo Server and Denodo Monitor Logs. We will also learn how to use Azure Monitor to create alerts and dashboards based on Denodo Log data.

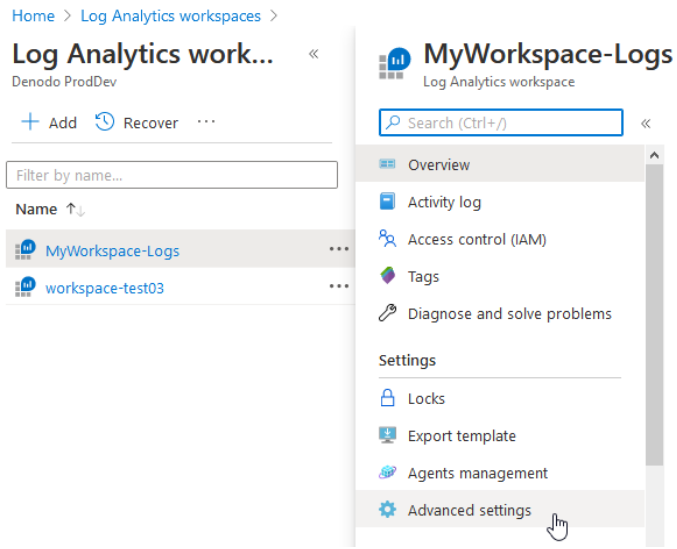
More information about Denodo Server and Denodo Monitor Logs can be found in the Denodo Knowledge Base.

2 INGESTING DENODO LOGS IN AZURE MONITOR

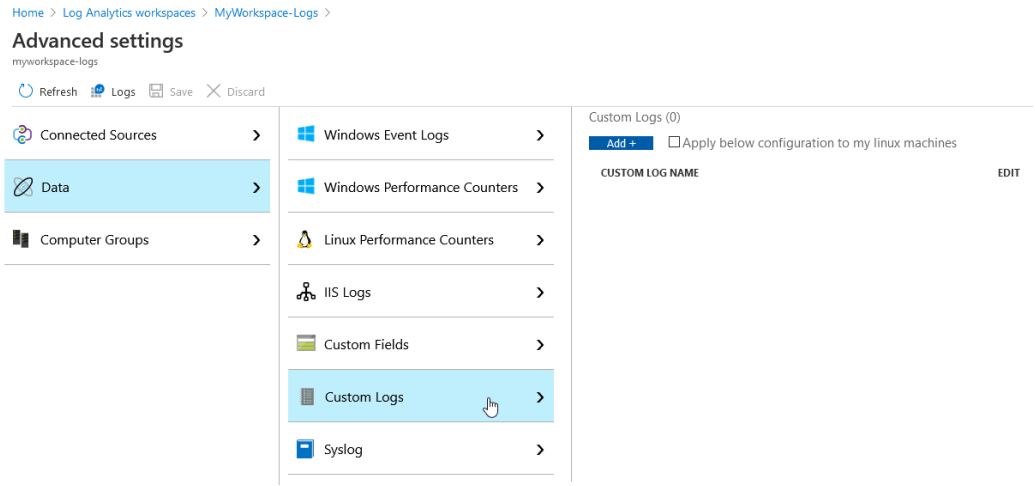
The Azure Log Analytics agent, previously referred to as the Microsoft Monitoring Agent (MMA) or OMS Linux agent, attaches to an Azure Monitor and stores collected log data from different sources in the Log Analytics workspace. Therefore, Denodo Log files can be ingested and stored as custom logs in a Log Analytics workspace using the Azure Log Analytics agent.

In order to collect Denodo Log data, first we need to create a Log Analytics workspace and install and connect agents for all the machines to send data to the Azure Monitor service. After performing these steps, define a custom log file in the Azure portal:

1. Select **Log Analytics workspaces** > ***my_workspace*** > **Advanced Settings**.

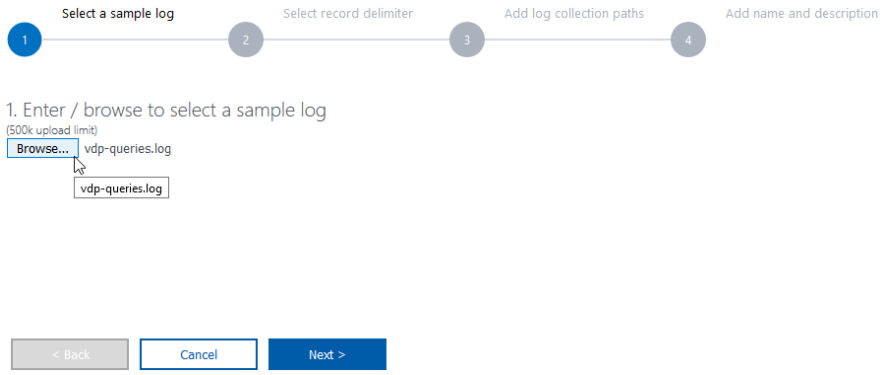


2. Click on **Data** > **Custom logs**.

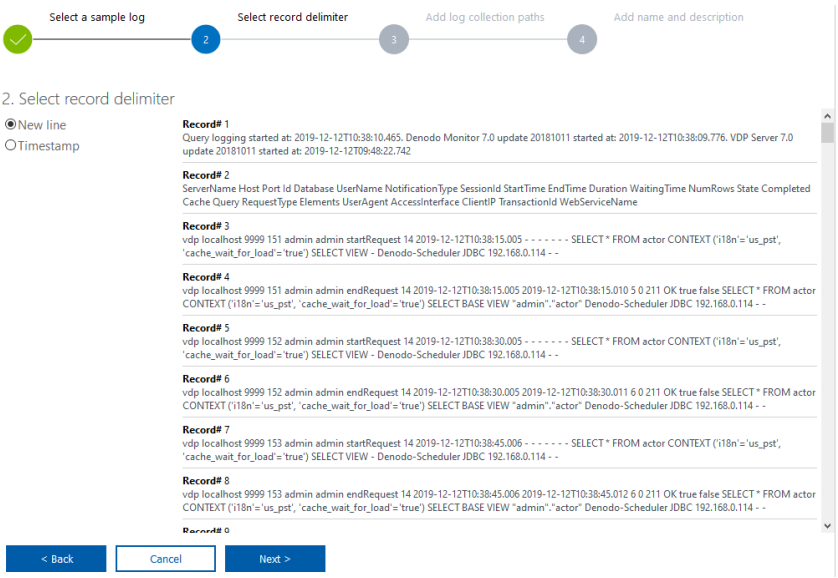


3. Click the **Add +** button to open the Custom Log Wizard.

4. Upload a Denodo Log file in order to use it as a sample file with the data to monitor.



5. The wizard will parse and display the entries in the uploaded file to validate.



   Given that Denodo Logs have a single entry per line, use the **`New line`** as delimiter.

6. Click Next.

7. Define the log collection path on the agent where it can locate the Denodo Log.

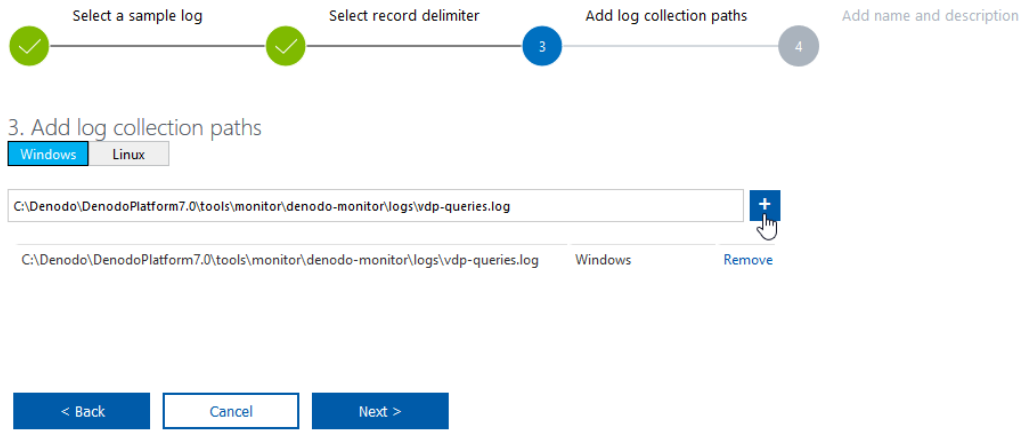
   Denodo Server Logs are under `<DENODO_HOME>/logs/vdp` while Denodo Monitor Logs are under `<DENODO_HOME>/tools/monitor/denodo-monitor/logs` or, when the Denodo Monitor is launched by the Solution Manager, under `<SOLUTION_MANAGER_HOME>/resources/solution-manager-monitor/work`. Take into account that working with the Solution Manager, the Denodo Monitor generates log files in different folders:

   - Log files of environments: `<EnvironmentName>/logs`
   - Log files of clusters: `<EnvironmentName>/<ClusterName>/logs`
   - Log files of servers: `<EnvironmentName>/<ClusterName>/<ServerName>/logs`

Note that these paths are different in Denodo 7.0. In this version, a `<timestamp>` folder containing the `<logs>` folder, among other data, is created every time the Solution Manager starts the Denodo Monitor:

- Log files of environments: `<EnvironmentName>/<timestamp>/logs`
- Log files of clusters: `<EnvironmentName>/<ClusterName>/<timestamp>/logs`
- Log files of servers: `<EnvironmentName>/<ClusterName>/<ServerName>/<timestamp>/logs`

Be aware that the path you establish to add a log collection can be a specific path + name for the log file with which the agent will load the Denodo Log data. For example, to monitor the `vdp-queries.log` file, from the Denodo Monitor, the folder `<DENODO_HOME>\tools\monitor\denodo-monitor\logs\vdp-queries.log` should be added to the collection paths.



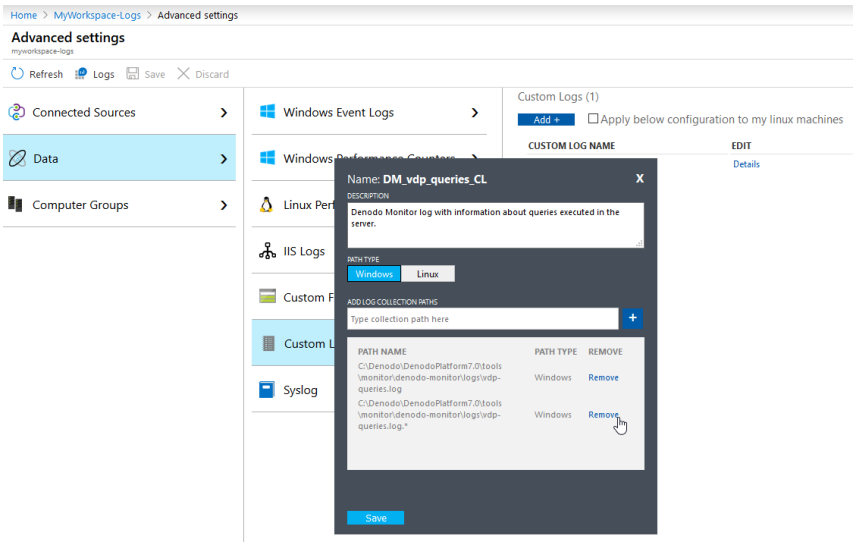
Furthermore, it is possible provide a path with a wildcard for the name in order to load files that had already been created before starting monitoring due to the log rotation policy:

```
<DENODO_HOME>/tools/monitor/denodo-monitor/logs/vdp-queries.log.*
```

Note that it is also possible to provide multiple paths for a single log file to load Denodo Log data adding more than one path, for example:

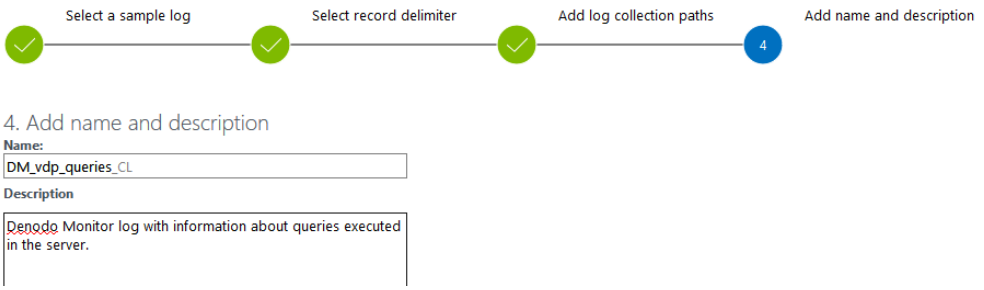
```
<DENODO_HOME>/tools/monitor/denodo-monitor/logs/vdp-queries.log
<DENODO_HOME>/tools/monitor/denodo-monitor/logs/vdp-queries.log.*
```

This means that it is possible to add a path to monitor a specific log file and, in the same log definition, a path to load previously created files due to a rotation mechanism. Once they have been collected, edit the log definition in order to remove the path that refers to rotated files to avoid loading duplicates in successive rotations of the main file. So, in the preceding example, edit the collection path after the initial load in order to leave only the path for the specific log file (`<DENODO_HOME>/tools/monitor/denodo-monitor/logs/vdp-queries.log`):



8. Click Next.

9. Provide a name and description for the log. The name must have the _CL suffix, it is automatically provided and it is used to distinguish the log definition as a custom log.



Finally, click Next to save the custom log definition.

Note that, by default, all configuration changes are automatically pushed to all agents.

Now the Denodo Log data collected by Azure Monitor can be analyzed with queries using a version of the Kusto query language. You can create and test queries using Log Analytics in the Azure portal and then either directly analyze the data using these tools or save queries for use with visualizations or alert rules. Also note that telemetry such as events and traces are stored as logs in addition to performance data so that it can all be combined for analysis.

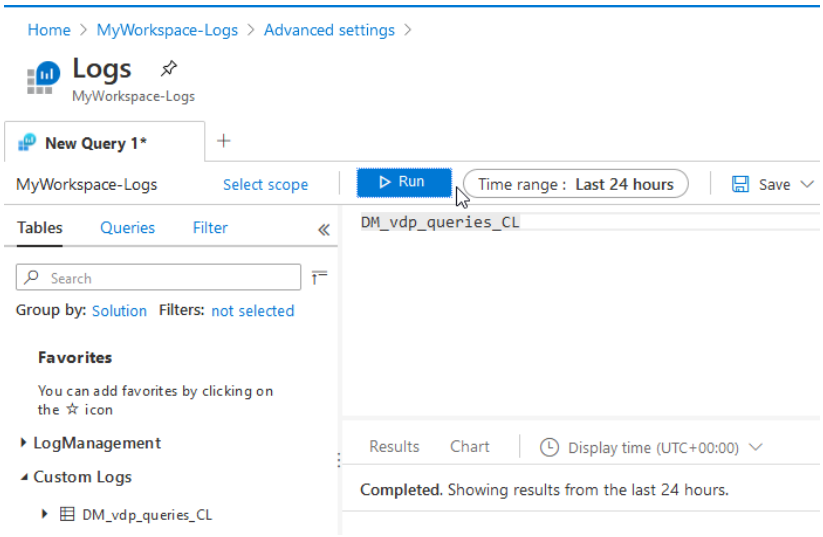
2.1 DATA COLLECTION

It may take up to an hour for the initial data from a new custom log to appear in Azure Monitor. It will start collecting entries from the logs found in the path specified from the point defined the custom log. It will not retain the entries uploaded during the custom log creation, but it will collect already existing entries in the log files that it locates.

3 INSPECTING AND SEARCHING LOG FILES

Once the Azure Log Analytics agent collects and stores the log files, it is possible to go to the Azure portal and use the Log Analytics web tool to write Azure Monitor log queries in order to retrieve and analyze Denodo Log data.

Custom log records will be available with a log query using the name that you gave the custom log as the `Type` in the query and setting a suitable time range:



Custom log records have several properties:

- **TimeGenerated**. Date and time that the record was collected by Azure Monitor. Be aware that Azure Monitor will collect new entries from each custom log approximately every 5 minutes. Nonetheless, the specific latency for any particular data will be affected for a variety of factors. In any case, keep in mind that it does not represent the instant in which the entry was created in the Denodo Log file. Also note that the Time range selected to run a query uses the `TimeGenerated` property.
- **SourceSystem**. Type of agent the record was collected from.
- **RawData**. Full text of the collected entry.
- **ManagementGroupName**. Name of the management group for System Center Operations Manage agents. For other agents, this is `AOI-<workspace ID>`.

3.1 PARSING TEXT DATA

The entire log entry will be stored in a single property called `RawData`, however it can be separated into individual properties in order to use each piece of information. This separation will allow us to use those properties to create metric measurements and publish them in a shared dashboard or use the query to create an alert.

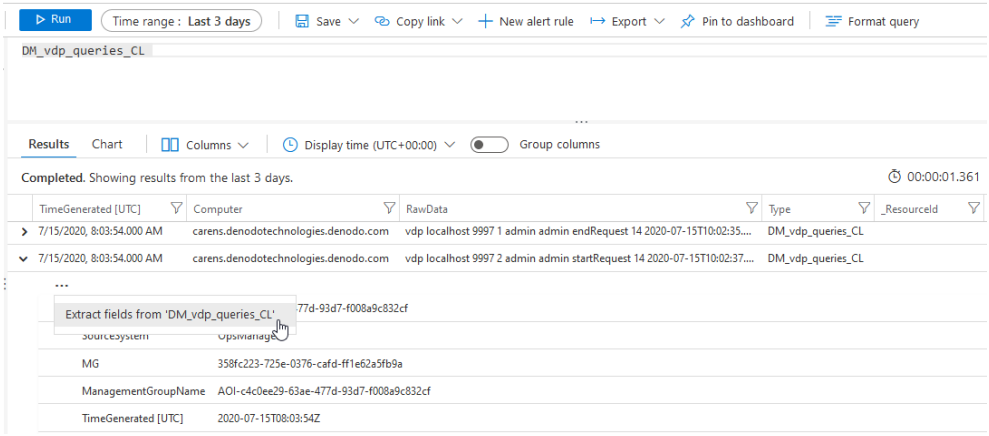
3.1.1 Parsing data at collection time

Azure Monitor has the **Custom Fields** feature that allows to extend existing records in a Log Analytics workspace by adding searchable fields and these custom fields

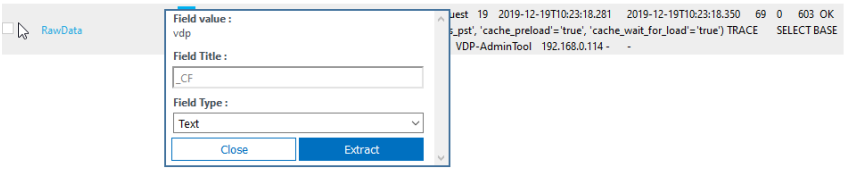
are automatically populated from data extracted from other properties in the same record. This is a parsing method at collection time which means that queries do not have to include any parsing logic.

In this scenario, Azure Monitor learns about the data to extract from record examples using a **Field Extraction Wizard**. To do this:

1. Create a log query and select a record that Log Analytics will use to act as a model for extracting the fields.
2. Expand the record properties, click the ellipse to the left of the top property of the record, and select **Extract fields from**.



3. The Field Extraction Wizard is opened, and the record selected is displayed in the Main Example column. Select the fields and give them a name and a type. If the selection is not exactly the desired one, select additional fields to narrow the criteria.



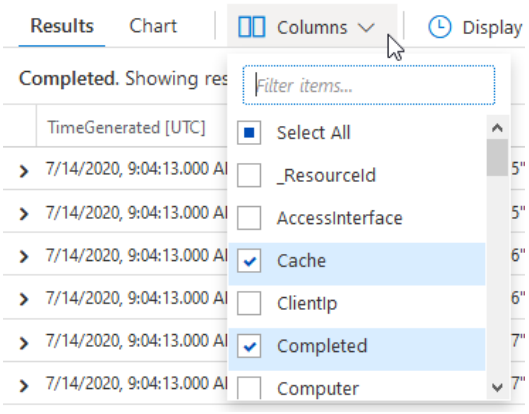
For more detailed information about creating custom fields see [Create custom fields in a Log Analytics workspace in Azure Monitor](#).

Since this method offers few parsing options, it is difficult to configure to get a proper parsing and it increases latency time for collecting data; parsing data at query time is the recommended option in order to parse records that come from Denodo Logs.

3.1.2 Parsing data at query time

The Kusto query language allows to extract fields from the `RawData` field using the Log Analytics web tool.

Notice that the columns displayed in a query result may not be all those included in the query, by default it only shows 20 columns. You can `SELECT ALL` using the Columns menu:



3.1.2.1 Parsing data at query time: Denodo Monitor Logs

Denodo Monitor Logs are available under `<DENODO_HOME>/tools/monitor/denodo-monitor/logs` or under `<SOLUTION_MANAGER_HOME>/resources/solution-manager-monitor/work`, in case the Denodo Monitor of the Solution Manager has been launched. In this second scenario, the Denodo Monitor can collect the execution logs from a single Virtual DataPort server or from all the servers of a cluster or environment. Each log file name will start with the respective server name. See the Monitoring section of the Solution Manager Administration Guide for more information.

Since Denodo Monitor Logs are tab delimited files, it is possible to use the `split function` offered by Kusto to get a string array with substrings representing fields. Besides, using the headers that can be found in the Denodo Monitor logs, it is possible to give a name to the fields. Sample queries are available in the **Appendix A: Parsing data at query time from Denodo Monitor Logs**.

In addition, take into consideration that:

- In Azure Monitor all dates are expressed in UTC. This means that dates from Denodo Log files are loaded disregarding the time zone. To convert the values in the log files from a specific timezone to UTC it is possible to add the time offset of the timezone in the logs to the date extracted from them:

```
| extend StartTime = todatetime(vdpQueriesFields[8]) + 6h
```

- To discard some records such as headers filters can be used:

```
| where RawData !startswith "ServerName" and RawData !startswith
"Connection logging started at:"
```

3.1.2.2 Parsing data at query time: Denodo Server Logs

Denodo Server Logs (`<DENODO_HOME>/logs/vdp/`) are not tab delimited so regular expressions must be used in the log query to get the Denodo Server Log fields. You can find examples of queries in the **Appendix B: Parsing data at query time from Denodo Server Logs**.

4 CREATING AN ALERT

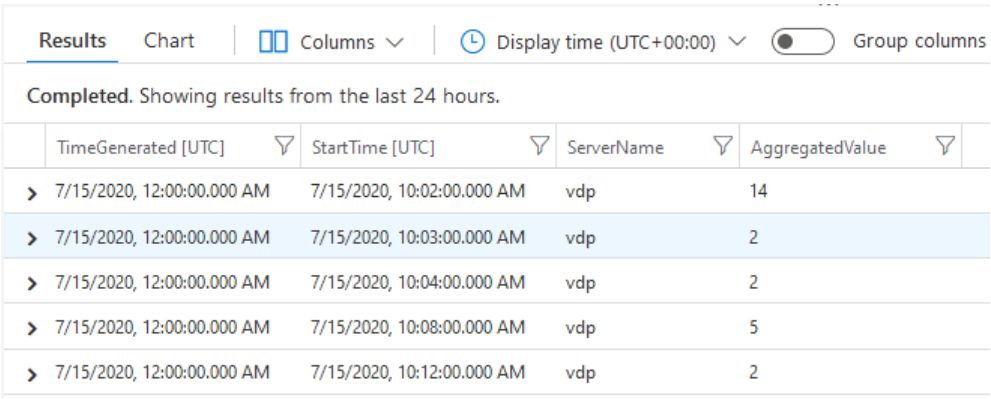
Alerts proactively send notifications when important conditions are found in your monitoring data. Azure Monitor provides Log alerts that allow you to use Azure's analytics platform as a base for alerts.

4.1 CREATING LOG QUERIES FOR ALERTS

As the **Parsing data at query time** section explains, it is possible to parse Denodo Logs (Denodo Monitor or Denodo Server logs) to get their properties. Afterwards, it will be easy to create a log query to process the data and return metric measurements. For example, you can monitor the log `vdp-queries.log` from Denodo Monitor in order to create a metric measurement with the number of queries initiated per minute.

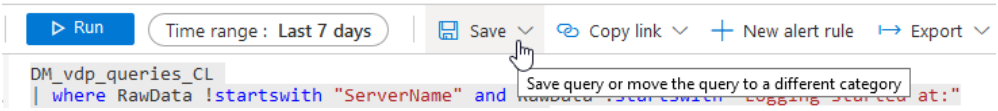
```
DM_vdp_queries_CL
| where RawData !startswith "ServerName" and RawData !startswith "Logging
started at:"
| extend vdpQueriesFields = split(RawData, '\t')
| extend ServerName = tostring(vdpQueriesFields[0])
| extend NotificationType = tostring(vdpQueriesFields[6])
| extend StartTime = todatetime(vdpQueriesFields[8])
| where NotificationType == "startRequest" and StartTime > ago(15m)
| summarize AggregatedValue=count() by bin(TimeGenerated,1d),
bin(StartTime,1m), ServerName
| sort by StartTime asc
```

This log search will return rows with the number of queries started each minute, grouped by `ServerName`. The second `where` operator sets a 15 minute time range to take into account this time span over which to execute the query.

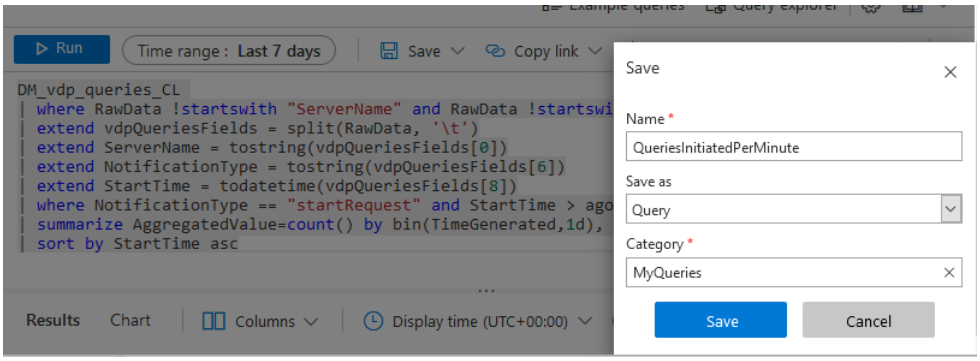


Taking into account that the aim of this query is to get a metric measurement to create an alert, the search query should contain `'AggregatedValue'`, with the values for the metric measurement, and `'bin(TimeGenerated, [roundTo])'`.

Once you have created the query, you can save it using the **Save** icon on the top bar:

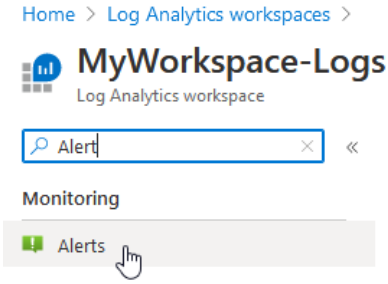


You must establish a name, `QueriesInitiatedPerMinute`, and a category (used to organize saved queries in Query explorer):

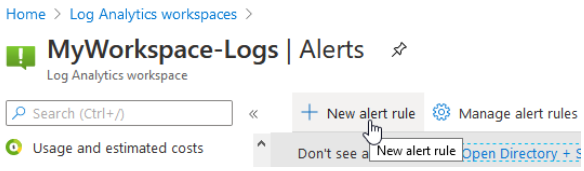


4.2 MANAGING A LOG ALERT

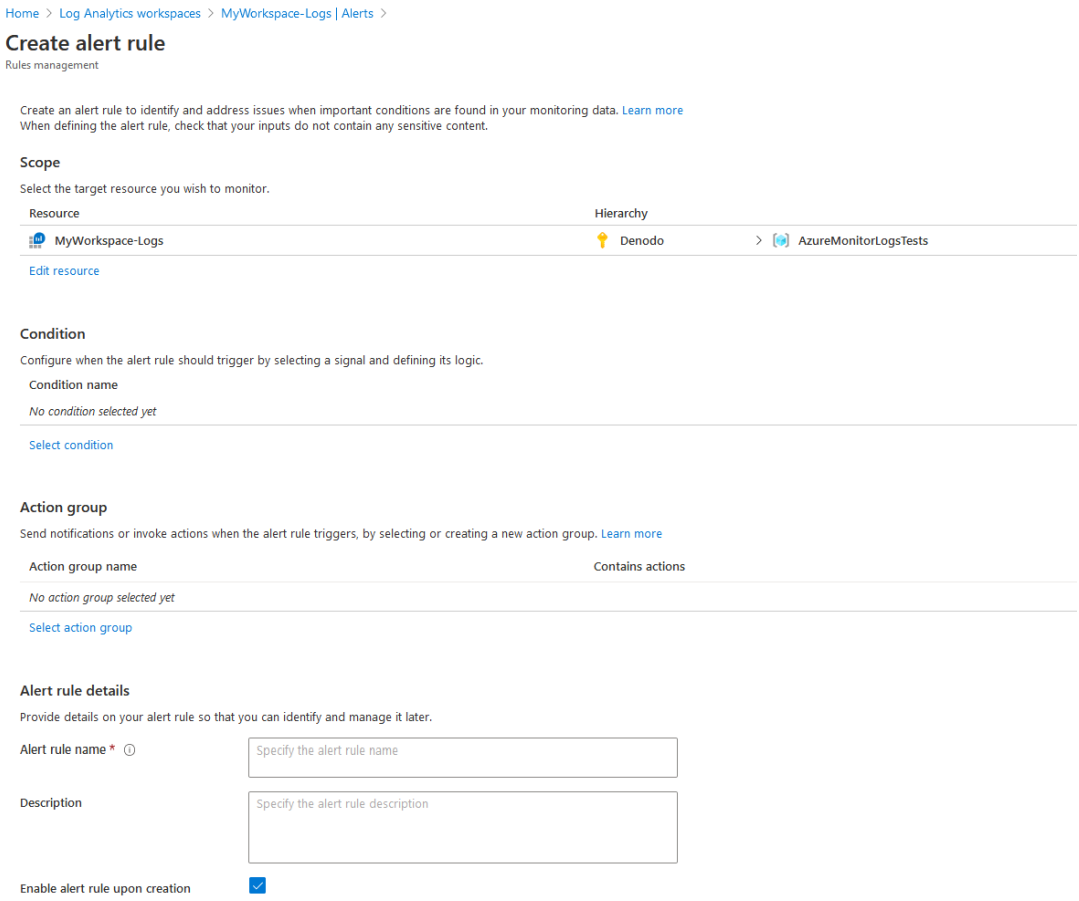
1. Select **Log Analytics workspaces** > ***my_workspace*** > **Alerts**.



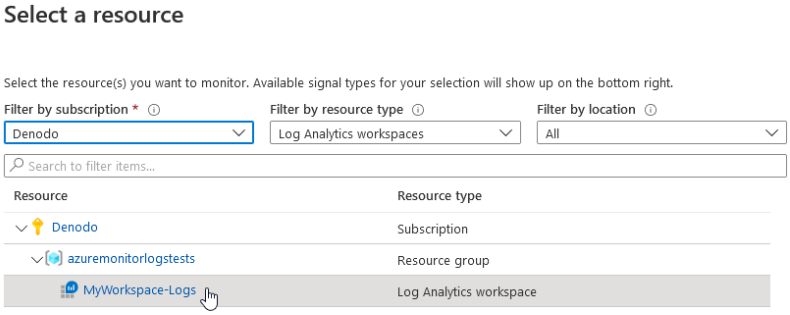
2. Select the **New Alert Rule** button to create a new alert:



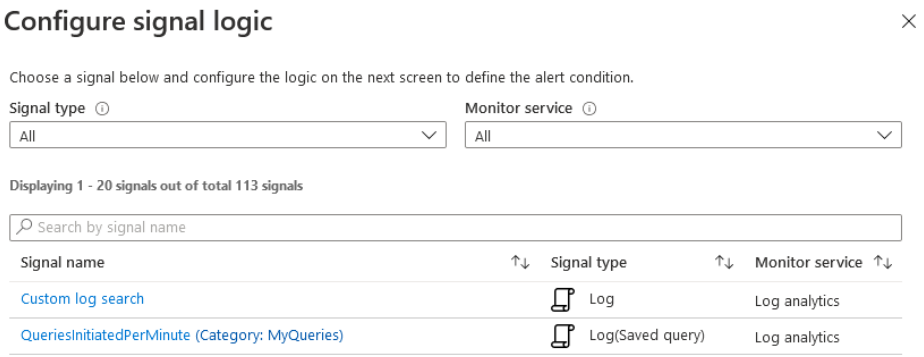
The Create Alert section is shown:



3. In the **Scope** section select the resource to monitor. In our example, the workspace:



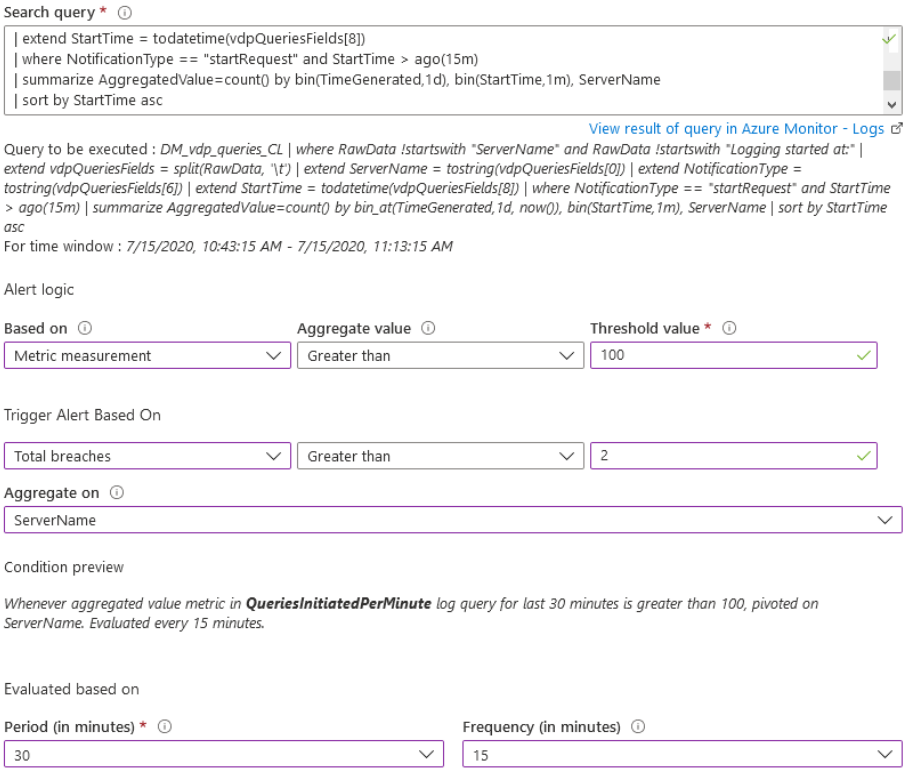
4. Click on the **Select condition** link of the **Condition** section to view the list of signal options available for the selected resource. Select the saved query, `QueriesInitiatedPerMinute`, from the signal list:



The **Custom log search** option can also be used but it means that the query is written during the configuration.

After selecting the `QueriesInitiatedPerMinute` query, configure the alert logic.

The alert will be based on **Metric measurement**, this means that an alert is created if each aggregate value in the results, the number of queries initiated per minute grouped by `ServerName`, exceeds 100 units.



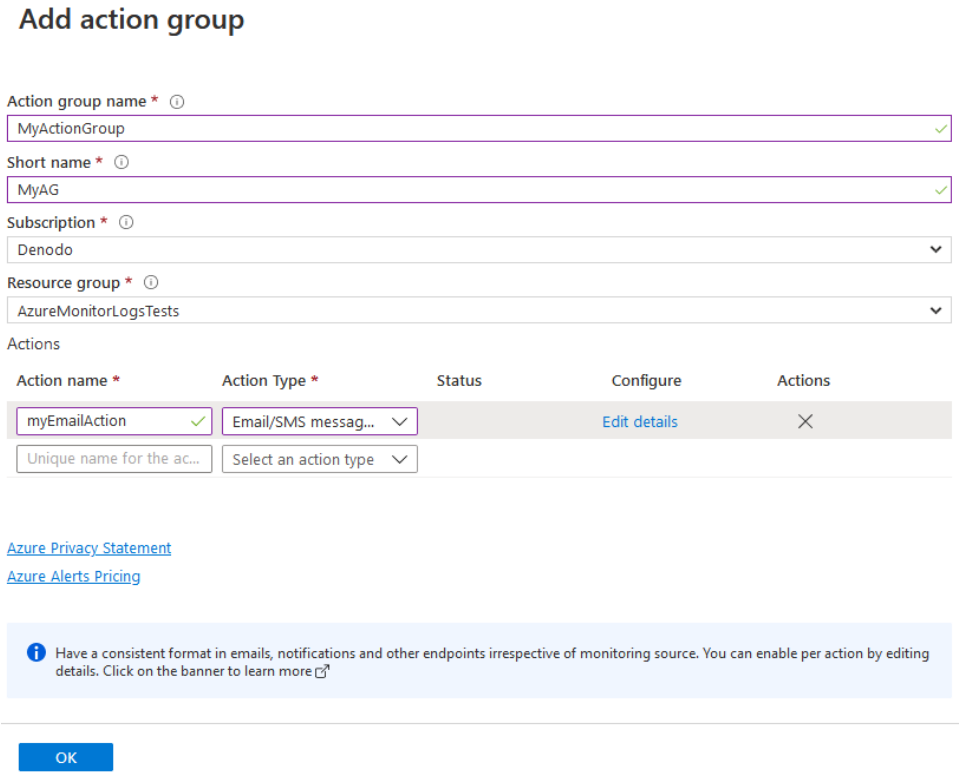
The alert will be created when the total number of breaches (the number of times the threshold is exceeded) is greater than 2 in the last 15 minutes. Note that the period is set to 30 minutes but the query only retrieves values whose StartTime is included in the last 15 minutes. This value established in the log query (`StartTime > ago(15m)`) must be the same as the one selected in the frequency field. This is done to try to avoid losing data whose `TimeGenerated` value is not included in the desired range, although based on its `StartTime` this data should be included in this execution results.

In this step, using the `Search query` field, it is possible to see and modify the search query or view its results in Azure Monitor.

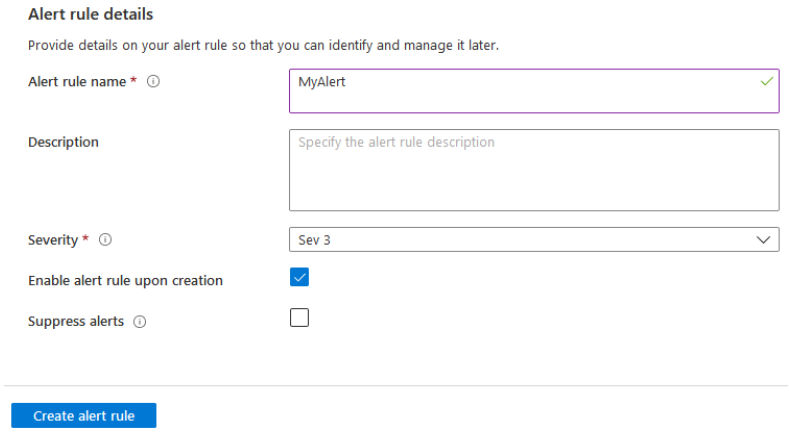
5. The **Action group** section allows users to select or create an action group to be triggered for the alert rule when the alert condition is met.

   Click on the **Select action group** link and then click on the **Create action group** button. Set the `Action group name`, the `Short name` (used to identify which Action group was the source of the notification), the `Subscription` in which the group will be saved and the `Resource group` the group will associated with.

   Define an action with the type `Email/SMS message/Push/Voice` to send an email notification and click on the `OK` button.



6. Finally, set the `Alert rule name` and, optionally, a `Description`, change the `Severity` or check the `Suppress Alerts` box to turn on suppressions for the alert rule.

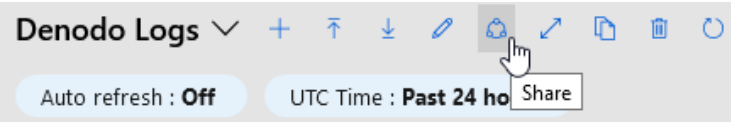


7. Click on the **Create alert rule** button to create the alert in Azure Monitor.

5 CREATING A DASHBOARD OF LOG ANALYTICS DATA

Log Analytics dashboards can visualize log queries giving the ability to find, correlate, and share IT operational data.

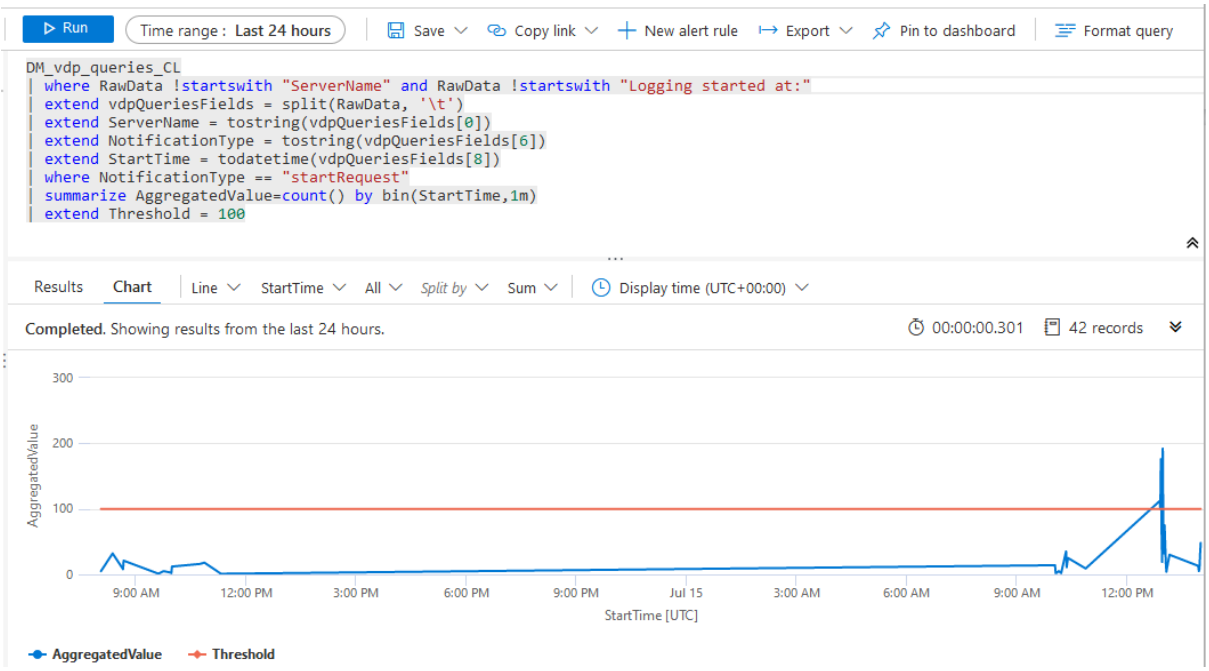
For this purpose, a shared dashboard is needed as private dashboards do not allow the addition of log queries. Dashboards are private by default, to make it visible to others, use the **Share** button that appears alongside the other dashboard commands.



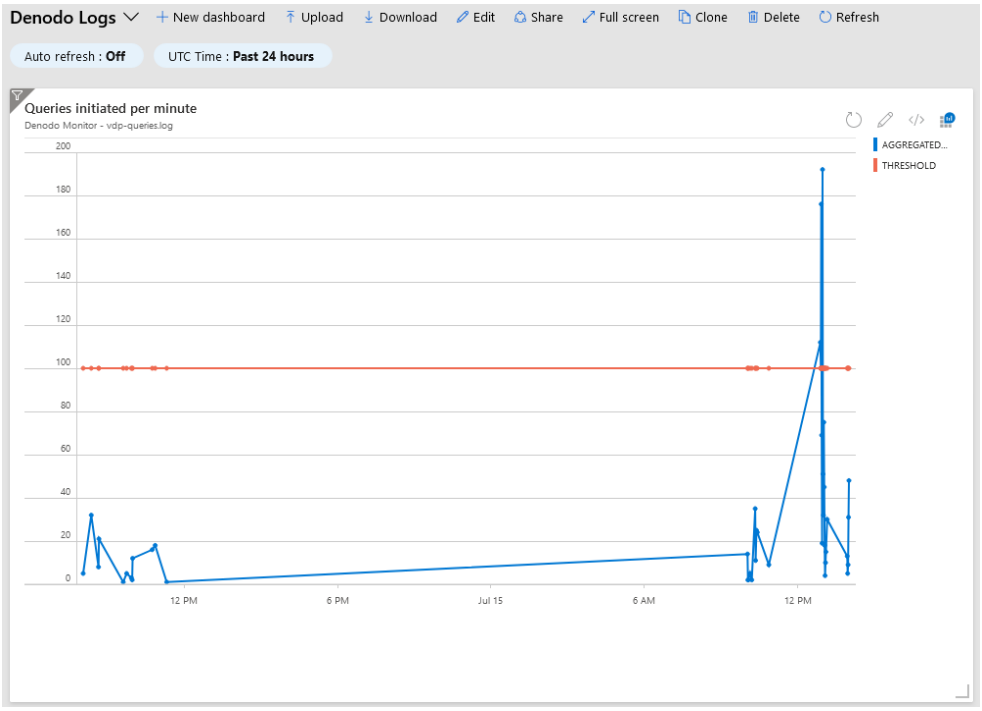
After creating the shared dashboard, write the log query and check its results in the Log Analytics portal. For example, you can perform a query, over the `vdp-queries.log` from the Denodo Monitor, to see the number of queries initiated per minute:

```
DM_vdp_queries_CL
| where RawData !startswith "ServerName" and RawData !startswith "Logging
started at:"
| extend vdpQueriesFields = split(RawData, '\t')
| extend ServerName = tostring(vdpQueriesFields[0])
| extend NotificationType = tostring(vdpQueriesFields[6])
| extend StartTime = todatetime(vdpQueriesFields[8])
| where NotificationType == "startRequest"
| summarize AggregatedValue=count() by bin(StartTime,1m)
| extend Threshold = 100
```

Using the `extend` command it is possible to add a constant column to the dataset and use it to add a reference line to the chart that can help to easily identify if the number of queries exceeded a specific threshold.



Finally, pin this to the shared dashboard created earlier by selecting the **Pin to dashboard** button from the top right corner of the page and then selecting the dashboard name.



Click on the edit button to change the title and subtitle.

Remember to click on **Publish changes**, in the banner, after performing the modifications.

6 APPENDIX A: PARSING DATA AT QUERY TIME FROM DENODO MONITOR LOGS

Query for `vdp-connections.log` or `<DenodoServerName>-connections.log`:

```
<custom_log_name>
|  where RawData !startswith "ServerName" and RawData !startswith
"Connection logging started at:"
| extend fields = split(RawData, '\t')
| extend ServerName = tostring(fields[0])
| extend Host = tostring(fields[1])
| extend Port = tolong(fields[2])
| extend NotificationType = tostring(fields[3])
| extend ConnectionId = tolong(fields[4])
| extend ConnectionStartTime = todatetime(fields[5])
| extend ConnectionEndTime = todatetime(fields[6])
| extend ClientIP = tostring(fields[7])
| extend UserAgent = tostring(fields[8])
| extend AccessInterface = tostring(fields[9])
| extend SessionId = tolong(fields[10])
| extend SessionStartTime = todatetime(fields[11])
| extend SessionEndTime = todatetime(fields[12])
| extend Login = tostring(fields[13])
| extend DatabaseName = tostring(fields[14])
| extend WebServiceName = tostring(fields[15])
| extend JMSQueueName = tostring(fields[16])
| extend IntermediateClientIP = trim_end("\r\n|\n", tostring(fields[17]))
```

Query for `vdp-datasources.log` or `<DenodoServerName>-datasources.log`:

```
<custom_log_name>
|  where RawData !startswith "Date" and RawData !startswith "Logging
started at:"
| extend fields = split(RawData, '\t')
| extend Date = todatetime(fields[0])
| extend DatabaseName = tostring(fields[1])
| extend DataSourceType = tostring(fields[2])
| extend DataSourceName = tostring(fields[3])
| extend ActiveRequests = tolong(fields[4])
| extend NumRequests = tolong(fields[5])
| extend MaxActive = tolong(fields[6])
| extend NumActive = tolong(fields[7])
| extend NumIdle = tolong(fields[8])
| extend PingStatus = tostring(fields[9])
```

```
| extend PingExecutionTime = todatetime(fields[10])
| extend PingDuration = tolong(fields[11])
| extend PingDownCause = trim_end("\r\n|\n", tostring(fields[12]))
```

Query for `vdp-loadcacheprocesses.log` or `<DenodoServerName>-loadcacheprocesses.log`:

```
<custom_log_name>
| where RawData !startswith "SessionId" and RawData !startswith "Load cache processes logging started at:"
| extend fields = split(RawData, '\t')
| extend SessionId= tolong(fields[0])
| extend ServerName = tostring(fields[1])
| extend Host = tostring(fields[2])
| extend Port = tolong(fields[3])
| extend NotificationType = tostring(fields[4])
| extend NotificationTimestamp = todatetime(fields[5])
| extend Id = tolong(fields[6])
| extend QueryPatternId = tolong(fields[7])
| extend DatabaseName = tostring(fields[8])
| extend ViewName = tostring(fields[9])
| extend SqlViewName = tostring(fields[10])
| extend ProjectedFields = tostring(fields[11])
| extend NumConditions = tolong(fields[12])
| extend VDPConditionList = tostring(fields[13])
| extend CacheStatus = tostring(fields[14])
| extend TtlStatusInCache = tostring(fields[15])
| extend TtlInCache = tolong(fields[16])
| extend QueryPatternState = tostring(fields[17])
| extend Exception = tostring(fields[18])
| extend NumOfInsertedRows = tolong(fields[19])
| extend NumOfReceivedRows = tolong(fields[20])
| extend StartQueryPatternStorageTime = todatetime(fields[21])
| extend EndQueryPatternStorageTime = todatetime(fields[22])
| extend QueryPatternStorageTime = todatetime(fields[23])
| extend StartCachedResultMetadataStorageTime = todatetime(fields[24])
| extend EndCachedResultMetadataStorageTime = todatetime(fields[25])
| extend CachedResultMetadataStorageTime = tolong(fields[26])
| extend StartDataStorageTime = todatetime(fields[27])
| extend EndDataStorageTime = todatetime(fields[28])
| extend DataStorageTime = tolong(fields[29])
```

Query for `vdp-queries.log` or `<DenodoServerName>-queries.log`:

```
<custom_log_name>
| where RawData !startswith "ServerName" and RawData !startswith "Logging started at:"
| extend fields = split(RawData, '\t')
| extend ServerName = tostring(fields[0])
| extend Host = tostring(fields[1])
| extend Port = tolong(fields[2])
| extend Id = tolong(fields[3])
| extend Database = tostring(fields[4])
```

```
| extend UserName = tostring(fields[5])
| extend NotificationType = tostring(fields[6])
| extend SessionId = tolong(fields[7])
| extend StartTime = todatetime(fields[8])
| extend EndTime = todatetime(fields[9])
| extend Duration = tolong(fields[10])
| extend WaitingTime = tolong(fields[11])
| extend NumRows = tolong(fields[12])
| extend State = tostring(fields[13])
| extend Completed = tostring(fields[14])
| extend Cache = tostring(fields[15])
| extend Query = tostring(fields[16])
| extend RequestType = tostring(fields[17])
| extend Elements = tostring(fields[18])
| extend UserAgent = tostring(fields[19])
| extend AccessInterface = tostring(fields[20])
| extend ClientIp = tostring(fields[21])
| extend TransactionId = tostring(fields[22])
| extend WebServiceName = trim_end("\r\n|\n", tostring(fields[23]))
```

Query for `vdp-resources.log` or `<DenodoServerName>-resources.log`:

```
<custom_log_name>
| where RawData !startswith "ServerName" and RawData !startswith "Logging
started at:"
| extend fields = split(RawData, '\t')
| extend ServerName = tostring(fields[0])
| extend Host = tostring(fields[1])
| extend Port = tolong(fields[2])
| extend Date = todatetime(fields[3])
| extend Metaspace = tostring(fields[4])
| extend PS_Survivor_Space = tostring(fields[5])
| extend PS_Old_Gen = tostring(fields[6])
| extend PS_Eden_Space = tostring(fields[7])
| extend Code_Cache = tostring(fields[8])
| extend HeapMemoryUsage = tostring(fields[9])
| extend NonHeapMemoryUsage = tostring(fields[10])
| extend LoadedClassCount = tolong(fields[11])
| extend TotalLoadedClassCount = tolong(fields[12])
| extend ThreadCount = tolong(fields[13])
| extend PeakThreadCount = tolong(fields[14])
| extend VDPTotalConn= tolong(fields[15])
| extend VDPActiveConn = tolong(fields[16])
| extend VDPActiveRequests = tolong(fields[17])
| extend VDPWaitingRequests = tolong(fields[18])
| extend VDPTotalMem = tolong(fields[19])
| extend VDPMaxMem = tolong(fields[20])
| extend CPU_percentage= todecimal(fields[21])
| extend GC_CC_PS_MarkSweep = tolong(fields[22])
| extend GC_CC_PS_Scavenge = tolong(fields[23])
| extend GC_CT_PS_MarkSweep = tolong(fields[24])
```

```
| extend GC_CT_PS_Scavenge = tolong(fields[25])
| extend GC_percentage_PS_MarkSweep = tolong(fields[26])
| extend GC_percentage_PS_Scavenge = todecimal(fields[27])
|     extend     GC_percentage     =     todecimal(     trim_end("\r\n|\n",
tostring(fields[28])))
```

7 APPENDIX B: PARSING DATA AT QUERY TIME FROM DENODO SERVER LOGS

Query for `vdp.log`:

```
<custom_log_name>
| where RawData matches regex "^\\d+.*?"
| parse kind = regex flags = U RawData with Id:long "\\s" Thread:string
"\\s" Level:string "(\\s\\s|\\s)" Date:datetime "\\s" Category:string
"\\s" NDC:string "\\s-\\s" Message:string "\\n"
```

Query for `vdp-cache.log`:

```
<custom_log_name>
| parse kind = regex flags = U RawData with Id:long "\\s" Thread:string
"\\s" Date:datetime "\\s" NDC:string "\\s-\\s" Message:string "\\n"
```

Query for `vdp-queries.log`:

```
<custom_log_name>
| parse kind = regex flags = U RawData with Id:long "\\s" Thread:string
"\\s" Date:datetime "\\s" NDC:string "\\s-\\t" QueryId:long "\\t"
Database:string "\\t" UserName:string "\\t" NotificationType:string "\\t"
SessionId:long "\\t" StartTime:string "\\t" EndTime:string "\\t"
Duration:long "\\t" WaitingTime:string "\\t" NumRows:string "\\t"
State:string "\\t"
| parse kind = regex flags = U RawData with
"\\d+\\s.*\\s.*\\s.*\\s-\\t.*\\t.*\\t.*\\t.*\\t\\d+\\t.*\\t.*\\t.*\\t.*\\
t.*\\t.*\\t" Completed:string "\\t" Cache:string "\\t" Query:string "\\t"
TotalRequests:long "\\t" TotalJDBCConnectionTime:string "\\t"
PartialJDBCConnectionTime:string "\\t" TotalJDBCResponseTime:string "\t"
PartialJDBCResponseTime:string "\\t" FeedCache:string "\\t"
CachedResultsViewNames:string "\\t" TotalCacheResultsCheckTime:string
"\\t" PartialCacheResultCheckTime:string "\\t" JDBCConnectionIds:string
"\\t" RequestType:string "\\t" Elements:string "\\t"
| parse kind = regex flags = U RawData with
"\\d+\\s.*\\s.*\\s.*\\s-\\t.*\\t.*\\t.*\\t.*\\t\\d+\\t.*\\t.*\\t.*\\t.*\\
t.*\\t.*\\t.*\\t.*\\t.*\\t\\d+\\t.*\\t.*\\t.*\\t.*\\t.*\\t.*\\t.*\\t.*\\t
.*\\t.*\\t.*\\t" UserAgent:string "\\t" AccessInterface:string "\\t"
ClientIP:string "\\t" TransactionID:string "\\n"
```

Query for `vdp-requests.log`:

```
<custom_log_name>
```

```
| parse kind = regex flags = U RawData with Id:long "\\s" Thread:string
"\\s" Level:string "(\\s\\s|\\s)" Date:datetime "\\s" Category:string
"\\s" NDC:string "\\s-\\s" Request:string "\\n"
```