Information Query Store Collects

The Query Store collects a fairly narrow but extremely rich set of data. Figure 11-2 represents the system tables and their relationships.

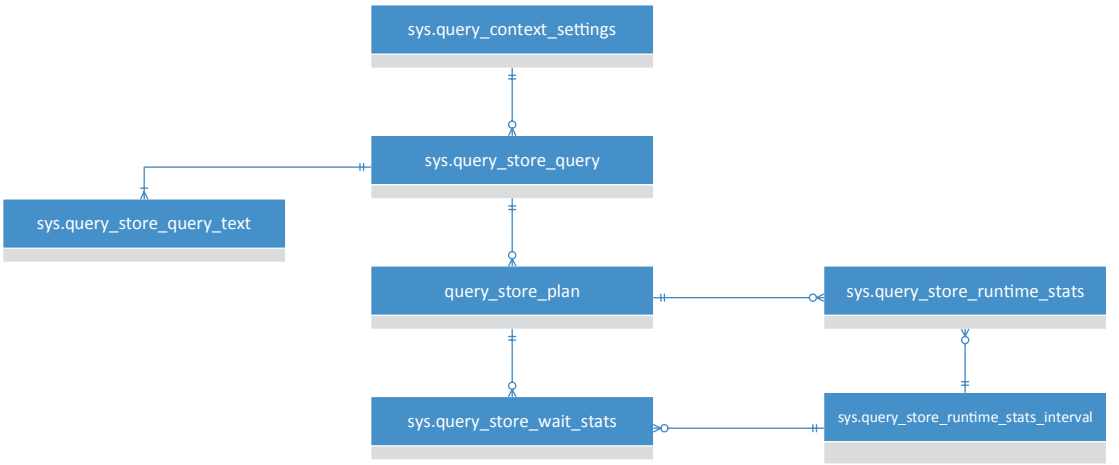


Figure 11-2. *System views of the Query Store*

The information stored within the Query Store breaks down into two basic sets. There is the information about the query itself, including the query text, the execution plan, and the query context settings. Then there is the runtime information that consists of the runtime intervals, the wait statistics, and the query runtime statistics. We'll approach each section of information separately, starting with the information about the query.

Query Information

The core piece of data to the Query Store is the query itself. The query is independent from, though it may be part of, stored procedures or batches. It comes down to the fundamental query text and the `query_hash` value (a hash of the query text) that lets you identify any given query. This data is then combined with the query plans and the actual query text. Figure 11-3 shows the basic structure and some of the data.

www.EBooksWorld.ir

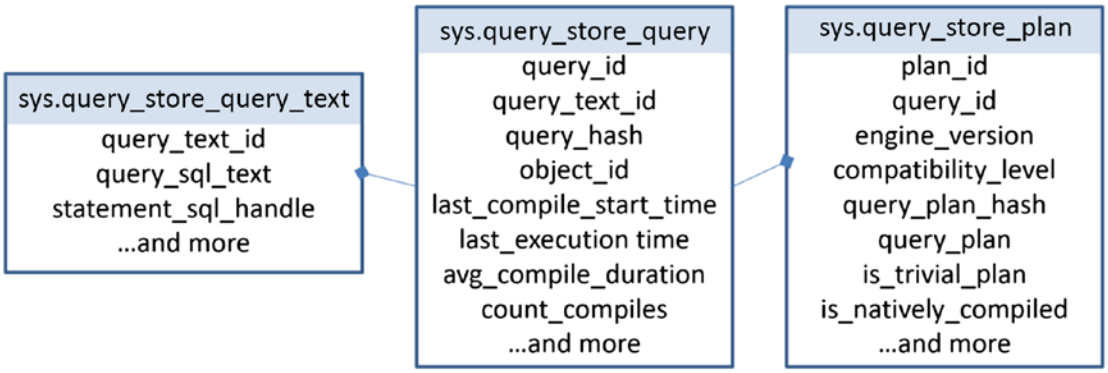


Figure 11-3. *Query information stored within the Query Store*

These are system tables stored in the Primary file group of any database that has the Query Store enabled. While there are good reports built into the Management Studio interface, you can write your own queries to access the information from the Query Store. For example, this query could retrieve all the query statements for a given stored procedure along with the execution plan:

```
SELECT qsq.query_id,
       qsq.object_id,
       qsqt.query_sql_text,
       CAST(qsp.query_plan AS XML) AS QueryPlan
FROM sys.query_store_query AS qsq
    JOIN sys.query_store_query_text AS qsqt
        ON qsq.query_text_id = qsqt.query_text_id
    JOIN sys.query_store_plan AS qsp
        ON qsp.query_id = qsq.query_id
WHERE qsq.object_id = OBJECT_ID('dbo.ProductTransactionHistoryByReference');
```

While each individual query statement is stored within the Query Store, you also get the `object_id`, so you can use functions such as `OBJECT_ID()` as I did to retrieve the information. Note that I also had to use the `CAST` command on the `query_plan` column. This is because the Query Store rightly stores this column as text, not as XML. The XML data type in SQL Server has a nesting limit that would require two columns, XML for those that meet the requirement and `NVARCHAR(MAX)` for those that don't. When building

www.EBooksWorld.ir

the Query Store, they addressed that issue by design. If you want to be able to click the results, similar to Figure 11-4, to see the execution plan, you'll need to use CAST as I did earlier.

	query_id	object_id	query_sql_text	plan_id	QueryPlan
1	75	1255675521	(@ReferenceOrderID int)SELECT p.Nam...	75	<ShowPlanXML xmlns="http://schemas.microsoft.com...
2	75	1255675521	(@ReferenceOrderID int)SELECT p.Nam...	76	<ShowPlanXML xmlns="http://schemas.microsoft.com...
3	75	1255675521	(@ReferenceOrderID int)SELECT p.Nam...	82	<ShowPlanXML xmlns="http://schemas.microsoft.com...

Figure 11-4. *Information retrieved from the Query Store using T-SQL*

In this instance, for a single query, `query_id` = 75, which is a one-statement stored procedure, I have three distinct execution plans as identified by the three different `plan_id` values. We'll be looking at these plans a little later.

Another thing to note from the results of the Query Store is how the text is stored. Since this statement is part of a stored procedure with parameters, the parameter values that are used in the T-SQL text are defined. This is what the statement looks like within the Query Store (formatting left as is):

```
(@ReferenceOrderID int)SELECT  p.Name,              p.ProductNumber,
              th.ReferenceOrderID      FROM    Production.Product
AS p      JOIN    Production.TransactionHistory AS
th              ON th.ProductID = p.ProductID      WHERE
th.ReferenceOrderID = @ReferenceOrderID
```

Note the parameter definition at the start of the statement. Just a reminder from earlier, this is what the actual stored procedure definition looks like:

```
CREATE OR ALTER PROC dbo.ProductTransactionHistoryByReference (
     @ReferenceOrderID int
    )
AS
BEGIN
    SELECT  p.Name,
            p.ProductNumber,
            th.ReferenceOrderID
    FROM    Production.Product AS p
    JOIN    Production.TransactionHistory AS th
```


www.EBooksWorld.ir

```
            ON th.ProductID = p.ProductID
    WHERE   th.ReferenceOrderID = @ReferenceOrderID;
END
```

The statements within the procedure and statement as stored in the Query Store are different. This can lead to some issues when attempting to find a particular query within the Query Store. Let's look at a different example, shown here:

```
SELECT a.AddressID,
       a.AddressLine1
FROM Person.Address AS a
WHERE a.AddressID = 72;
```

This is a batch instead of a stored procedure. Executing this for the first time will load it into the Query Store using the process outlined earlier. If we run some T-SQL to retrieve information on this statement as follows, there will be nothing returned:

```
SELECT qsq.query_id,
       qsq.query_hash,
       qsqt.query_sql_text
FROM sys.query_store_query AS qsq
    JOIN sys.query_store_query_text AS qsqt
        ON qsqt.query_text_id = qsq.query_text_id
WHERE qsqt.query_sql_text = 'SELECT a.AddressID,
       a.AddressLine1
FROM Person.Address AS a
WHERE a.AddressID = 72;';
```

Because this statement was so simple, the optimizer was able to perform a process called *simple parameterization* on it. Luckily, the Query Store has a function for dealing with automatic parameterization, `sys.fn_stmt_sql_handle_from_sql_stmt`. That function allows you to find the information from the query as follows:

```
SELECT qsq.query_id,
       qsq.query_hash,
       qsqt.query_sql_text,
       qsq.query_parameterization_type
FROM sys.query_store_query_text AS qsqt
```


www.EBooksWorld.ir

```
    JOIN sys.query_store_query AS qsq
        ON qsq.query_text_id = qsqt.query_text_id
    JOIN sys.fn_stmt_sql_handle_from_sql_stmt(
            'SELECT a.AddressID,
       a.AddressLine1
FROM Person.Address AS a
WHERE a.AddressID = 72;',
            2)  AS fsshfss
        ON fsshfss.statement_sql_handle = qsqt.statement_sql_handle;
```

The formatting and the white space all have to be the same in order for this to work. The hard-coded value can change, but all the rest has to be the same. Running the query results in what you see in Figure 11-5.

	query_id	query_hash	query_sql_text	query_parameterization_type
1	1054	0xDE0BD0B755E53296	(@1 tinyint)SELECT [a].[AddressID],[a].[AddressL...	2

Figure 11-5. *Results showing simple parameterization*

You can see in the `query_sql_text` column where the parameter value for the simple parameterization has been added to the text just as it was for the stored procedure. The bad news is `sy.fn_stmt_sql_handl_from_sql_stmt` currently works only with automatic parameterization. It won't help you locate parameterized statements from any other source. To retrieve that information, you will be forced to use the `LIKE` command to search through the text or, as I did earlier, use the `object_id` for queries in stored procedures.

Query Runtime Data

After you retrieve information about the query and the plan, the next thing you're going to want is to see runtime metrics. There are two keys to understanding the runtime metrics. First, the metrics connect back to the plan, not to the query. Since each plan could behave differently, with different operations against different indexes with different join types and all the rest, capturing runtime data and wait statistics means tying back to the plan. Second, the runtime and wait statistics are aggregated, but they are aggregated by the runtime interval. The default value for the runtime interval is 60 minutes. This means you'll have a different set of metrics for each plan for each runtime interval.

www.EBooksWorld.ir

All this information is available as shown in Figure 11-6.

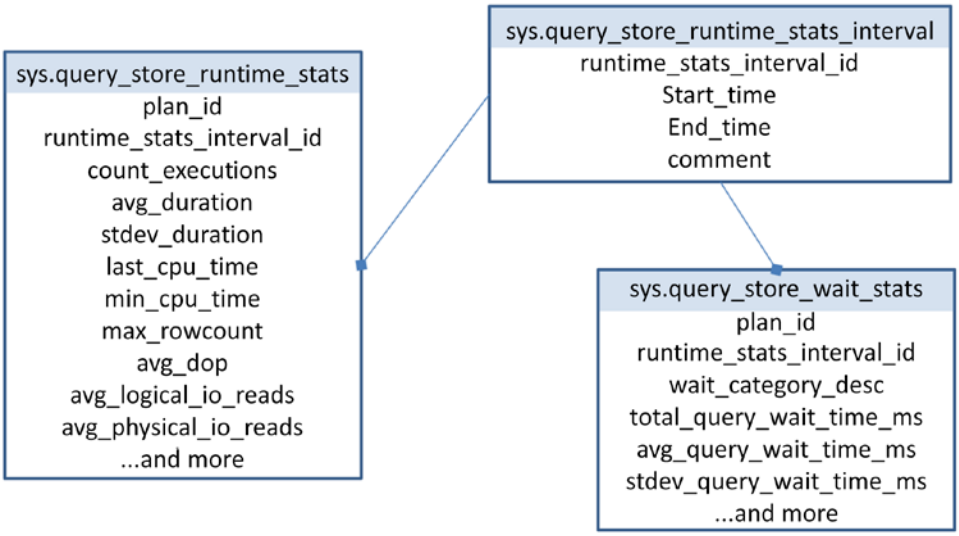


Figure 11-6. *System tables containing runtime and wait statistics*

When you begin to query the runtime metrics, you can easily combine them with the information on the query itself. You will have to deal with the intervals, and the best way to deal with them may be to group them and aggregate them, taking averages of the averages, and so on. That may seem like a pain, but you need to understand why the information is broken up that way. When you're looking at query performance, you need several numbers, such as current performance, hoped-for performance, and future performance after we make changes. Without these numbers to compare, you can't know whether something is slow or whether you have improved it. The same thing goes for the information in the Query Store. By breaking everything apart into intervals, you can compare today to yesterday, one moment in time to another. That's how you can know that performance truly did degrade (or improve), that it ran faster/slower yesterday, and so on. If you have only averages and not averages over time, then you won't see how the behavior changes over time. With the time intervals, you get some of the granularity of capturing the metrics yourself using Extended Events combined with the ease of use of querying the cache.

www.EBooksWorld.ir

A query that retrieves performance metrics for a given moment in time can be written just like this:

```
DECLARE @CompareTime DATETIME = '2017-11-28 21:37';

SELECT CAST(qsp.query_plan AS XML),
       qsrs.count_executions,
       qsrs.avg_duration,
       qsrs.stdev_duration,
       qsws.wait_category_desc,
       qsws.avg_query_wait_time_ms,
       qsws.stdev_query_wait_time_ms
FROM sys.query_store_plan AS qsp
    JOIN sys.query_store_runtime_stats AS qsrs
        ON qsrs.plan_id = qsp.plan_id
    JOIN sys.query_store_runtime_stats_interval AS qsrsi
        ON qsrsi.runtime_stats_interval_id = qsrs.runtime_stats_interval_id
    JOIN sys.query_store_wait_stats AS qsws
        ON qsws.plan_id = qsrs.plan_id
           AND qsws.execution_type = qsrs.execution_type
           AND qsws.runtime_stats_interval_id = qsrs.runtime_stats_
interval_id
WHERE qsq.object_id = OBJECT_ID('dbo.ProductTransactionHistoryByReference')
      AND @CompareTime BETWEEN qsrsi.start_time
                       AND     qsrsi.end_time;
```

Let's break this down. You can see that we're starting off with a query plan just like in the earlier queries, from `sys.query_store_plan`. Then we're combining this with the table that has all the runtime metrics like average duration and standard deviation of the duration, `sys.query_store_runtime_stats`. Because I intend to filter based on a particular time, I want to be sure to join to the `sys.query_store_runtime_stats_interval` table where that data is stored. Then, I'm joining to the `sys.query_store_wait_stats`. There I have to use the compound key that directly links the waits and the runtime stats, the `plan_id`, the `execution_type`, and the `runtime_stats_interval_id`. I'm using a `plan_id` from earlier in the chapter, and I'm setting the data to return a particular time range. Figure 11-7 shows the resulting data.

www.EBooksWorld.ir

	(No column name)	count_executions	avg_duration	stdev_duration	wait_category_desc	avg_query_wait_time_ms	stdev_query_wait_time_ms
1	<ShowPlanXML xmlns="http://schemas.microsoft.com/...	992	579.01310483871	530.744137979874	Memory	0.00604838709677419	0

Figure 11-7. *Runtime metrics and wait statistics for one query in one time interval*

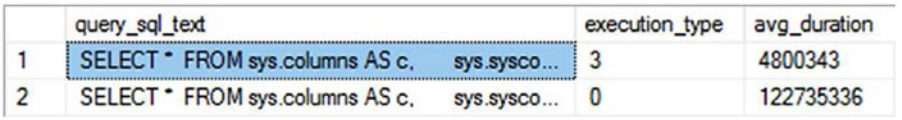
It's important to understand how the information in `query_store_wait_stats` and `query_store_runtime_stats` gets aggregated. It's not simply by `runtime_stats_interval_id` and `plan_id`. The `execution_type` also determines the aggregation because a given query may have an error or it could be canceled. This affects how the query behaves and the data is collected so it's included in the performance metrics to differentiate one set of behaviors from another. Let's see this by running the following script:

```
SELECT *
FROM sys.columns AS c,
     sys.syscolumns AS s;
```

That script results in a Cartesian join and takes about two minutes to run on my system. If we cancel the query while it's running once and let it complete once, we can then see what's in the Query Store.

```
SELECT qsqt.query_sql_text,
       qsrs.execution_type,
       qsrs.avg_duration
FROM sys.query_store_query AS qsq
    JOIN sys.query_store_query_text AS qsqt
        ON qsqt.query_text_id = qsq.query_text_id
    JOIN sys.query_store_plan AS qsp
        ON qsp.query_id = qsq.query_id
    JOIN sys.query_store_runtime_stats AS qsrs
        ON qsrs.plan_id = qsp.plan_id
WHERE qsqt.query_sql_text like '%FROM sys.columns AS c%';
```

You can see the results in Figure 11-8.



	query_sql_text	execution_type	avg_duration
1	SELECT * FROM sys.columns AS c, sys.sysco...	3	4800343
2	SELECT * FROM sys.columns AS c, sys.sysco...	0	122735336

Figure 11-8. *Aborted execution shown as different execution type*

www.EBooksWorld.ir

You'll see aborted queries and queries that had errors showing different types. Also, their durations, waits, and so on, within the runtime metrics are stored separately. To get a proper set of waits and duration measures from the two respective tables, you must include the execution_type.

If you were interested in all the query metrics for a given query, you could retrieve the information from the Query Store with something like this:

```
WITH QSAggregate
AS (SELECT qsrs.plan_id,
           SUM(qsrs.count_executions) AS CountExecutions,
           AVG(qsrs.avg_duration) AS AvgDuration,
           AVG(qsrs.stdev_duration) AS StdDevDuration,
           qsws.wait_category_desc,
           AVG(qsws.avg_query_wait_time_ms) AS AvgWaitTime,
           AVG(qsws.stdev_query_wait_time_ms) AS StDevWaitTime
    FROM sys.query_store_runtime_stats AS qsrs
        JOIN sys.query_store_wait_stats AS qsws
            ON qsws.plan_id = qsrs.plan_id
               AND qsws.runtime_stats_interval_id = qsrs.runtime_stats_
interval_id
    GROUP BY qsrs.plan_id,
             qsws.wait_category_desc)
SELECT CAST(qsp.query_plan AS XML),
       qsa.*
FROM sys.query_store_plan AS qsp
    JOIN QSAggregate AS qsa
        ON qsa.plan_id = qsp.plan_id
WHERE qsq.object_id = OBJECT_ID('dbo.
ProductTransactionHistoryByReference');
```

The results of this query will be all the information currently contained within the Query Store for the plan_id specified. You can combine the information within the Query Store in any way you need going forward. Next, let's take control of the Query Store.

www.EBooksWorld.ir

Controlling the Query Store

You've already see how to enable the Query Store for a database. To disable the Query Store, similar actions will work.

```
ALTER DATABASE AdventureWorks2017 SET QUERY_STORE = OFF;
```

This command will disable the Query Store, but it won't remove the Query Store information. That data collected and managed by the Query Store will persist through reboots, failovers, backups, and the database going offline. It will even persist beyond disabling the Query Store. To remove the Query Store data, you have to take direct control like this:

```
ALTER DATABASE AdventureWorks2017 SET QUERY_STORE CLEAR;
```

That will remove all data from the Query Store. You can get more selective if you want. You can simply remove a given query.

```
EXEC sys.sp_query_store_remove_query
    @query_id = @queryid;
```

You can remove a query plan.

```
EXEC sys.sp_query_store_remove_plan @plan_id = @PlanID;
```

You can also reset the performance metrics.

```
EXEC sys.sp_query_store_reset_exec_stats
    @plan_id = @PlanID;
```

All these simply require that you track down the plan or query in which you're interested in taking control of, and then you can do so. You may also find that you want to preserve the data in the Query Store that has been written to cache but not yet written to disk. You can force a flush of the cache.

```
EXEC sys.sp_query_store_flush_db;
```

Finally, you can change the default settings within the Query Store. First, it's a good idea to know where to go to get that information. You retrieve the current settings on the Query Store on a per-database basis by running the following:

```
SELECT * FROM sys.database_query_store_options AS dqso;
```


www.EBooksWorld.ir

As with so many other aspects of the Query Store, these settings are controlled on a per-database level. This enables you to, for example, change the statistics aggregation time interval on one database and not another. Controlling the various aspects of the Query Store settings is simply a matter of running this query:

```
ALTER DATABASE AdventureWorks2017 SET QUERY_STORE (MAX_STORAGE_SIZE_MB = 200);
```

That command changes the default storage size of the Query Store from 100MB to 200MB, allowing for more space in the database that was altered. When making these changes, no reboots of the server are required. You also won't affect the behavior of plans in the plan cache or any other part of the query processing within the database you are modifying. The default settings should be adequate for most people in most situations. Depending on your circumstances, you may want to modify the manner in which the Query Store behaves. Be sure that you monitor your servers when you make these changes to ensure that you haven't negatively impacted the server.

The only setting that I suggest you consider changing out of the box is the Query Store Capture Mode. By default, it captures all queries and all query plans, regardless of how often they are called, how long they run, or any other settings. For many of us, this behavior is adequate. However, if you have changed your system settings to use Optimize for Ad Hoc, you've done this because you get a lot of ad hoc queries and you're trying to manage memory use (more on this in Chapter 16). That setting means you're less interested in capturing every single plan. You may also be in the situation where because of the volume of transactions, you simply don't want to capture every single query or plan. These situations may lead you to change the Query Store Capture Mode setting. The other options are None and Auto. None will stop the Query Store from capturing queries and metrics but still allow for plan forcing if you set that for any queries (you'll find details on plan forcing later in this chapter). Auto will only capture queries that run for a certain length of time, consume a certain amount of resources, or get called a certain number of times. These values are all subject to change from Microsoft and are controlled internally within the Query Store. You can't control the values here, only whether they get used. On most systems, just to help reduce the noise and overhead, I recommend changing from All to Auto. However, this is absolutely an individual decision, and your situation may dictate otherwise.

You have the ability to take control of the Query Store using the SQL Server Management Studio GUI as well. Right-click any database in the Object Explorer window and from the context menu select Properties. When the Properties window opens, you can click the pane for Query Store and should see something similar to Figure 11-9.

www.EBooksWorld.ir

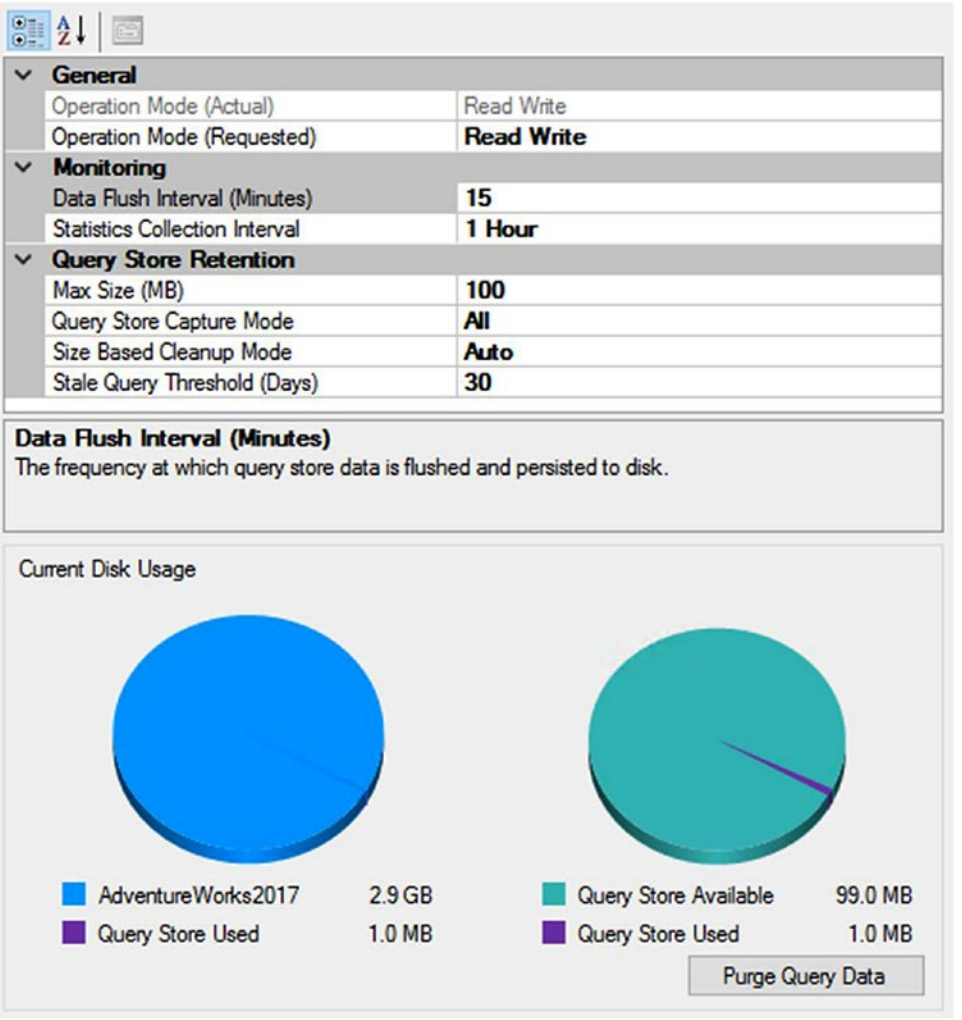


Figure 11-9. *SSMS GUI for controlling the Query Store*

Immediately you can see some of the settings that we've already covered in our exploration of the Query Store within this chapter. You also get to see just how much data the Query Store is using and how much room is left in the allocated space. As with using the T-SQL command shown earlier, any changes made here are immediately reflected in the Query Store behavior and will not require any sort of reboot of the system.

www.EBooksWorld.ir

Query Store Reporting

For some of your work, using T-SQL to take direct control over the Query Store and using the system tables to retrieve data about the Query Store is going to be the preferred approach. However, for a healthy percentage of the work, we can take advantage of the built-in reports and their behavior when working with the Query Store.

To see these reports, you just have to expand the database within the Object Explorer window in Management Studio. For any database with the Query Store enabled, there will be a new folder with the reports visible, as shown in Figure 11-10.

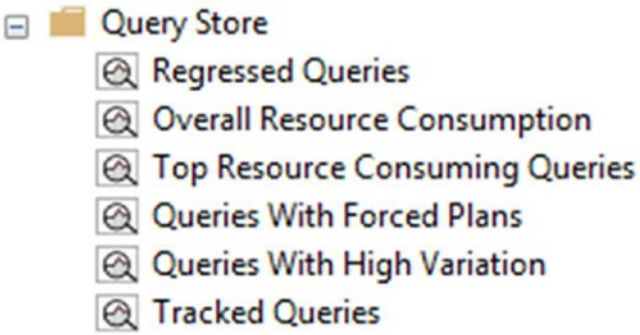


Figure 11-10. *Query Store reports within the AdventureWorks2017 database*

The reports are as follows:

- *Regressed Queries*: You'll see queries that have changed their performance in a negative way over time.
- *Overall Resource Consumption*: This report shows the resource consumption by various queries across a defined time frame. The default is the last month.
- *Top Resource Consuming Queries*: Here you find the queries that are using the most resources, without regard to a timeframe.
- *Query With Forced Plans*: Any queries that you have defined to have a forced plan will be visible in this report.

www.EBooksWorld.ir

- *Queries With High Variation*: This report displays queries that have a high degree of variation in their runtime statistics, frequently with more than one execution plan.
- *Tracked Queries*: With the Query Store, you can define a query as being of interest and instead of having to attempt to track it down in the other reports, you can mark the query and find it here.

Each of these reports is unique, and each one is useful for differing purposes, but we don't have the time and space to explore them all in detail. Instead, I'll focus on the behavior of one, Top Resource Consuming Queries, because it generally represents the behavior of all the others and because it's one that you're likely to use fairly frequently. Opening the report, you'll see something similar to Figure 11-11.

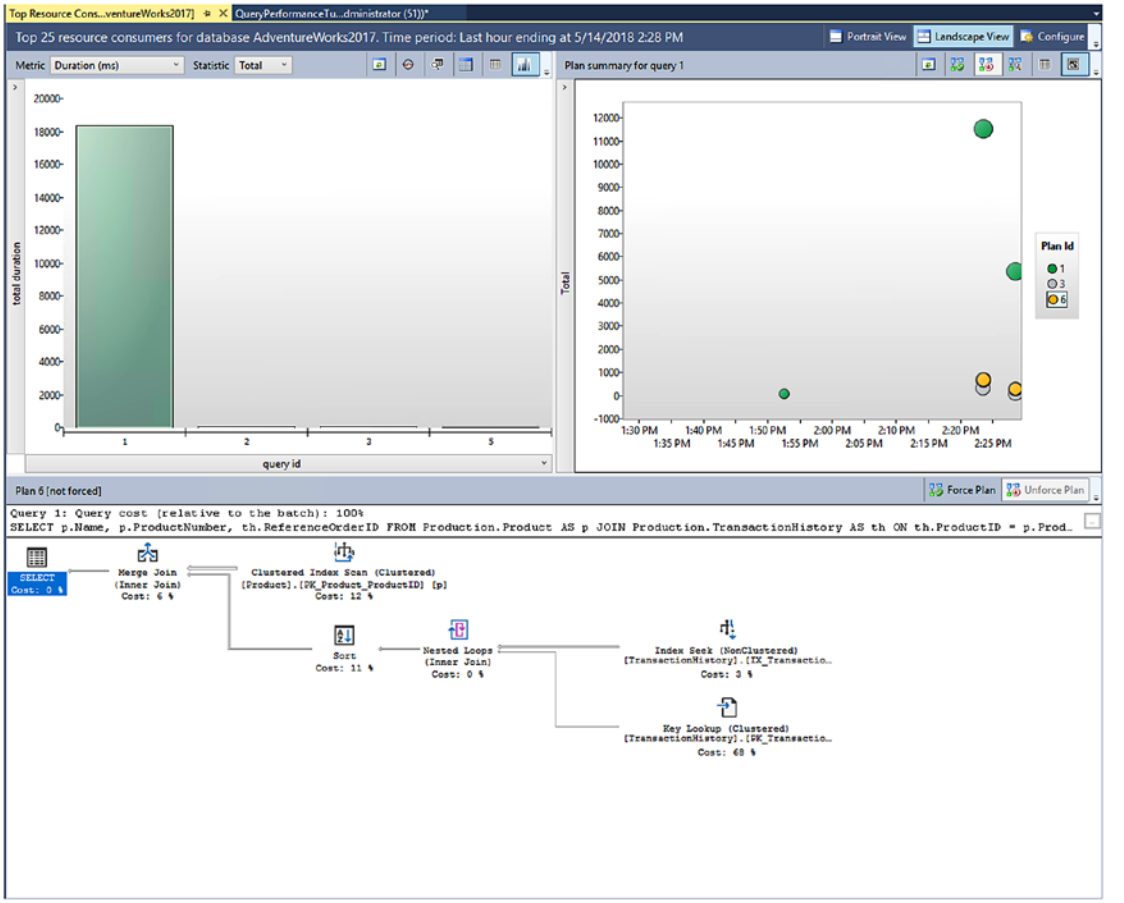


Figure 11-11. *Top 25 Resource Consumers report for the Query Store*

www.EBooksWorld.ir

There are three windows in the report. The first in the upper left shows queries, aggregated by the `query_id` value. The second window on the right shows the various query behaviors over time as well as the different plans for those queries. You can see that the number-one query, highlighted in the first pane, has three different execution plans. Clicking any one of those plans opens that plan in the third window on the bottom of the screen.

You're not limited to the default behavior. The first window, showing the queries aggregated by Duration by default, drives the other two. You have a drop-down at the top of the screen that gives you 13 choices currently, as shown in Figure 11-12.

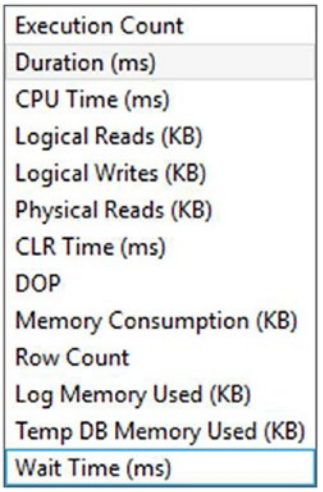


Figure 11-12. *Different aggregations for the Top 25 Resource Consumers report*

Selecting any one of them will change the values being aggregated for the report. You can also change how the report is aggregated using another drop-down. This list includes, average, minimum, maximum, total, and standard deviation. Additional functionality for the first window includes the ability to change to a grid format, mark a query for tracking later (in the Tracked Queries report), refresh the report, and look at the query text. All this is useful in attempting to identify the query to spend time with when you are working to determine performance issues.

www.EBooksWorld.ir

The next window shows the performance metrics from those selected in the first window. Each dot represents both a moment in time and a particular execution plan. The information in Figure 11-13 illustrates how query performance varied from 8:45 a.m. to 9:45 a.m. and how the query's performance and execution plans changed over that time frame.

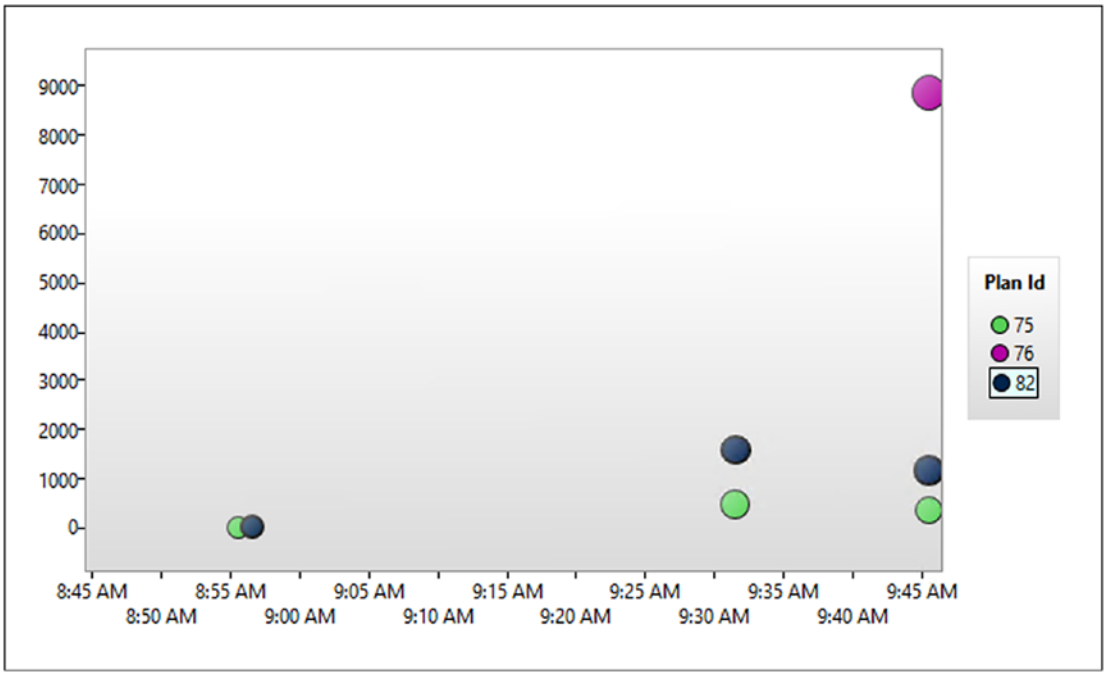


Figure 11-13. *Different performance behaviors and different execution plans for one query*

The size of each of the dots corresponds to the number of executions of the given plan within the given time frame. If you hover over any given dot, it will show you additional information about that moment in time. Figure 11-14 shows the information about the dot at the top of the screen, `plan_id` = 76, at the 9:45 a.m. time frame.

www.EBooksWorld.ir

Plan Id	76
Execution Type	Completed
Plan Forced	No
Interval Start	2017-12-08 09:45:00.000 -08:00
Interval End	2017-12-08 09:46:00.000 -08:00
Execution Count	63591
Total Duration (ms)	8871.63
Avg Duration (ms)	0.14
Min Duration (ms)	0.08
Max Duration (ms)	17.6
Std Dev Duration (ms)	0.25
Variation Duration (ms)	1.8

Figure 11-14. *Details of the information on display for a given plan*

You can see the number of executions and other metrics about that particular plan for the query in question. Whichever dot you click, you'll see the execution plan for that dot in the final window. The execution plans shown function like any other graphical plan within Management Studio, so I won't detail the behavior here. One additional piece of functionality that is on display here is the ability to force a plan. You'll see two buttons in the upper right of the execution plan window, as shown in Figure 11-15.

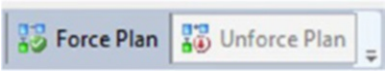


Figure 11-15. *Forcing and unforcing plans from the reports*

You have the ability to force, or unforce, a plan directly from the report. I'll cover plan forcing in detail in the next section.

Plan Forcing

While the majority of the functionality around the Query Store is all about collecting and observing the behavior of the queries and the query plans, one piece of functionality changes all that, plan forcing. *Plan forcing* is where you mark a particular plan as being the plan you would like SQL Server to use. Since everything within the Query Store is

www.EBooksWorld.ir

written to the database and so survives reboots, backups, and so on, this means you can ensure that a given plan will always be used. This process does change somewhat how the Query Store interacts with the optimization process and the plan cache, as illustrated in Figure 11-16.

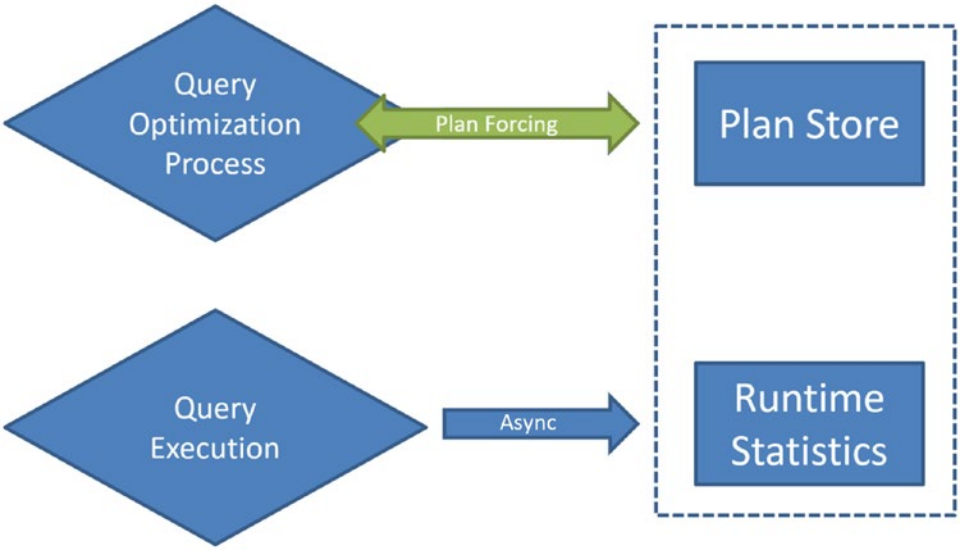


Figure 11-16. *The query optimization process with plan forcing added*

What happens now is that if a plan has been marked as being forced, when the optimizer completes its process, before it stores the plan in cache for use with the query, it first checks with the plans in the Query Store. If this query has a forced plan, that plan will always be used instead. The only exception to this is if something has changed internally in the system to make that plan an invalid plan for the query.

The function of plan forcing is actually quite simple. You have to supply a `plan_id` and a `query_id`, and you can force a plan. For example, my system has three possible plans for the query whose syntax, query hash, and query settings match the `query_id` value of 75. Note, while I'm using the `query_id` to mark a query, that's an artificial key. The identifying factors of a query are the text, the hash, and the context settings. The query to force a plan is then extremely simple.

```
EXEC sys.sp_query_store_force_plan 75,82;
```


www.EBooksWorld.ir

That is all that is required. From this point forward, no matter if a query is recompiled or removed from cache, when the optimization process is complete, the plan that corresponds to the `plan_id` of 82 will be used. With this in place, we can look at the Queries with Forced Plans report to see what gets displayed, as shown in Figure 11-17.

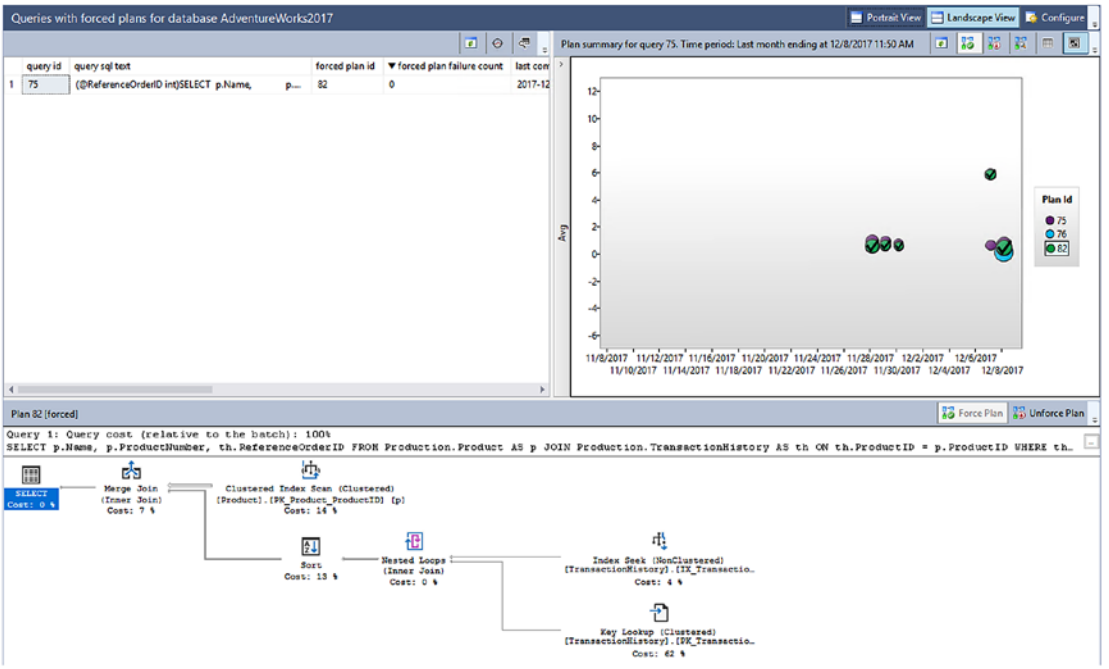


Figure 11-17. *Queries with Forced Plans report*

You can see that while overall this report is the same as the Top Resource Consuming Queries report, there are differences. The listing of queries in the first window is just that, a listing of the queries. The second window corresponds almost exactly with the previous window on display in Figures 11-11 and 11-13. However, the difference is, the plan that has been marked as being forced has a check mark in place. The final window is the same with one minor difference. At the top, instead of Force Plan being enabled, Unforce Plan is. You can easily unforce the plan from here by clicking that button. You can also unforce a plan with a single command.

```
EXEC sys.sp_query_store_unforce_plan 214,248;
```

Just as with clicking the button, this will stop the plan forcing. From this point forward, the optimization process goes back to normal. I'm going to save a full demonstration of plan forcing until we get to Chapter 17 when we talk about parameter

www.EBooksWorld.ir

sniffing. Suffice to say that plan forcing becomes extremely useful when dealing with bad parameter sniffing. It also is handy when dealing with regressions, situations where the changes to SQL Server cause previously well-behaving queries to suddenly generate badly performing execution plans. This most often occurs during an upgrade when the compatibility mode gets changed without testing.

Query Store for Upgrades

While general query performance monitoring and tuning may be a day-to-day common use for the Query Store, one of the most powerful purposes behind the tool is its use as a safety net for upgrading SQL Server.

Let's assume you are planning to migrate from SQL Server 2012 to SQL Server 2017. Traditionally you would upgrade your database on a test instance somewhere and then run a battery of tests to ensure that things are going to work well. If you catch and document all the issues, great. Unfortunately, it might require some code rewrites because of some of the changes to the optimizer or the cardinality estimator. That could cause delays to the upgrade, or the business might even decide to try to avoid it altogether (a frequent, if poor, choice). That assumes you catch the issues. It's entirely possible to miss that a particular query has suddenly started behaving poorly because of changes in estimated row counts or something else.

This is where the Query Store becomes your safety net for upgrades. First, you should do all the testing and attempt to address issues using standard methods. That shouldn't change. However, the Query Store adds additional functionality to the standard methods. Here are the steps to follow:

1. Restore your database to the new SQL Server instance or upgrade your instance. This assumes the production machine, but you can do this with a test machine as well.
2. Leave the database in the older compatibility mode. Do not change it to the new mode because you will enable both the new optimizer and the new cardinality estimator before you capture data.
3. Enable the Query Store. It can gather metrics running in the old compatibility mode.

www.EBooksWorld.ir