Relational data warehouses are at the center of most enterprise business intelligence (BI) solutions. While the specific details may vary across data warehouse implementations, a common pattern based on a denormalized, multidimensional schema has emerged as the standard design for a relational data warehouse.

Azure Synapse Analytics includes a highly scalable relational database engine that is optimized for data warehousing workloads. By using *dedicated SQL pools* in Azure Synapse Analytics, you can create databases that are capable of hosting and querying huge volumes of data in relational tables.

In this module, you'll learn how to:

- Design a schema for a relational data warehouse.
- Create fact, dimension, and staging tables.
- Use SQL to load data into data warehouse tables.
- Use SQL to query relational data warehouse tables.

Next unit: Design a data warehouse schema

Continue >

---

100 XP

Design a data warehouse schema

7 minutes

Like all relational databases, a data warehouse contains tables in which the data you want to analyze is stored. Most commonly, these tables are organized in a schema that is optimized for multidimensional modeling, in which numerical measures associated with events known as *facts* can be aggregated by the attributes of associated entities across multiple *dimensions*. For example, measures associated with a sales order (such as the amount paid or the quantity of items ordered) can be aggregated by attributes of the date on which the sale occurred, the customer, the store, and so on.

Tables in a data warehouse

A common pattern for relational data warehouses is to define a schema that includes two kinds of table: *dimension* tables and *fact* tables.

Dimension tables

Dimension tables describe business entities, such as products, people, places, and dates. Dimension tables contain columns for attributes of an entity. For example, a customer entity might have a first name, a last name, an email address, and a postal address (which might consist of a street address, a city, a postal code, and a country or region). In addition to attribute columns, a dimension table contains a unique key column that uniquely identifies each row in the table. In fact, it's common for a dimension table to include ***two*** key columns:

- a *surrogate* key that is specific to the data warehouse and uniquely identifies each row in the dimension table in the data warehouse - usually an incrementing integer number.
- An *alternate* key, often a *natural* or *business* key that is used to identify a specific instance of an entity in the transactional source system from which the entity record originated - such as a product code or a customer ID.

> **Note**
>
> Why have two keys? There are a few good reasons:
>
> - The data warehouse may be populated with data from multiple source systems, which can lead to the risk of duplicate or incompatible business keys.
> - Simple numeric keys generally perform better in queries that join lots of tables - a common pattern in data warehouses.
> - Attributes of entities may change over time - for example, a customer might change their address. Since the data warehouse is used to support historic reporting, you may want to retain a record for each instance of an entity at multiple points in time; so that, for example, sales orders for a specific customer are counted for the city where they lived at the time the order was placed. In this case, multiple customer records would have the same business key associated with the customer, but different surrogate keys for each discrete address where the customer lived at various times.

An example of a dimension table for customer might contain the following data:

CustomerKey	CustomerAltKey	Name	Email	Street	City	PostalCode	CountryRegion
123	I-543	Navin Jones	navin1@contoso.com	1 Main St.	Seattle	90000	United States
124	R-589	Mary Smith	mary2@contoso.com	234 190th Ave	Buffalo	50001	United States
125	I-321	Antoine Dubois	antoine1@contoso.com	2 Rue Jolie	Paris	20098	France
126	I-543	Navin Jones	navin1@contoso.com	24 125th Ave.	New York	50000	United States
...	...	...	...	...	...	...	...

> **Note**

Observe that the table contains two records for *Navin Jones*. Both records use the same alternate key to identify this person (*I-543*), but each record has a different surrogate key. From this, you can surmise that the customer moved from Seattle to New York. Sales made to the customer while living in Seattle are associated with the key *123*, while purchases made after moving to New York are recorded against record *126*.

In addition to dimension tables that represent business entities, it's common for a data warehouse to include a dimension table that represents *time*. This table enables data analysts to aggregate data over temporal intervals. Depending on the type of data you need to analyze, the lowest granularity (referred to as the *grain*) of a time dimension could represent times (to the hour, second, millisecond, nanosecond, or even lower), or dates.

An example of a time dimension table with a grain at the date level might contain the following data:

DateKey	DateAltKey	DayOfWeek	DayOfMonth	Weekday	Month	MonthName	Quarter	Year
19990101	01-01-1999	6	1	Friday	1	January	1	1999
...	...	...	...	...	...	...	...	...
20220101	01-01-2022	7	1	Saturday	1	January	1	2022
20220102	02-01-2022	1	2	Sunday	1	January	1	2022
...	...	...	...	...	...	...	...	...
20301231	31-12-2030	3	31	Tuesday	12	December	4	2030

The timespan covered by the records in the table must include the earliest and latest points in time for any associated events recorded in a related fact table. Usually there's a record for every interval at the appropriate grain in between.

Fact tables

Fact tables store details of observations or events; for example, sales orders, stock balances, exchange rates, or recorded temperatures. A fact table contains columns for numeric values that can be aggregated by dimensions. In addition to the numeric columns, a fact table contains key columns that reference unique keys in related dimension tables.

For example, a fact table containing details of sales orders might contain the following data:

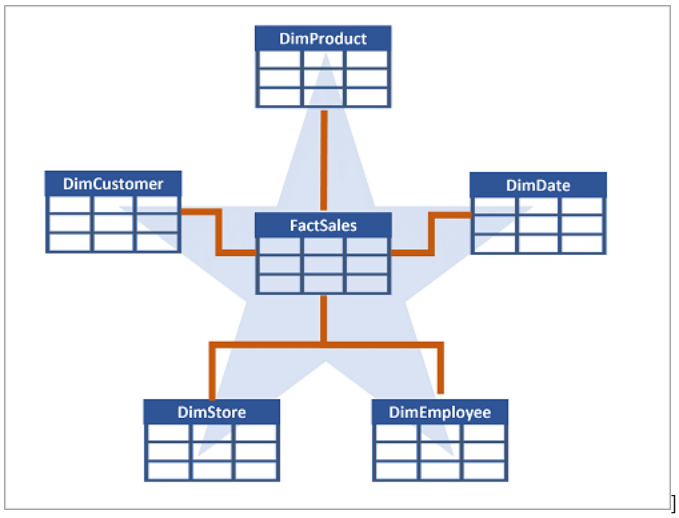
OrderDateKey	CustomerKey	StoreKey	ProductKey	OrderNo	LineItemNo	Quantity	UnitPrice	Tax	ItemTotal
20220101	123	5	701	1001	1	2	2.50	0.50	5.50
20220101	123	5	765	1001	2	1	2.00	0.20	2.20
20220102	125	2	723	1002	1	1	4.99	0.49	5.48
20220103	126	1	823	1003	1	1	7.99	0.80	8.79
...	...	...	...	...	...	...	...		...

A fact table's dimension key columns determine its grain. For example, the sales orders fact table includes keys for dates, customers, stores, and products. An order might include multiple products, so the grain represents line items for individual products sold in stores to customers on specific days.

Data warehouse schema designs

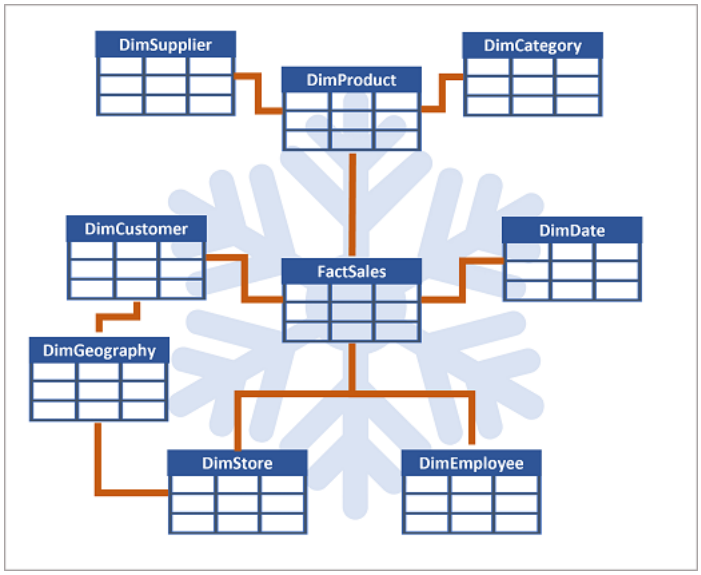
In most transactional databases that are used in business applications, the data is *normalized* to reduce duplication. In a data warehouse however, the dimension data is generally *de-normalized* to reduce the number of joins required to query the data.

Often, a data warehouse is organized as a *star* schema, in which a fact table is directly related to the dimension tables, as shown in this example:



]

The attributes of an entity can be used to aggregate measures in fact tables over multiple hierarchical levels - for example, to find total sales revenue by country or region, city, postal code, or individual customer. The attributes for each level can be stored in the same dimension table. However, when an entity has a large number of hierarchical attribute levels, or when some attributes can be shared by multiple dimensions (for example, both customers and stores have a geographical address), it can make sense to apply some normalization to the dimension tables and create a *snowflake* schema, as shown in the following example:



In this case, the **DimProduct** table has been normalized to create separate dimension tables for product categories and suppliers, and a **DimGeography** table has been added to represent geographical attributes for both customers and stores. Each row in the **DimProduct** table contains key values for the corresponding rows in the **DimCategory** and **DimSupplier** tables; and each row in the **DimCustomer** and **DimStore** tables contains a key value for the corresponding row in the **DimGeography** table.

Next unit: Create data warehouse tables

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

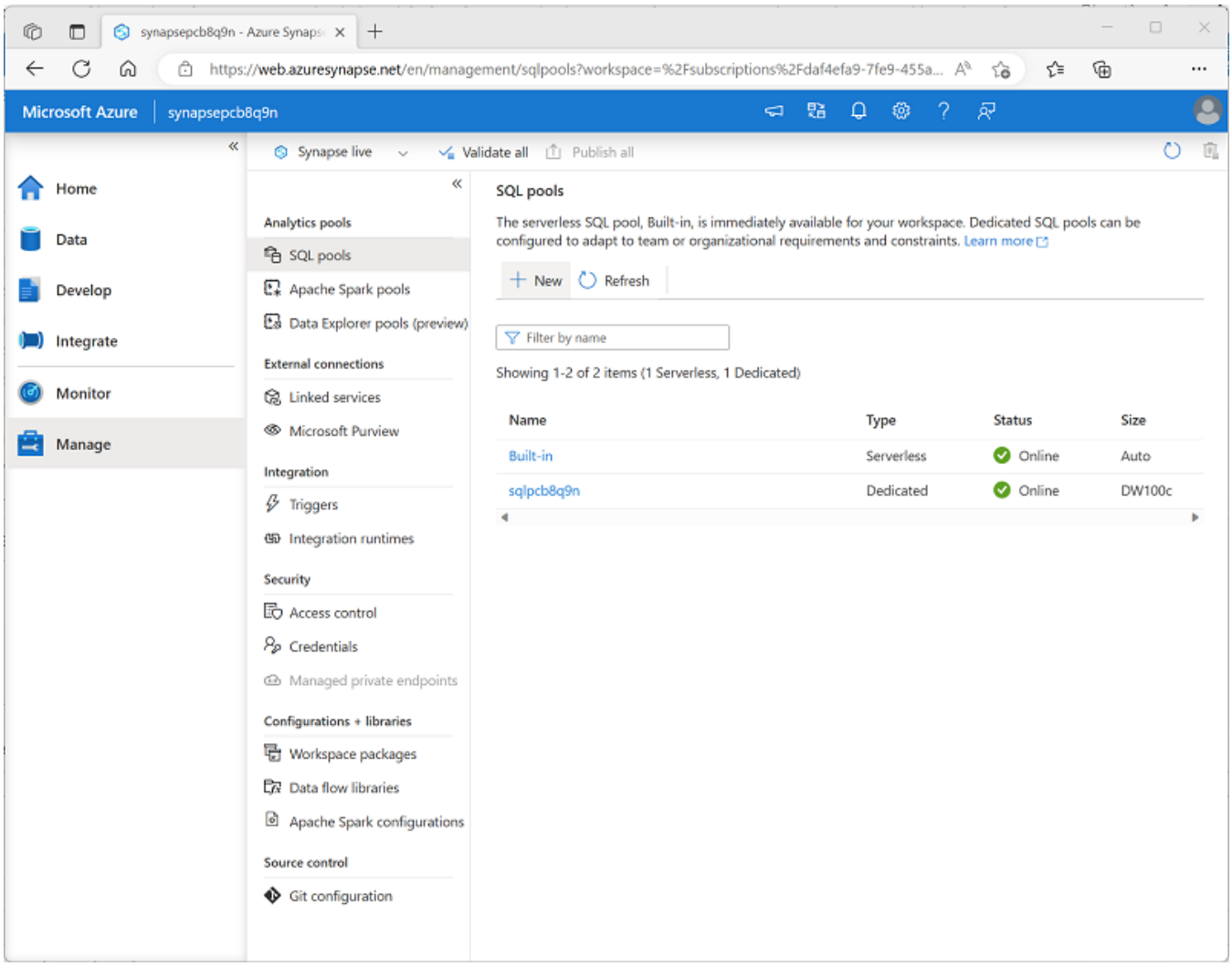
Create data warehouse tables

10 minutes

Now that you understand the basic architectural principles for a relational data warehouse schema, let's explore how to create a data warehouse.

Creating a dedicated SQL pool

To create a relational data warehouse in Azure Synapse Analytics, you must create a dedicated SQL Pool. The simplest way to do this in an existing Azure Synapse Analytics workspace is to use the **Manage** page in Azure Synapse Studio, as shown here:



When provisioning a dedicated SQL pool, you can specify the following configuration settings:

- A unique name for the dedicated SQL pool.

- A performance level for the SQL pool, which can range from *DW100c* to *DW30000c* and which determines the cost per hour for the pool when it's running.
- Whether to start with an empty pool or restore an existing database from a backup.
- The *collation* of the SQL pool, which determines sort order and string comparison rules for the database. (*You can't change the collation after creation*).

After creating a dedicated SQL pool, you can control its running state in the **Manage** page of Synapse Studio; pausing it when not required to prevent unnecessary costs.

When the pool is running, you can explore it on the **Data** page, and create SQL scripts to run in it.

Considerations for creating tables

To create tables in the dedicated SQL pool, you use the `CREATE TABLE` (or sometimes the `CREATE EXTERNAL TABLE`) Transact-SQL statement. The specific options used in the statement depend on the type of table you're creating, which can include:

- Fact tables
- Dimension tables
- Staging tables

> **Note**
>
> The data warehouse is composed of *fact* and *dimension* tables as discussed previously. *Staging* tables are often used as part of the data warehousing loading process to ingest data from source systems.

When designing a star schema model for small or medium sized datasets you can use your preferred database, such as Azure SQL. For larger data sets you may benefit from implementing your data warehouse in Azure Synapse Analytics instead of SQL Server. It's important to understand some key differences when creating tables in Synapse Analytics.

Data integrity constraints

Dedicated SQL pools in Synapse Analytics don't support *foreign key* and *unique* constraints as found in other relational database systems like SQL Server. This means that jobs used to load data must maintain uniqueness and referential integrity for keys, without relying on the table definitions in the database to do so.

> **Tip**

For more information about constraints in Azure Synapse Analytics dedicated SQL pools, see **Primary key, foreign key, and unique key using dedicated SQL pool in Azure Synapse Analytics**.

Indexes

While Synapse Analytics dedicated SQL pools support *clustered* indexes as found in SQL Server, the default index type is *clustered columnstore*. This index type offers a significant performance advantage when querying large quantities of data in a typical data warehouse schema and should be used where possible. However, some tables may include data types that can't be included in a clustered columnstore index (for example, VARBINARY(MAX)), in which case a clustered index can be used instead.

> **Tip**
>
> For more information about indexing in Azure Synapse Analytics dedicated SQL pools, see **Indexes on dedicated SQL pool tables in Azure Synapse Analytics**.

Distribution

Azure Synapse Analytics dedicated SQL pools use a massively parallel processing (MPP) architecture, as opposed to the symmetric multiprocessing (SMP) architecture used in most OLTP database systems. In an MPP system, the data in a table is distributed for processing across a pool of nodes. Synapse Analytics supports the following kinds of distribution:

- **Hash**: A deterministic hash value is calculated for the specified column and used to assign the row to a compute node.
- **Round-robin**: Rows are distributed evenly across all compute nodes.
- **Replicated**: A copy of the table is stored on each compute node.

The table type often determines which option to choose for distributing the table.

Table type	Recommended distribution option
Dimension	Use replicated distribution for smaller tables to avoid data shuffling when joining to distributed fact tables. If tables are too large to store on each compute node, use hash distribution.

Table type	Recommended distribution option
Fact	Use hash distribution with clustered columnstore index to distribute fact tables across compute nodes.
Staging	Use round-robin distribution for staging tables to evenly distribute data across compute nodes.

> **Tip**
>
> For more information about distribution strategies for tables in Azure Synapse Analytics, see **Guidance for designing distributed tables using dedicated SQL pool in Azure Synapse Analytics**.

Creating dimension tables

When you create a dimension table, ensure that the table definition includes surrogate and alternate keys as well as columns for the attributes of the dimension that you want to use to group aggregations. It's often easiest to use an `IDENTITY` column to auto-generate an incrementing surrogate key (otherwise you need to generate unique keys every time you load data). The following example shows a `CREATE TABLE` statement for a hypothetical **DimCustomer** dimension table.

SQL

```
CREATE TABLE dbo.DimCustomer
(
    CustomerKey INT IDENTITY NOT NULL,
    CustomerAlternateKey NVARCHAR(15) NULL,
    CustomerName NVARCHAR(80) NOT NULL,
    EmailAddress NVARCHAR(50) NULL,
    Phone NVARCHAR(25) NULL,
    StreetAddress NVARCHAR(100),
    City NVARCHAR(20),
    PostalCode NVARCHAR(10),
    CountryRegion NVARCHAR(20)
)
WITH
(
    DISTRIBUTION = REPLICATE,
    CLUSTERED COLUMNSTORE INDEX
);
```

> **Note**
>
> If desired, you can create a specific *schema* as a namespace for your tables. In this example, the default **dbo** schema is used.

If you intend to use a *snowflake* schema in which dimension tables are related to one another, you should include the key for the *parent* dimension in the definition of the *child* dimension table. For example, the following SQL code could be used to move the geographical address details from the **DimCustomer** table to a separate **DimGeography** dimension table:

SQL

```
CREATE TABLE dbo.DimGeography
(
    GeographyKey INT IDENTITY NOT NULL,
    GeographyAlternateKey NVARCHAR(10) NULL,
    StreetAddress NVARCHAR(100),
    City NVARCHAR(20),
    PostalCode NVARCHAR(10),
    CountryRegion NVARCHAR(20)
)
WITH
(
    DISTRIBUTION = REPLICATE,
    CLUSTERED COLUMNSTORE INDEX
);

CREATE TABLE dbo.DimCustomer
(
    CustomerKey INT IDENTITY NOT NULL,
    CustomerAlternateKey NVARCHAR(15) NULL,
    GeographyKey INT NULL,
    CustomerName NVARCHAR(80) NOT NULL,
    EmailAddress NVARCHAR(50) NULL,
    Phone NVARCHAR(25) NULL
)
WITH
(
    DISTRIBUTION = REPLICATE,
    CLUSTERED COLUMNSTORE INDEX
);
```

Time dimension tables

Most data warehouses include a *time* dimension table that enables you to aggregate data by multiple hierarchical levels of time interval. For example, the following example creates a **DimDate** table with attributes that relate to specific dates.

SQL

```
CREATE TABLE dbo.DimDate
(
    DateKey INT NOT NULL,
    DateAltKey DATETIME NOT NULL,
    DayOfMonth INT NOT NULL,
    DayOfWeek INT NOT NULL,
    DayName NVARCHAR(15) NOT NULL,
    MonthOfYear INT NOT NULL,
    MonthName NVARCHAR(15) NOT NULL,
    CalendarQuarter INT  NOT NULL,
    CalendarYear INT NOT NULL,
    FiscalQuarter INT NOT NULL,
    FiscalYear INT NOT NULL
)
WITH
(
    DISTRIBUTION = REPLICATE,
    CLUSTERED COLUMNSTORE INDEX
);
```

Tip

A common pattern when creating a dimension table for dates is to use the numeric date in *DDMMYYYY* or *YYYYMMDD* format as an integer surrogate key, and the date as a `DATE` or `DATETIME` datatype as the alternate key.

Creating fact tables

Fact tables include the keys for each dimension to which they're related, and the attributes and numeric measures for specific events or observations that you want to analyze.

The following code example creates a hypothetical fact table named **FactSales** that is related to multiple dimensions through key columns (date, customer, product, and store)

SQL

```
CREATE TABLE dbo.FactSales
(
    OrderDateKey INT NOT NULL,
    CustomerKey INT NOT NULL,
    ProductKey INT NOT NULL,
    StoreKey INT NOT NULL,
    OrderNumber NVARCHAR(10) NOT NULL,
    OrderLineItem INT NOT NULL,
    OrderQuantity SMALLINT NOT NULL,
```

```
    UnitPrice DECIMAL NOT NULL,
    Discount DECIMAL NOT NULL,
    Tax DECIMAL NOT NULL,
    SalesAmount DECIMAL NOT NULL
)
WITH
(
    DISTRIBUTION = HASH(OrderNumber),
    CLUSTERED COLUMNSTORE INDEX
);
```

Creating staging tables

Staging tables are used as temporary storage for data as it's being loaded into the data warehouse. A typical pattern is to structure the table to make it as efficient as possible to ingest the data from its external source (often files in a data lake) into the relational database, and then use SQL statements to load the data from the staging tables into the dimension and fact tables.

The following code example creates a staging table for product data that will ultimately be loaded into a dimension table:

SQL

```
CREATE TABLE dbo.StageProduct
(
    ProductID NVARCHAR(10) NOT NULL,
    ProductName NVARCHAR(200) NOT NULL,
    ProductCategory NVARCHAR(200) NOT NULL,
    Color NVARCHAR(10),
    Size NVARCHAR(10),
    ListPrice DECIMAL NOT NULL,
    Discontinued BIT NOT NULL
)
WITH
(
    DISTRIBUTION = ROUND_ROBIN,
    CLUSTERED COLUMNSTORE INDEX
);
```

Using external tables

In some cases, if the data to be loaded is in files with an appropriate structure, it can be more effective to create external tables that reference the file location. This way, the data can be read directly from the source files instead of being loaded into the relational store. The

following example, shows how to create an external table that references files in the data lake associated with the Synapse workspace:

SQL

```
-- External data source links to data lake location
CREATE EXTERNAL DATA SOURCE StagedFiles
WITH (
    LOCATION = 'https://mydatalake.blob.core.windows.net/data/stagedfiles/'
);
GO

-- External format specifies file format
CREATE EXTERNAL FILE FORMAT ParquetFormat
WITH (
    FORMAT_TYPE = PARQUET,
    DATA_COMPRESSION = 'org.apache.hadoop.io.compress.SnappyCodec'
);
GO

-- External table references files in external data source
CREATE EXTERNAL TABLE dbo.ExternalStageProduct
(
    ProductID NVARCHAR(10) NOT NULL,
    ProductName NVARCHAR(200) NOT NULL,
    ProductCategory NVARCHAR(200) NOT NULL,
    Color NVARCHAR(10),
    Size NVARCHAR(10),
    ListPrice DECIMAL NOT NULL,
    Discontinued BIT NOT NULL
)
WITH
(
    DATA_SOURCE = StagedFiles,
    LOCATION = 'products/*.parquet',
    FILE_FORMAT = ParquetFormat
);
GO
```

> **Note**
>
> For more information about using external tables, see **Use external tables with Synapse SQL** in the Azure Synapse Analytics documentation.

Next unit: Load data warehouse tables

100 XP

Load data warehouse tables

10 minutes

At a basic level, loading a data warehouse is typically achieved by adding new data from files in a data lake into tables in the data warehouse. The `COPY` statement is an effective way to accomplish this task, as shown in the following example:

SQL

```
COPY INTO dbo.StageProducts
    (ProductID, ProductName, ProductCategory, Color, Size, ListPrice,
Discontinued)
FROM 'https://mydatalake.blob.core.windows.net/data/stagedfiles/prod-
ucts/*.parquet'
WITH
(
    FILE_TYPE = 'PARQUET',
    MAXERRORS = 0,
    IDENTITY_INSERT = 'OFF'
);
```

Considerations for designing a data warehouse load process

One of the most common patterns for loading a data warehouse is to transfer data from source systems to files in a data lake, ingest the file data into staging tables, and then use SQL statements to load the data from the staging tables into the dimension and fact tables. Usually data loading is performed as a periodic batch process in which inserts and updates to the data warehouse are coordinated to occur at a regular interval (for example, daily, weekly, or monthly).

In most cases, you should implement a data warehouse load process that performs tasks in the following order:

1. Ingest the new data to be loaded into a data lake, applying pre-load cleansing or transformations as required.
2. Load the data from files into staging tables in the relational data warehouse.
3. Load the dimension tables from the dimension data in the staging tables, updating existing rows or inserting new rows and generating surrogate key values as necessary.

4. Load the fact tables from the fact data in the staging tables, looking up the appropriate surrogate keys for related dimensions.
5. Perform post-load optimization by updating indexes and table distribution statistics.

After using the `COPY` statement to load data into staging tables, you can use a combination of `INSERT`, `UPDATE`, `MERGE`, and `CREATE TABLE AS SELECT` (*CTAS*) statements to load the staged data into dimension and fact tables.

> **Note**
>
> Implementing an effective data warehouse loading solution requires careful consideration of how to manage surrogate keys, slowly changing dimensions, and other complexities inherent in a relational data warehouse schema. To learn more about techniques for loading a data warehouse, consider completing the **Load data into a relational data warehouse** module.

Next unit: Query a data warehouse

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Query a data warehouse

10 minutes

When the dimension and fact tables in a data warehouse have been loaded with data, you can use SQL to query the tables and analyze the data they contain. The Transact-SQL syntax used to query tables in a Synapse dedicated SQL pool is similar to SQL used in SQL Server or Azure SQL Database.

Aggregating measures by dimension attributes

Most data analytics with a data warehouse involves aggregating numeric measures in fact tables by attributes in dimension tables. Because of the way a star or snowflake schema is implemented, queries to perform this kind of aggregation rely on `JOIN` clauses to connect fact tables to dimension tables, and a combination of aggregate functions and `GROUP BY` clauses to define the aggregation hierarchies.

For example, the following SQL queries the **FactSales** and **DimDate** tables in a hypothetical data warehouse to aggregate sales amounts by year and quarter:

SQL

```
SELECT  dates.CalendarYear,
        dates.CalendarQuarter,
        SUM(sales.SalesAmount) AS TotalSales
FROM dbo.FactSales AS sales
JOIN dbo.DimDate AS dates ON sales.OrderDateKey = dates.DateKey
GROUP BY dates.CalendarYear, dates.CalendarQuarter
ORDER BY dates.CalendarYear, dates.CalendarQuarter;
```

The results from this query would look similar to the following table:

CalendarYear	CalendarQuarter	TotalSales
2020	1	25980.16
2020	2	27453.87
2020	3	28527.15
2020	4	31083.45
2021	1	34562.96
2021	2	36162.27

CalendarYear	CalendarQuarter	TotalSales
...	...	...

You can join as many dimension tables as needed to calculate the aggregations you need. For example, the following code extends the previous example to break down the quarterly sales totals by city based on the customer's address details in the **DimCustomer** table:

SQL

```
SELECT  dates.CalendarYear,
        dates.CalendarQuarter,
        custs.City,
        SUM(sales.SalesAmount) AS TotalSales
FROM dbo.FactSales AS sales
JOIN dbo.DimDate AS dates ON sales.OrderDateKey = dates.DateKey
JOIN dbo.DimCustomer AS custs ON sales.CustomerKey = custs.CustomerKey
GROUP BY dates.CalendarYear, dates.CalendarQuarter, custs.City
ORDER BY dates.CalendarYear, dates.CalendarQuarter, custs.City;
```

This time, the results include a quarterly sales total for each city:

CalendarYear	CalendarQuarter	City	TotalSales
2020	1	Amsterdam	5982.53
2020	1	Berlin	2826.98
2020	1	Chicago	5372.72
...	...	...	..
2020	2	Amsterdam	7163.93
2020	2	Berlin	8191.12
2020	2	Chicago	2428.72
...	...	...	..
2020	3	Amsterdam	7261.92
2020	3	Berlin	4202.65
2020	3	Chicago	2287.87

CalendarYear	CalendarQuarter	City	TotalSales
...	...	...	..
2020	4	Amsterdam	8262.73
2020	4	Berlin	5373.61
2020	4	Chicago	7726.23
...	...	...	..
2021	1	Amsterdam	7261.28
2021	1	Berlin	3648.28
2021	1	Chicago	1027.27
...	...	...	..

Joins in a snowflake schema

When using a snowflake schema, dimensions may be partially normalized; requiring multiple joins to relate fact tables to snowflake dimensions. For example, suppose your data warehouse includes a **DimProduct** dimension table from which the product categories have been normalized into a separate **DimCategory** table. A query to aggregate items sold by product category might look similar to the following example:

SQL

```
SELECT  cat.ProductCategory,
        SUM(sales.OrderQuantity) AS ItemsSold
FROM dbo.FactSales AS sales
JOIN dbo.DimProduct AS prod ON sales.ProductKey = prod.ProductKey
JOIN dbo.DimCategory AS cat ON prod.CategoryKey = cat.CategoryKey
GROUP BY cat.ProductCategory
ORDER BY cat.ProductCategory;
```

The results from this query include the number of items sold for each product category:

ProductCategory	ItemsSold
Accessories	28271
Bits and pieces	5368

ProductCategory	ItemsSold
...	...

> **Note**
>
> JOIN clauses for **FactSales** and **DimProduct** and for **DimProduct** and **DimCategory** are both required, even though no fields from **DimProduct** are returned by the query.

Using ranking functions

Another common kind of analytical query is to partition the results based on a dimension attribute and *rank* the results within each partition. For example, you might want to rank stores each year by their sales revenue. To accomplish this goal, you can use Transact-SQL *ranking* functions such as `ROW_NUMBER`, `RANK`, `DENSE_RANK`, and `NTILE`. These functions enable you to partition the data over categories, each returning a specific value that indicates the relative position of each row within the partition:

- **ROW_NUMBER** returns the ordinal position of the row within the partition. For example, the first row is numbered 1, the second 2, and so on.
- **RANK** returns the ranked position of each row in the ordered results. For example, in a partition of stores ordered by sales volume, the store with the highest sales volume is ranked 1. If multiple stores have the same sales volumes, they'll be ranked the same, and the rank assigned to subsequent stores reflects the number of stores that have higher sales volumes - including ties.
- **DENSE_RANK** ranks rows in a partition the same way as **RANK**, but when multiple rows have the same rank, subsequent rows are ranking positions ignore ties.
- **NTILE** returns the specified percentile in which the row falls. For example, in a partition of stores ordered by sales volume, `NTILE(4)` returns the quartile in which a store's sales volume places it.

For example, consider the following query:

SQL

```
SELECT  ProductCategory,
        ProductName,
        ListPrice,
        ROW_NUMBER() OVER
            (PARTITION BY ProductCategory ORDER BY ListPrice DESC) AS RowNumber,
        RANK() OVER
            (PARTITION BY ProductCategory ORDER BY ListPrice DESC) AS Rank,
        DENSE_RANK() OVER
            (PARTITION BY ProductCategory ORDER BY ListPrice DESC) AS DenseRank,
        NTILE(4) OVER
            (PARTITION BY ProductCategory ORDER BY ListPrice DESC) AS Quartile
FROM dbo.DimProduct
ORDER BY ProductCategory;
```

The query partitions products into groupings based on their categories, and within each category partition, the relative position of each product is determined based on its list price. The results from this query might look similar to the following table:

ProductCategory	ProductName	ListPrice	RowNumber	Rank	DenseRank	Quartile
Accessories	Widget	8.99	1	1	1	1
Accessories	Knicknak	8.49	2	2	2	1
Accessories	Sprocket	5.99	3	3	3	2
Accessories	Doodah	5.99	4	3	3	2
Accessories	Spangle	2.99	5	5	4	3
Accessories	Badabing	0.25	6	6	5	4
Bits and pieces	Flimflam	7.49	1	1	1	1
Bits and pieces	Snickity wotsit	6.99	2	2	2	1
Bits and pieces	Flange	4.25	3	3	3	2
...	...	...	...	...	...	...

> **Note**
>
> The sample results demonstrate the difference between `RANK` and `DENSE_RANK`. Note that in the *Accessories* category, the *Sprocket* and *Doodah* products have the same list price; and are both ranked as the 3rd highest priced product. The next highest priced product has a *RANK* of 5 (there are four products more expensive than it) and a *DENSE_RANK* of 4 (there are three higher prices).
>
> To learn more about ranking functions, see **Ranking Functions (Transact-SQL)** in the Azure Synapse Analytics documentation.

Retrieving an approximate count

While the purpose of a data warehouse is primarily to support analytical data models and reports for the enterprise; data analysts and data scientists often need to perform some initial data exploration, just to determine the basic scale and distribution of the data.

For example, the following query uses the `COUNT` function to retrieve the number of sales for each year in a hypothetical data warehouse:

SQL

```
SELECT dates.CalendarYear AS CalendarYear,
    COUNT(DISTINCT sales.OrderNumber) AS Orders
FROM FactSales AS sales
JOIN DimDate AS dates ON sales.OrderDateKey = dates.DateKey
GROUP BY dates.CalendarYear
ORDER BY CalendarYear;
```

The results of this query might look similar to the following table:

CalendarYear	Orders
2019	239870
2020	284741
2021	309272
...	...

The volume of data in a data warehouse can mean that even simple queries to count the number of records that meet specified criteria can take a considerable time to run. In many cases, a precise count isn't required - an approximate estimate will suffice. In such cases, you can use the `APPROX_COUNT_DISTINCT` function as shown in the following example:

SQL

```
SELECT dates.CalendarYear AS CalendarYear,
    APPROX_COUNT_DISTINCT(sales.OrderNumber) AS ApproxOrders
FROM FactSales AS sales
JOIN DimDate AS dates ON sales.OrderDateKey = dates.DateKey
GROUP BY dates.CalendarYear
ORDER BY CalendarYear;
```

The `APPROX_COUNT_DISTINCT` function uses a *HyperLogLog* algorithm to retrieve an approximate count. The result is guaranteed to have a maximum error rate of 2% with 97% probability, so the results of this query with the same hypothetical data as before might look similar to the following table:

CalendarYear	ApproxOrders
2019	235552
2020	290436

CalendarYear	ApproxOrders
2021	304633
...	...

The counts are less accurate, but still sufficient for an approximate comparison of yearly sales. With a large volume of data, the query using the `APPROX_COUNT_DISTINCT` function completes more quickly, and the reduced accuracy may be an acceptable trade-off during basic data exploration.

> **Note**
>
> See the **APPROX_COUNT_DISTINCT** function documentation for more details.

Next unit: Exercise - Explore a data warehouse

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

200 XP

Knowledge check

5 minutes

1. In which of the following table types should an insurance company store details of customer attributes by which claims will be aggregated? *

- Staging table
- Dimension table

  ✔ Correct. Attributes of an entity by which numeric measures will be aggregated are stored in a dimension table.
- Fact table

2. You create a dimension table for product data, assigning a unique numeric key for each row in a column named `ProductKey`. The `ProductKey` is only defined in the data warehouse. What kind of key is `ProductKey`? *

- A surrogate key

  ✔ Correct. A surrogate key uniquely identifies each row in a dimension table, irrespective of keys used in source systems.
- An alternate key
- A business key

3. What distribution option would be best for a sales fact table that will contain billions of records? *

- HASH

  ✔ Correct. Hash distribution provides good read performance for a large table by distributing records across compute nodes based on the hash key.
- ROUND_ROBIN

✗ Incorrect. Round robin distribution distributes the data evenly but does not optimize queries on commonly used distribution key fields.

- REPLICATE

4. You need to write a query to return the total of the **`UnitsProduced`** numeric measure in the **`FactProduction`** table aggregated by the **`ProductName`** attribute in the **`FactProduct`** table. Both tables include a **`ProductKey`** surrogate key field. What should you do? *

- Use two `SELECT` queries with a `UNION ALL` clause to combine the rows in the `FactProduction` table with those in the `FactProduct` table.
- Use a `SELECT` query against the `FactProduction` table with a `WHERE` clause to filter out rows with a `ProductKey` that doesn't exist in the `FactProduct` table.
- Use a `SELECT` query with a `SUM` function to total the `UnitsProduced` metric, using a `JOIN` on the `ProductKey` surrogate key to match the `FactProduction` records to the `FactProduct` records and a `GROUP BY` clause to aggregate by `ProductName`.

✓ Correct. To aggregate measures in a fact table by attributes in a dimension table, include an aggregate function for the measure, join the tables on the surrogate key, and group the results by the appropriate attributes.

5. You use the **`RANK`** function in a query to rank customers in order of the number of purchases they have made. Five customers have made the same number of purchases and are all ranked equally as 1. What rank will the customer with the next highest number of purchases be assigned? *

- two
- six

✓ Correct. There are five customers with a higher number of purchases, and **`RANK`** takes these into account.

- one

6. You need to compare approximate production volumes by product while optimizing query response time. Which function should you use? *

- COUNT

✗ Incorrect. The `COUNT` function will return accurate counts, but may take longer than other options.

- ○ NTILE
- ○ APPROX_COUNT_DISTINCT

✓ Correct. `APPROX_COUNT_DISTINCT` returns an approximate count within 2% of the actual count while optimizing for minimal response time.

Next unit: Summary

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Introduction

1 minute

Many enterprise analytical solutions include a relational data warehouse. Data engineers are responsible for implementing ingestion solutions that load data into the data warehouse tables, usually on a regular schedule.

As a data engineer, you need to be familiar with the considerations and techniques that apply to loading a data warehouse. In this module, we'll focus on ways that you can use SQL to load data into tables in a dedicated SQL pool in Azure Synapse Analytics.

Next unit: Load staging tables

Continue

100 XP

Load staging tables

5 minutes

One of the most common patterns for loading a data warehouse is to transfer data from source systems to files in a data lake, ingest the file data into staging tables, and then use SQL statements to load the data from the staging tables into the dimension and fact tables. Usually data loading is performed as a periodic batch process in which inserts and updates to the data warehouse are coordinated to occur at a regular interval (for example, daily, weekly, or monthly).

Creating staging tables

Many organized warehouses have standard structures for staging the database and may even use a specific schema for staging the data. The following code example creates a staging table for product data that will ultimately be loaded into a dimension table:

> **Note**
>
> This example creates a staging table in the default **dbo** schema. You can also create separate schemas for staging tables with a meaningful name, such as **stage** so architects and users understand the purpose of the schema.

SQL

```
CREATE TABLE dbo.StageProduct
(
    ProductID NVARCHAR(10) NOT NULL,
    ProductName NVARCHAR(200) NOT NULL,
    ProductCategory NVARCHAR(200) NOT NULL,
    Color NVARCHAR(10),
    Size NVARCHAR(10),
    ListPrice DECIMAL NOT NULL,
    Discontinued BIT NOT NULL
)
WITH
(
    DISTRIBUTION = ROUND_ROBIN,
    CLUSTERED COLUMNSTORE INDEX
);
```

Using the COPY command

You can use the COPY statement to load data from the data lake, as shown in the following example:

> **Note**
>
> This is generally the recommended approach to load staging tables due to its high performance throughput.

SQL

```
COPY INTO dbo.StageProduct
    (ProductID, ProductName, ...)
FROM 'https://mydatalake.../data/products*.parquet'
WITH
(
    FILE_TYPE = 'PARQUET',
    MAXERRORS = 0,
    IDENTITY_INSERT = 'OFF'
);
```

> **Tip**
>
> To learn more about the COPY statement, see **COPY (Transact-SQL)** in the Transact-SQL documentation.

Using external tables

In some cases, if the data to be loaded is stored in files with an appropriate structure, it can be more effective to create external tables that reference the file location. This way, the data can be read directly from the source files instead of being loaded into the relational store. The following example, shows how to create an external table that references files in the data lake associated with the Azure Synapse Analytics workspace:

SQL

```
CREATE EXTERNAL TABLE dbo.ExternalStageProduct
 (
     ProductID NVARCHAR(10) NOT NULL,
     ProductName NVARCHAR(10) NOT NULL,
 ...
 )
```

```
WITH
 (
    DATE_SOURCE = StagedFiles,
    LOCATION = 'folder_name/*.parquet',
    FILE_FORMAT = ParquetFormat
 );
GO
```

Tip

For more information about using external tables, see **Use external tables with Synapse SQL** in the Azure Synapse Analytics documentation.

Next unit: Load dimension tables

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Load dimension tables

5 minutes

After staging dimension data, you can load it into dimension tables using SQL.

Using a CREATE TABLE AS (CTAS) statement

One of the simplest ways to load data into a new dimension table is to use a `CREATE TABLE AS` (*CTAS*) expression. This statement creates a new table based on the results of a SELECT statement.

SQL

```
CREATE TABLE dbo.DimProduct
WITH
(
    DISTRIBUTION = REPLICATE,
    CLUSTERED COLUMNSTORE INDEX
)
AS
SELECT ROW_NUMBER() OVER(ORDER BY ProdID) AS ProdKey,
    ProdID as ProdAltKey,
    ProductName,
    ProductCategory,
    Color,
    Size,
    ListPrice,
    Discontinued
FROM dbo.StageProduct;
```

Note

You can't use `IDENTITY` to generate a unique integer value for the surrogate key when using a CTAS statement, so this example uses the `ROW_NUMBER` function to generate an incrementing row number for each row in the results ordered by the **ProductID** business key in the staged data.

You can also load a combination of new and updated data into a dimension table by using a CREATE TABLE AS (CTAS) statement to create a new table that UNIONs the existing rows from the dimension table with the new and updated records from the staging table. After creating

the new table, you can delete or rename the current dimension table, and rename the new table to replace it.

SQL

```
CREATE TABLE dbo.DimProductUpsert
WITH
(
    DISTRIBUTION = REPLICATE,
    CLUSTERED COLUMNSTORE INDEX
)
AS
-- New or updated rows
SELECT  stg.ProductID AS ProductBusinessKey,
        stg.ProductName,
        stg.ProductCategory,
        stg.Color,
        stg.Size,
        stg.ListPrice,
        stg.Discontinued
FROM    dbo.StageProduct AS stg
UNION ALL
-- Existing rows
SELECT  dim.ProductBusinessKey,
        dim.ProductName,
        dim.ProductCategory,
        dim.Color,
        dim.Size,
        dim.ListPrice,
        dim.Discontinued
FROM    dbo.DimProduct AS dim
WHERE NOT EXISTS
(   SELECT  *
    FROM dbo.StageProduct AS stg
    WHERE stg.ProductId = dim.ProductBusinessKey
);

RENAME OBJECT dbo.DimProduct TO DimProductArchive;
RENAME OBJECT dbo.DimProductUpsert TO DimProduct;
```

While this technique is effective in merging new and existing dimension data, lack of support for IDENTITY columns means that it's difficult to generate a surrogate key.

> **Tip**
>
> For more information, see **CREATE TABLE AS SELECT (CTAS)** in the Azure Synapse Analytics documentation.

Using an INSERT statement

When you need to load staged data into an existing dimension table, you can use an `INSERT` statement. This approach works if the staged data contains only records for new dimension entities (not updates to existing entities). This approach is much less complicated than the technique in the last section, which required a `UNION ALL` and then renaming table objects.

SQL

```
INSERT INTO dbo.DimCustomer
SELECT CustomerNo AS CustAltKey,
    CustomerName,
    EmailAddress,
    Phone,
    StreetAddress,
    City,
    PostalCode,
    CountryRegion
FROM dbo.StageCustomers
```

Note

Assuming the **DimCustomer** dimension table is defined with an `IDENTITY` **CustomerKey** column for the surrogate key (as described in the previous unit), the key will be generated automatically and the remaining columns will be populated using the values retrieved from the staging table by the `SELECT` query.

Next unit: Load time dimension tables

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Load time dimension tables

3 minutes

Time dimension tables store a record for each time interval based on the *grain* by which you want to aggregate data over time. For example, a time dimension table at the *date* grain contains a record for each date between the earliest and latest dates referenced by the data in related fact tables.

The following code example shows how you can generate a sequence of time dimension values based on a *date* grain.

SQL

```
-- Create a temporary table for the dates we need
CREATE TABLE #TmpStageDate (DateVal DATE NOT NULL)

-- Populate the temp table with a range of dates
DECLARE @StartDate DATE
DECLARE @EndDate DATE
SET @StartDate = '2019-01-01'
SET @EndDate = '2023-12-31'
DECLARE @LoopDate = @StartDate
WHILE @LoopDate <= @EndDate
BEGIN
    INSERT INTO #TmpStageDate VALUES
    (
        @LoopDate
    )
    SET @LoopDate = DATEADD(dd, 1, @LoopDate)
END

-- Insert the dates and calculated attributes into the dimension table
INSERT INTO dbo.DimDate
SELECT CAST(CONVERT(VARCHAR(8), DateVal, 112) as INT), -- date key
    DateVal, --date alt key
    Day(DateVal) -- day number of month
    --,  other derived temporal fields as required
FROM #TmpStageDate
GO

--Drop temporary table
DROP TABLE #TmpStageDate
```

Tip

Scripting this in SQL may be time-consuming in a dedicated SQL pool – it may be more efficient to prepare the data in Microsoft Excel or an external script and import it using the COPY statement.

As the data warehouse is populated in the future with new fact data, you periodically need to extend the range of dates related time dimension tables.

Next unit: Load slowly changing dimensions

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Load slowly changing dimensions

5 minutes

In most relational data warehouses, you need to handle updates to dimension data and support what are commonly referred to as *slowly changing dimensions* (SCDs).

Types of slowly changing dimension

There are multiple kinds of slowly changing dimension, of which three are commonly implemented:

Type 0

Type 0 dimension data can't be changed. Any attempted changes fail.

DateKey	DateAltKey	Day	Month	Year
20230101	01-01-2023	Sunday	January	2023

Type 1

In *type 1* dimensions, the dimension record is updated in-place. Changes made to an existing dimension row apply to all previously loaded facts related to the dimension.

StoreKey	StoreAltKey	StoreName
123	EH199J	~~High Street Store~~ Town Central Store

Type 2

In a *type 2* dimension, a change to a dimension results in a new dimension row. Existing rows for previous versions of the dimension are retained for historical fact analysis and the new row is applied to future fact table entries.

CustomerKey	CustomerAltKey	Name	Address	City	DateFrom	DateTo	IsCurrent
1211	jo@contoso.com	Jo Smith	999 Main St	Seattle	20190101	20230105	False
2996	jo@contoso.com	Jo Smith	1234 9th Ave	Boston	20230106		True

Note

Type 2 dimensions often include columns to track the effective time periods for each version of an entity, and/or a flag to indicate which row represents the current version of the entity. If you're using an incrementing surrogate key and you only need to track the most recently added version of an entity, then you may not need these columns; but before making that decision, consider how you'll look up the appropriate version of an entity when a new fact is entered based on the time at which the event the fact relates to occurred.

Combining INSERT and UPDATE statements

Logic to implement Type 1 and Type 2 updates can be complex, and there are various techniques you can use. For example, you could use a combination of `UPDATE` and `INSERT` statements.

SQL

```
-- New Customers
INSERT INTO dbo.DimCustomer
SELECT stg.*
FROM dbo.StageCustomers AS stg
WHERE NOT EXISTS
    (SELECT * FROM dbo.DimCustomer AS dim
    WHERE dim.CustomerAltKey = stg.CustNo)

-- Type 1 updates (name)
UPDATE dbo.DimCustomer
SET CustomerName = stg.CustomerName
FROM dbo.StageCustomers AS stg
WHERE dbo.DimCustomer.CustomerAltKey = stg.CustomerNo;

-- Type 2 updates (StreetAddress)
INSERT INTO dbo.DimCustomer
SELECT stg.*
FROM dbo.StageCustomers AS stg
JOIN dbo.DimCustomer AS dim
ON stg.CustNo = dim.CustomerAltKey
AND stg.StreetAddress <> dim.StreetAddress;
```

In the previous example, it's assumed that an incrementing surrogate key based on an `IDENTITY` column identifies each row, and that the highest value surrogate key for a given alternate key indicates the most recent or "current" instance of the dimension entity associated with that alternate key. In practice, many data warehouse designers include a Boolean column to indicate the current active instance of a changing dimension or use DateTime fields to indicate the active time periods for each version of the dimension instance. With these approaches, the logic for a type 2 change must include an `INSERT` of the new dimension row *and* an `UPDATE` to mark the current row as inactive.

Using a MERGE statement

As an alternative to using multiple `INSERT` and `UPDATE` statements, you can use a single `MERGE` statement to perform an "*upsert*" operation to insert new records and update existing ones.

SQL

```
MERGE dbo.DimProduct AS tgt
    USING (SELECT * FROM dbo.StageProducts) AS src
```

```
    ON src.ProductID = tgt.ProductBusinessKey
WHEN MATCHED THEN
    -- Type 1 updates
    UPDATE SET
        tgt.ProductName = src.ProductName,
        tgt.ProductCategory = src.ProductCategory,
        tgt.Color = src.Color,
        tgt.Size = src.Size,
        tgt.ListPrice = src.ListPrice,
        tgt.Discontinued = src.Discontinued
WHEN NOT MATCHED THEN
    -- New products
    INSERT VALUES
        (src.ProductID,
        src.ProductName,
        src.ProductCategory,
        src.Color,
        src.Size,
        src.ListPrice,
        src.Discontinued);
```

> **Note**
>
> For more information about the MERGE statement, see the **MERGE documentation for Azure Synapse Analytics**.

Next unit: Load fact tables

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Load fact tables

3 minutes

Typically, a regular data warehouse load operation loads fact tables after dimension tables. This approach ensures that the dimensions to which the facts will be related are already present in the data warehouse.

The staged fact data usually includes the business (alternate) keys for the related dimensions, so your logic to load the data must look up the corresponding surrogate keys. When the data warehouse slowly changing dimensions, the appropriate version of the dimension record must be identified to ensure the correct surrogate key is used to match the event recorded in the fact table with the state of the dimension at the time the fact occurred.

In many cases, you can retrieve the latest "current" version of the dimension; but in some cases you might need to find the right dimension record based on DateTime columns that indicate the period of validity for each version of the dimension.

The following example assumes that the dimension records have an incrementing surrogate key, and that the most recently added version of a specific dimension instance (which will have the highest key value) should be used.

SQL

```
INSERT INTO dbo.FactSales
SELECT  (SELECT MAX(DateKey)
         FROM dbo.DimDate
         WHERE FullDateAlternateKey = stg.OrderDate) AS OrderDateKey,
        (SELECT MAX(CustomerKey)
         FROM dbo.DimCustomer
         WHERE CustomerAlternateKey = stg.CustNo) AS CustomerKey,
        (SELECT MAX(ProductKey)
         FROM dbo.DimProduct
         WHERE ProductAlternateKey = stg.ProductID) AS ProductKey,
        (SELECT MAX(StoreKey)
         FROM dbo.DimStore
         WHERE StoreAlternateKey = stg.StoreID) AS StoreKey,
        OrderNumber,
        OrderLineItem,
        OrderQuantity,
        UnitPrice,
        Discount,
        Tax,
```

```
        SalesAmount
FROM dbo.StageSales AS stg
```

Next unit: Perform post load optimization

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Perform post load optimization

3 minutes

After loading new data into the data warehouse, it's a good idea to rebuild the table indexes and update statistics on commonly queried columns.

Rebuild indexes

The following example rebuilds all indexes on the **DimProduct** table.

SQL

```
ALTER INDEX ALL ON dbo.DimProduct REBUILD
```

Tip

For more information about rebuilding indexes, see the **Indexes on dedicated SQL pool tables in Azure Synapse Analytics** article in the Azure Synapse Analytics documentation.

Update statistics

The following example creates statistics on the **ProductCategory** column of the **DimProduct** table:

SQL

```
CREATE STATISTICS productcategory_stats
ON dbo.DimProduct(ProductCategory);
```

Tip

For more information about updating statistics, see the **Table statistics for dedicated SQL pool in Azure Synapse Analytics** article in the Azure Synapse Analytics documentation.

200 XP

Knowledge check

3 minutes

Choose the best response for each of the questions, then select **Check your answers**.

1. In which order should you load tables in the data warehouse? *

- ○ Staging tables, then dimension tables, then fact tables

  ✔ That's correct. The correct order of operations is stage, populate dimensions to transform keys and SCDs, then load facts with numeric values.

- ○ Staging tables, then fact tables, then dimension tables

  ✕ That's incorrect. Fact tables should be loaded last with the appropriate surrogate keys from dimensions.

- ○ Dimension tables, then staging tables, then fact tables

2. Which command should you use to load a staging table with data from files in the data lake? *

- ○ COPY

  ✔ That's Correct. With existing tables, the copy command is the most efficient way to populate the warehouse from the data lake.

- ○ LOAD
- ○ INSERT

3. When a customer changes their phone number, the change should be made in the existing row for that customer in the dimension table. What type of slowly changing dimension does this scenario require? *

- ○ Type 0
- ○ Type 1

✔ That's correct. In this case, simply changing the number with no history is appropriate.

○ Type 2

Next unit: Summary

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Introduction

3 minutes

In this module, you will learn some of the features you can use to manage and monitor Azure Synapse Analytics.

At the end of this module, you will

- Scale compute resources in Azure Synapse Analytics
- Pause compute in Azure Synapse Analytics
- Manage workloads in Azure Synapse Analytics
- Use Azure Advisor to review recommendations
- Use Dynamic Management Views to identify and troubleshoot query performance

Prerequisites

Before taking this module, it is recommended that the student is able to:

- Log into the Azure portal
- Create a Synapse Analytics Workspace
- Create an Azure Synapse Analytics SQL Pool

Next unit: Scale compute resources in Azure Synapse Analytics

Continue

100 XP

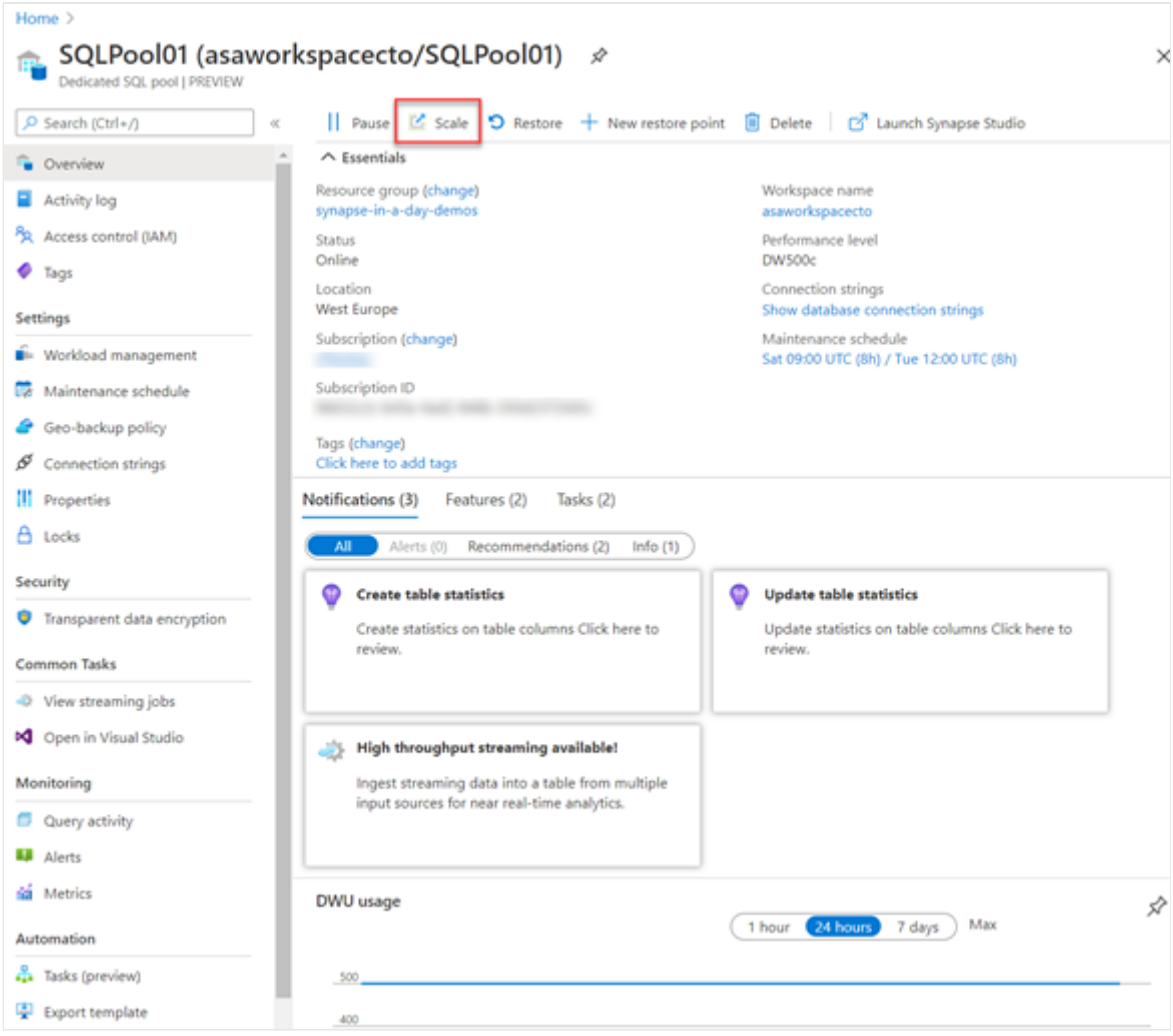
Scale compute resources in Azure Synapse Analytics

8 minutes

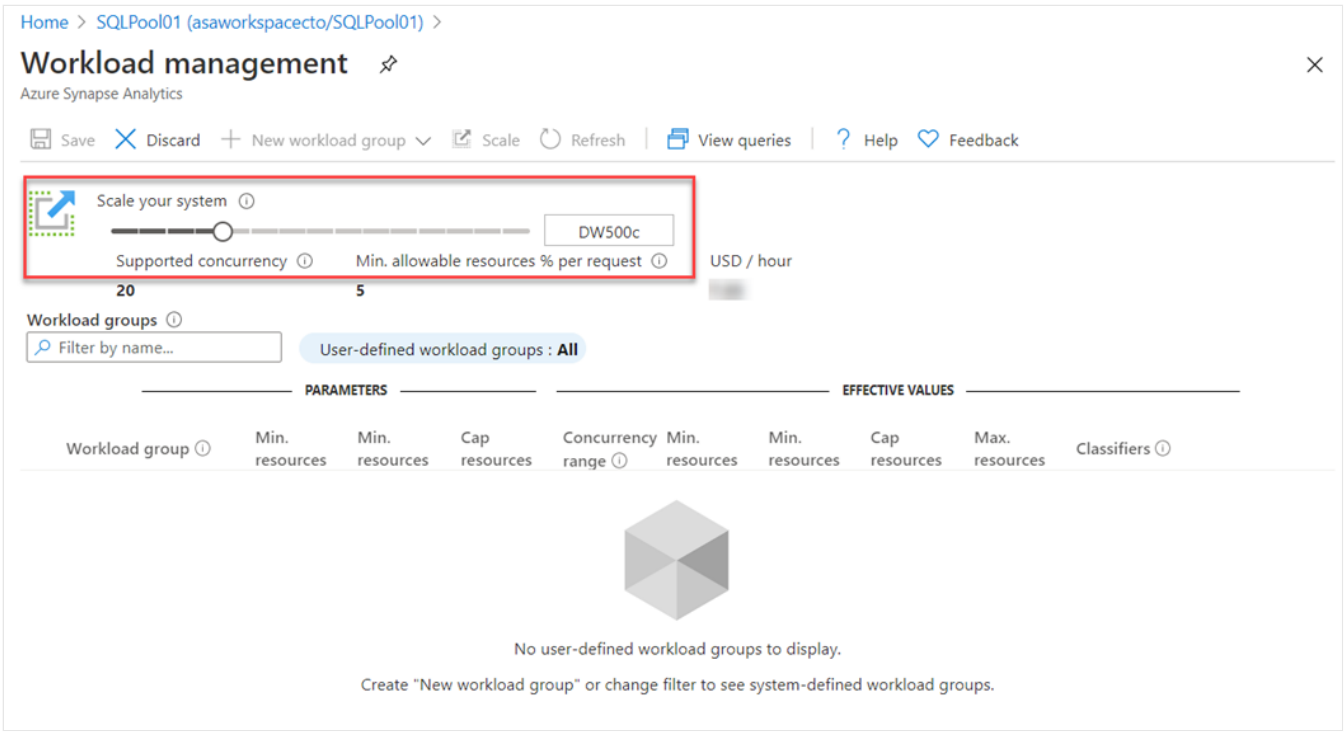
One of the key management features that you have at your disposal within Azure Synapse Analytics, is the ability to scale the compute resources for SQL or Spark pools to meet the demands of processing your data. In SQL pools, the unit of scale is an abstraction of compute power that is known as a data warehouse unit. Compute is separate from storage, which enables you to scale compute independently of the data in your system. This means you can scale up and scale down the compute power to meet your needs.

You can scale a Synapse SQL pool either through the Azure portal, Azure Synapse Studio or programmatically using TSQL or PowerShell.

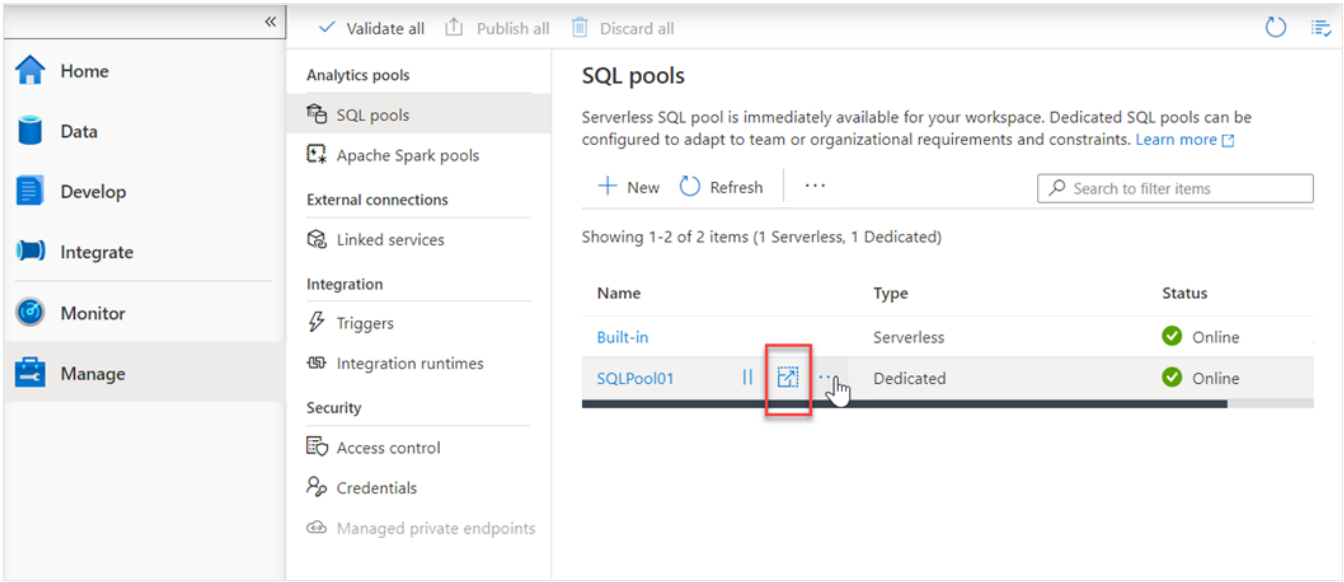
In the Azure portal, you can click on **scale** icon



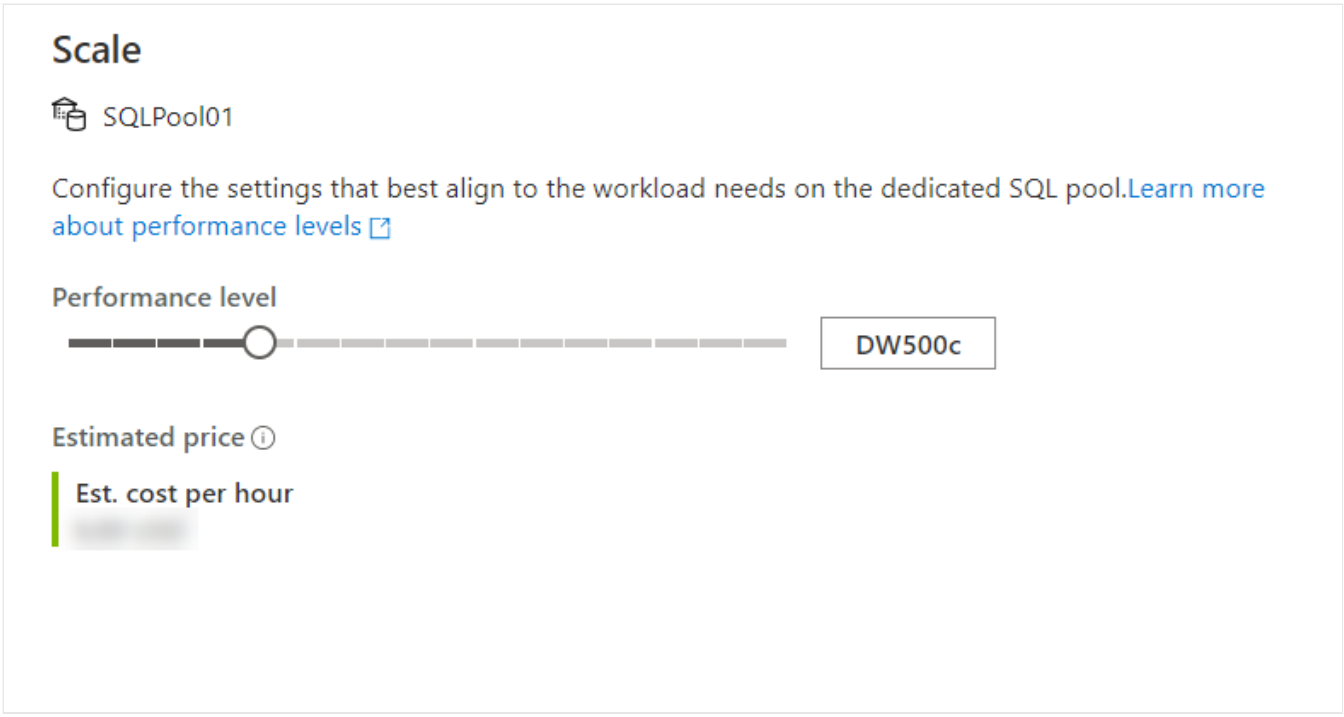
And then you can adjust the **slider** to scale the SQL Pool



Another option to scale is within Azure Synapse Studio, click on the **scale** icon:



And then move the **slider** as follows:



You can also make the modification using Transact-SQL

```SQL
ALTER DATABASE mySampleDataWarehouse
MODIFY (SERVICE_OBJECTIVE = 'DW300c');
```

Or by using PowerShell

```PowerShell
Set-AzSqlDatabase -ResourceGroupName "resourcegroupname" -DatabaseName
"mySampleDataWarehouse" -ServerName "sqlpoolservername" -
RequestedServiceObjectiveName "DW300c"
```

Scaling Apache Spark pools in Azure Synapse Analytics

Apache Spark pools for Azure Synapse Analytics uses an **Autoscale** feature that automatically scales the number of nodes in a cluster instance up and down. During the creation of a new Spark pool, a minimum and maximum number of nodes can be set when **Autoscale** is selected. Autoscale then monitors the resource requirements of the load and scales the number of nodes up or down. To enable the Autoscale feature, complete the following steps as part of the normal pool creation process:

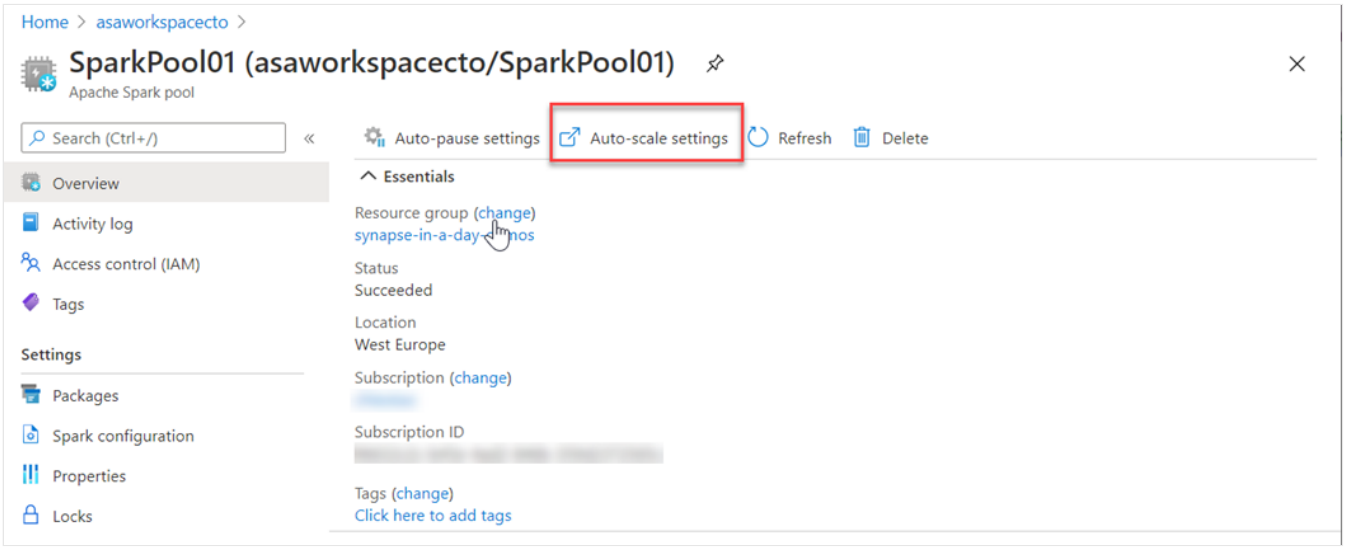
1. On the **Basics** tab, select the **Enable autoscale** checkbox.

2. Enter the desired values for the following properties:

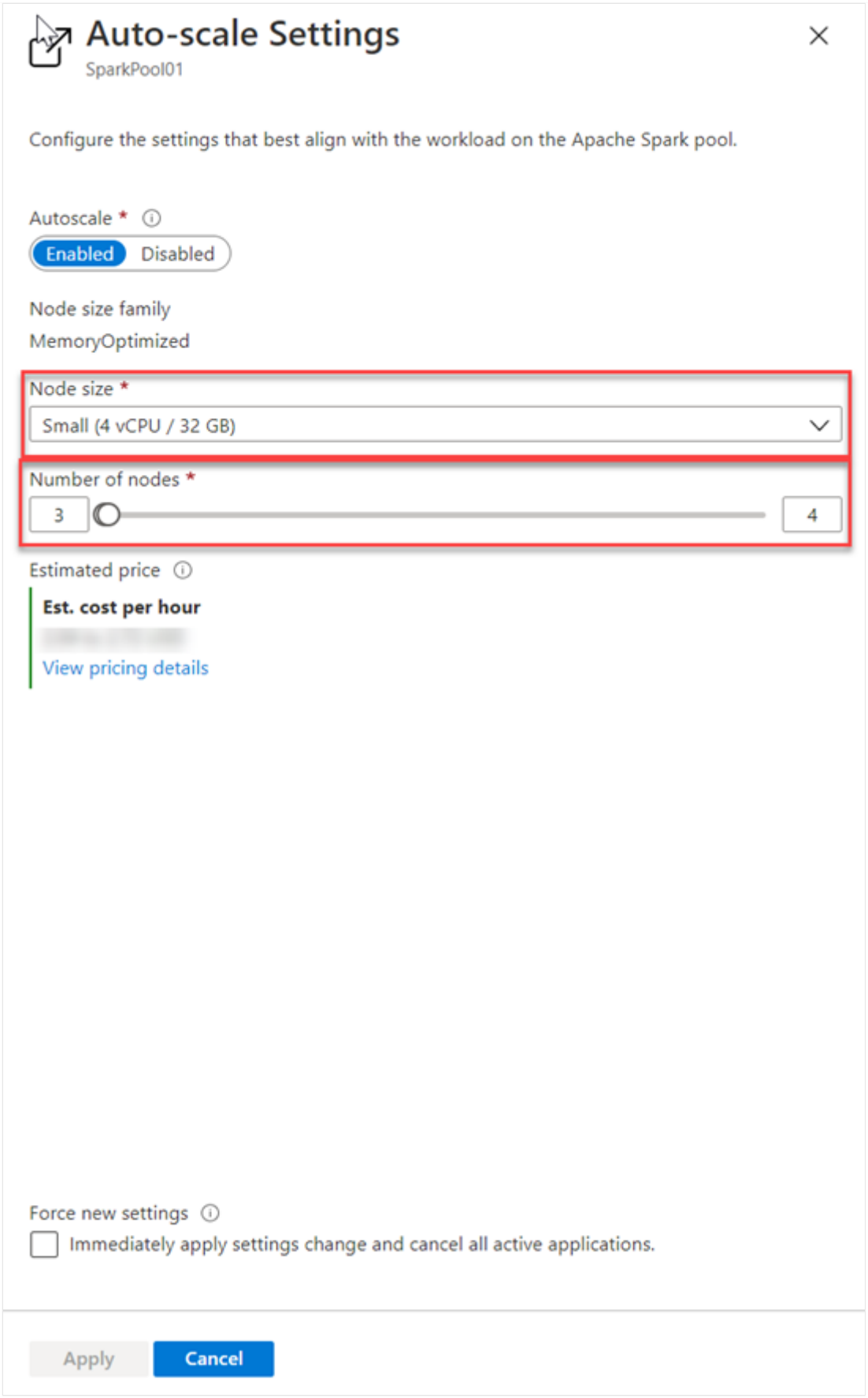
    - **Min** number of nodes.
    - **Max** number of nodes.

The initial number of nodes will be the minimum. This value defines the initial size of the instance when it's created. The minimum number of nodes can't be fewer than three.

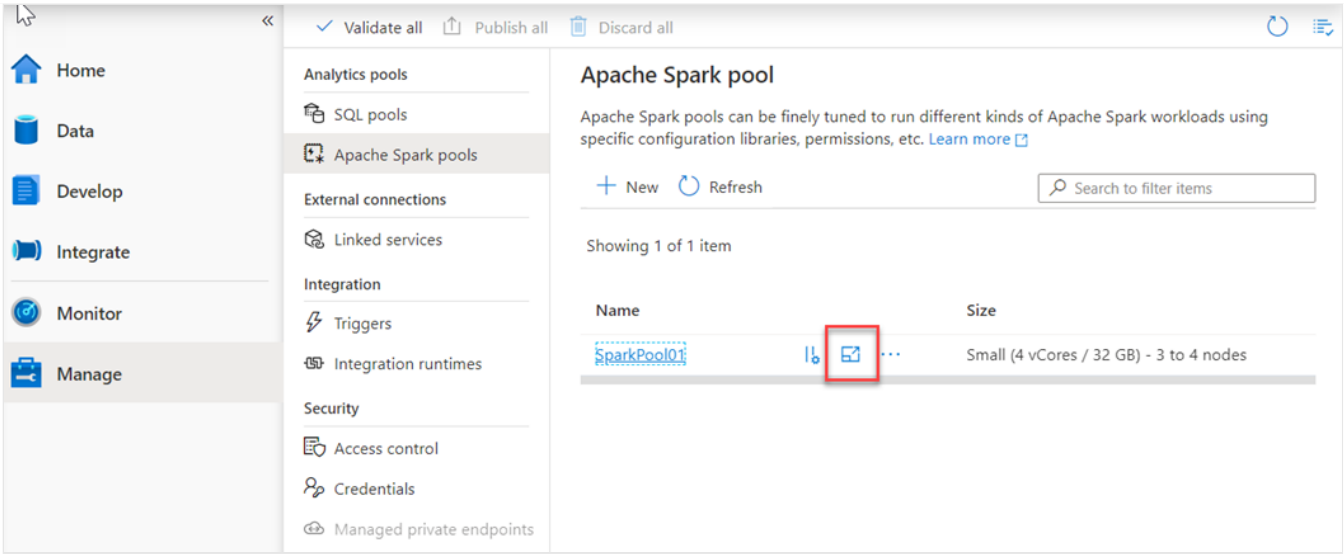
You can also modify this in the Azure portal, you can click on **auto-scale settings** icon



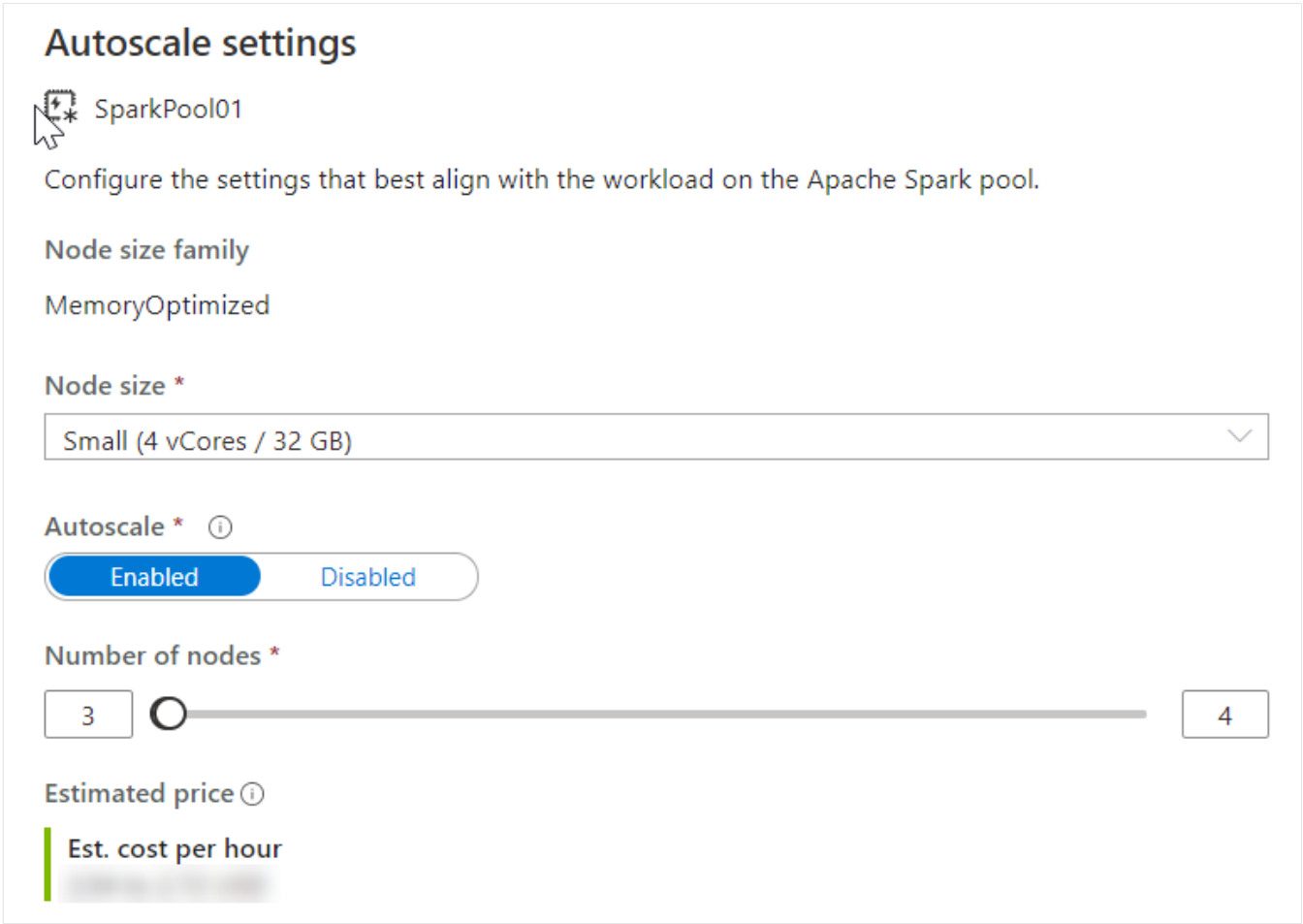
Choose the node size and the number of nodes



and for Azure Synapse Studio as follows



And Choose the node size and the number of nodes



Autoscale continuously monitors the Spark instance and collects the following metrics:

Metric	Description
Total Pending CPU	The total number of cores required to start execution of all pending nodes.

Metric	Description
Total Pending Memory	The total memory (in MB) required to start execution of all pending nodes.
Total Free CPU	The sum of all unused cores on the active nodes.
Total Free Memory	The sum of unused memory (in MB) on the active nodes.
Used Memory per Node	The load on a node. A node on which 10 GB of memory is used, is considered under more load than a worker with 2 GB of used memory.

The following conditions will then autoscale the memory or CPU

Scale-up	Scale-down
Total pending CPU is greater than total free CPU for more than 1 minute.	Total pending CPU is less than total free CPU for more than 2 minutes.
Total pending memory is greater than total free memory for more than 1 minute.	Total pending memory is less than total free memory for more than 2 minutes.

The scaling operation can take between 1 -5 minutes. During an instance where there is a scale down process, Autoscale will put the nodes in decommissioning state so that no new executors can launch on that node.

The running jobs will continue to run and finish. The pending jobs will wait to be scheduled as normal with fewer available nodes.

Next unit: Pause compute in Azure Synapse Analytics

Continue >

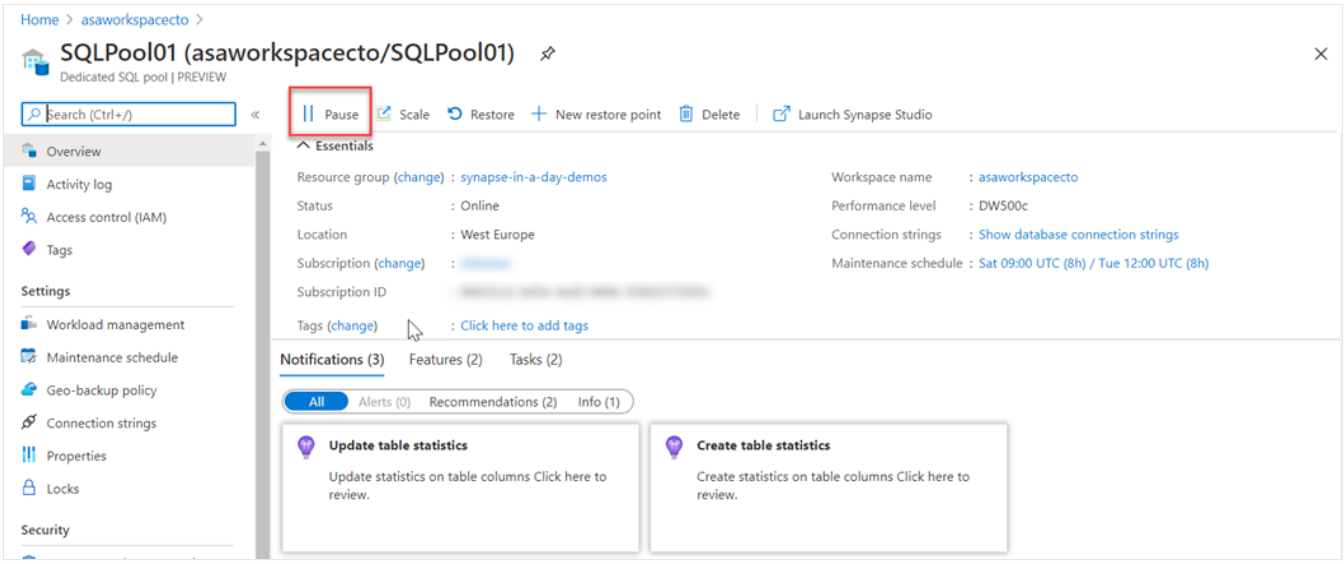
100 XP

Pause compute in Azure Synapse Analytics

3 minutes

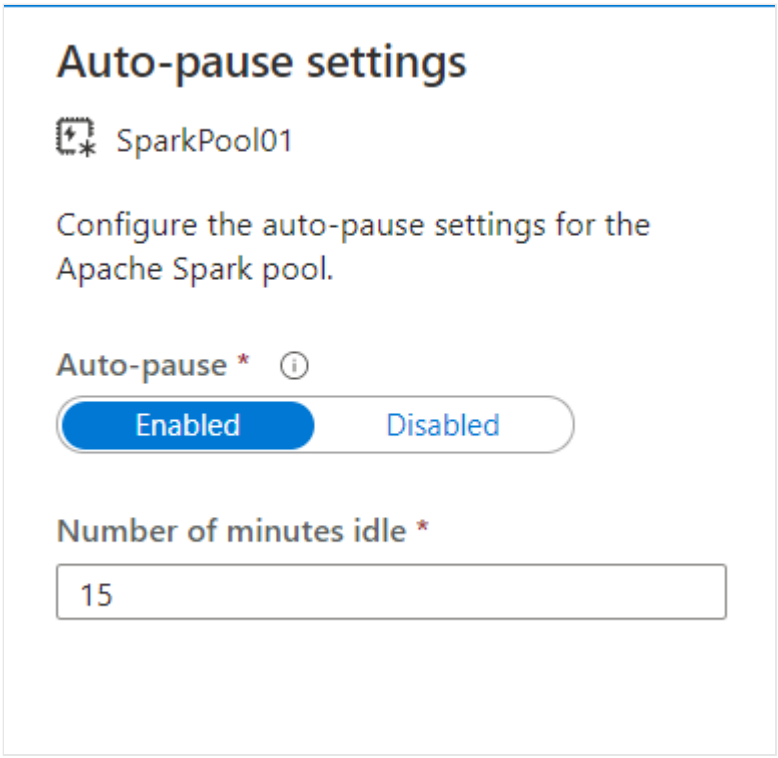
When performing the batch movement of data to populate a data warehouse, it is typical for the data engineer to understand the schedule on which the data loads take place. In these circumstances, you may be able to predict the periods of downtime in the data loading and querying process and take advantage of the pause operations to minimize your costs.

In the Azure portal you can use the Pause command within the dedicated SQL pool



And this can also be used within Azure Synapse Studio for Apache Spark pools too, in the Manage hub.

Which allows you to enable it, and set the number of minutes idle



Next unit: Manage workloads in Azure Synapse Analytics

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Manage workloads in Azure Synapse Analytics

10 minutes

Azure Synapse Analytics allows you to create, control and manage resource availability when workloads are competing. This allows you to manage the relative importance of each workload when waiting for available resources.

To facilitate faster load times, you can create a workload classifier for the load user with the "importance" set to above_normal or High. Workload importance ensures that the load takes precedence over other waiting tasks of a lower importance rating. Use this in conjunction with your own workload group definitions for workload isolation to manage minimum and maximum resource allocations during peak and quiet periods.

Dedicated SQL pool workload management in Azure Synapse consists of three high-level concepts:

- Workload Classification
- Workload Importance
- Workload Isolation

These capabilities give you more control over how your workload utilizes system resources.

Workload classification

Workload management classification allows workload policies to be applied to requests through assigning resource classes and importance.

While there are many ways to classify data warehousing workloads, the simplest and most common classification is load and query. You load data with insert, update, and delete statements. You query the data using selects. A data warehousing solution will often have a workload policy for load activity, such as assigning a higher resource class with more resources. A different workload policy could apply to queries, such as lower importance compared to load activities.

You can also subclassify your load and query workloads. Subclassification gives you more control of your workloads. For example, query workloads can consist of cube refreshes, dashboard queries or ad-hoc queries. You can classify each of these query workloads with

different resource classes or importance settings. Load can also benefit from subclassification. Large transformations can be assigned to larger resource classes. Higher importance can be used to ensure key sales data is loaded before weather data or a social data feed.

Not all statements are classified as they do not require resources or need importance to influence execution. DBCC commands, BEGIN, COMMIT, and ROLLBACK TRANSACTION statements are not classified.

Workload importance

Workload importance influences the order in which a request gets access to resources. On a busy system, a request with higher importance has first access to resources. Importance can also ensure ordered access to locks. There are five levels of importance: low, below_normal, normal, above_normal, and high. Requests that don't set importance are assigned the default level of normal. Requests that have the same importance level have the same scheduling behavior that exists today.

Workload isolation

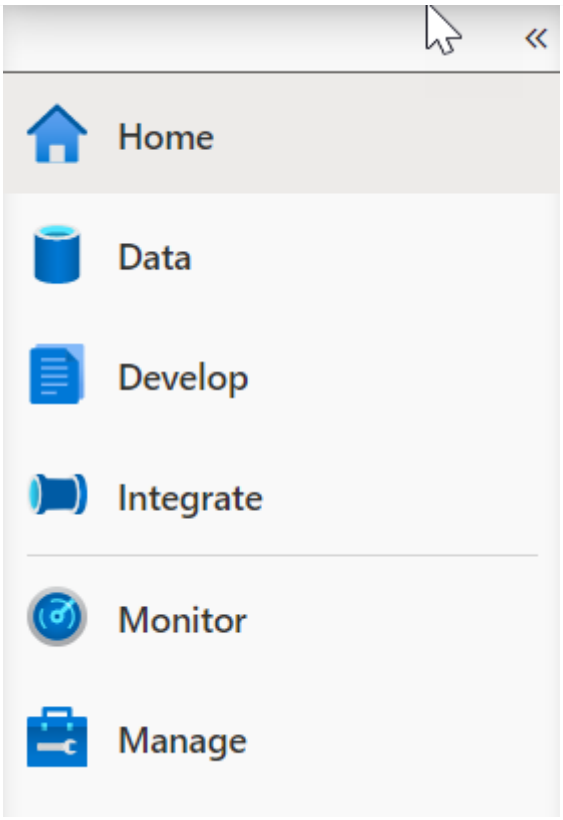
Workload isolation reserves resources for a workload group. Resources reserved in a workload group are held exclusively for that workload group to ensure execution. Workload groups also allow you to define the amount of resources that are assigned per request, much like resource classes do. Workload groups give you the ability to reserve or cap the amount of resources a set of requests can consume. Finally, workload groups are a mechanism to apply rules, such as query timeout, to requests.

You can perform the following steps to implement workload management

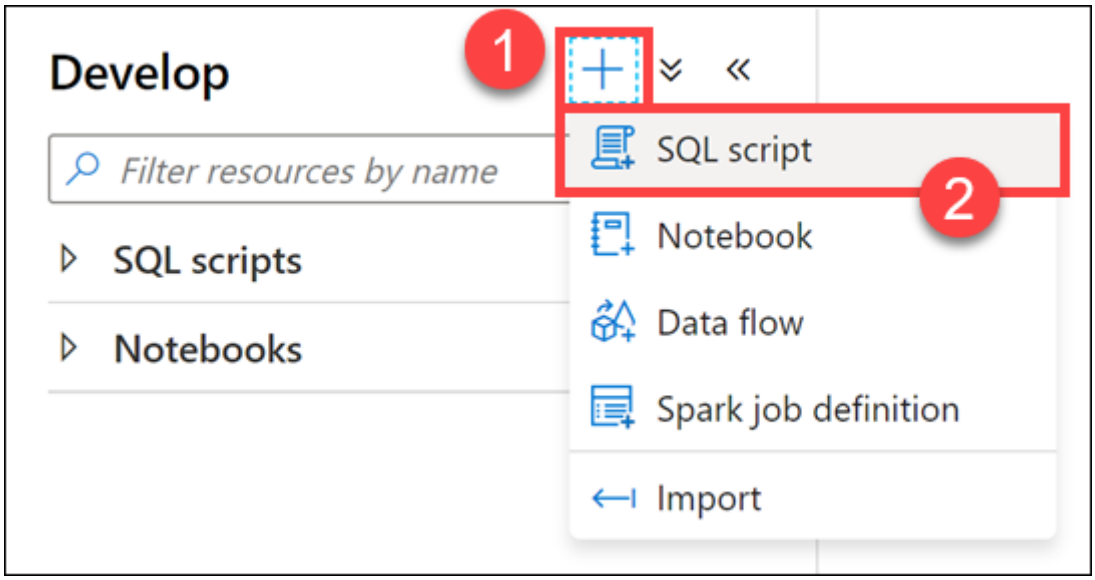
Create a workload classifier to add importance to certain queries

Your organization has asked you if there is a way to mark queries executed by the CEO as more important than others, so they don't appear slow due to heavy data loading or other workloads in the queue. You decide to create a workload classifier and add importance to prioritize the CEO's queries.

1. Select the **Develop** hub.



2. From the **Develop** menu, select the + button **(1)** and choose **SQL Script (2)** from the context menu.



3. In the toolbar menu, connect to the **SQL Pool** database to execute the query.

4. In the query window, replace the script with the following to confirm that there are no queries currently being run by users logged in as `asa.sql.workload01`, representing the CEO of the organization or `asa.sql.workload02` representing the data analyst working on the project:

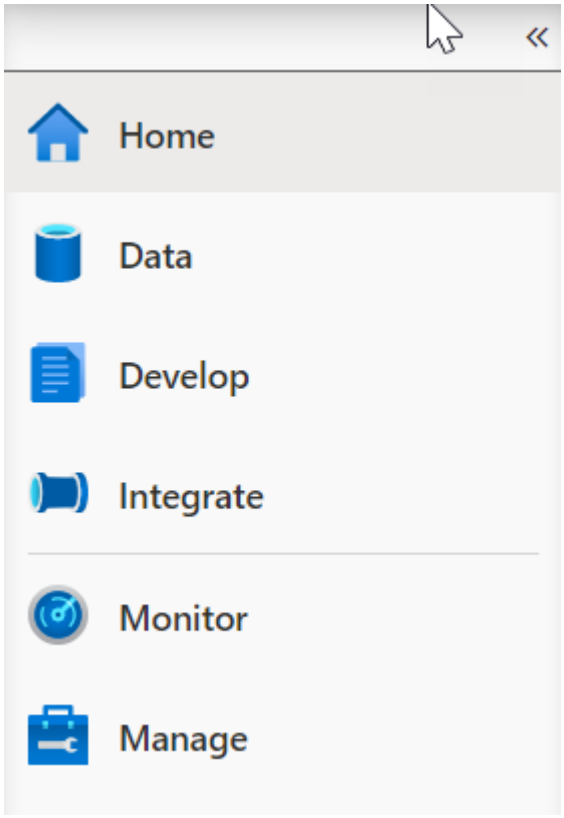
```
SQL
```

```
--First, let's confirm that there are no queries currently being run
by users logged in workload01 or workload02

SELECT s.login_name, r.[Status], r.Importance, submit_time,
start_time ,s.session_id FROM sys.dm_pdw_exec_sessions s
JOIN sys.dm_pdw_exec_requests r ON s.session_id = r.session_id
WHERE s.login_name IN ('asa.sql.workload01','asa.sql.workload02') and
Importance
is not NULL AND r.[status] in ('Running','Suspended')
--and submit_time>dateadd(minute,-2,getdate())
ORDER BY submit_time ,s.login_name
```

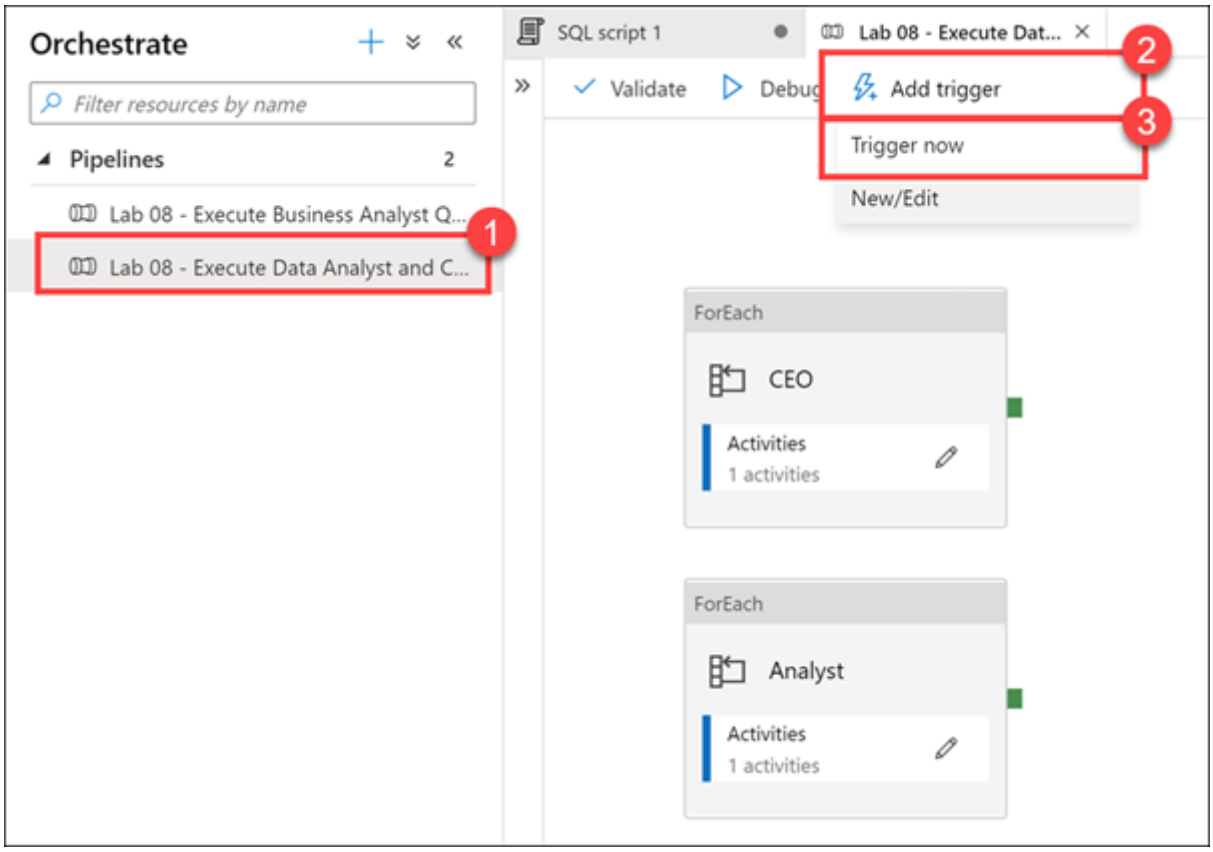
5. Select **Run** from the toolbar menu to execute the SQL command.

Now that we have confirmed that there are no running queries, we need to flood the system with queries and see what happens for `asa.sql.workload01` and `asa.sql.workload02`. To do this, we'll run a Azure Synapse Pipeline which triggers queries.

6. Select the **Integrate** hub.



7. Select the **Lab 08 - Execute Data Analyst and CEO Queries** Pipeline **(1)**, which will run / trigger the `asa.sql.workload01` and `asa.sql.workload02` queries. Select **Add trigger (2)**, then **Trigger now (3)**. In the dialog that appears, select **OK**.



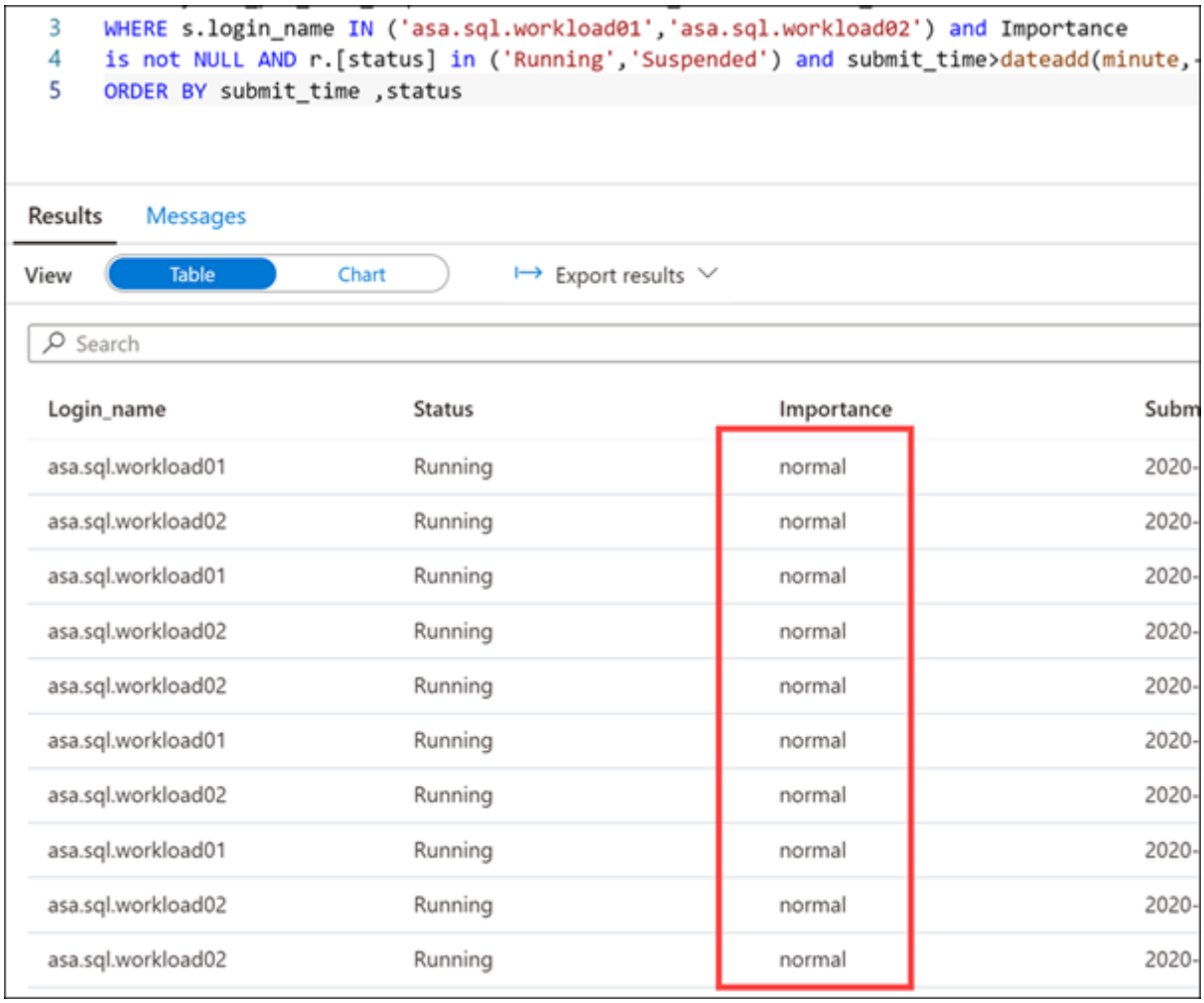
8. Let's see what happened to all the queries we just triggered as they flood the system. In the query window, replace the script with the following:

```
SQL
```

```
SELECT s.login_name, r.[Status], r.Importance, submit_time, start_time
,s.session_id FROM sys.dm_pdw_exec_sessions s
JOIN sys.dm_pdw_exec_requests r ON s.session_id = r.session_id
WHERE s.login_name IN ('asa.sql.workload01','asa.sql.workload02') and
Importance
is not NULL AND r.[status] in ('Running','Suspended') and
submit_time>dateadd(minute,-2,getdate())
ORDER BY submit_time ,status
```

9. Select **Run** from the toolbar menu to execute the SQL command.

You should see an output similar to the following:



Notice that the **Importance** level for all queries is set to **normal**.

10. We will give our `asa.sql.workload01` user queries priority by implementing the **Workload Importance** feature. In the query window, replace the script with the following:

```
SQL

IF EXISTS (SELECT * FROM sys.workload_management_workload_classifiers
WHERE name = 'CEO')
BEGIN
    DROP WORKLOAD CLASSIFIER CEO;
END
CREATE WORKLOAD CLASSIFIER CEO
  WITH (WORKLOAD_GROUP = 'largerc'
  ,MEMBERNAME = 'asa.sql.workload01',IMPORTANCE = High);
```

We are executing this script to create a new **Workload Classifier** named `CEO` that uses the `largerc` Workload Group and sets the **Importance** level of the queries to **High**.

11. Select **Run** from the toolbar menu to execute the SQL command.

12. Let's flood the system again with queries and see what happens this time for `asa.sql.workload01` and `asa.sql.workload02` queries. To do this, we'll run an Azure Synapse Pipeline which triggers queries. **Select** the `Integrate` Tab, **run** the **Lab 08 - Execute Data Analyst and CEO Queries** Pipeline, which will run / trigger the `asa.sql.workload01` and `asa.sql.workload02` queries.

13. In the query window, replace the script with the following to see what happens to the `asa.sql.workload01` queries this time:

```
SQL

SELECT s.login_name, r.[Status], r.Importance, submit_time, start_time
,s.session_id FROM sys.dm_pdw_exec_sessions s
JOIN sys.dm_pdw_exec_requests r ON s.session_id = r.session_id
WHERE s.login_name IN ('asa.sql.workload01','asa.sql.workload02') and
Importance
is not NULL AND r.[status] in ('Running','Suspended') and
submit_time>dateadd(minute,-2,getdate())
ORDER BY submit_time ,status desc
```

14. Select **Run** from the toolbar menu to execute the SQL command.

You should see an output similar to the following:

```
3   WHERE s.login_name IN ('asa.sql.workload01','asa.sql.workload02') and Importanc
4   is not NULL AND r.[status] in ('Running','Suspended') and submit_time>dateadd(m
5   ORDER BY submit_time ,status desc
```

Results Messages

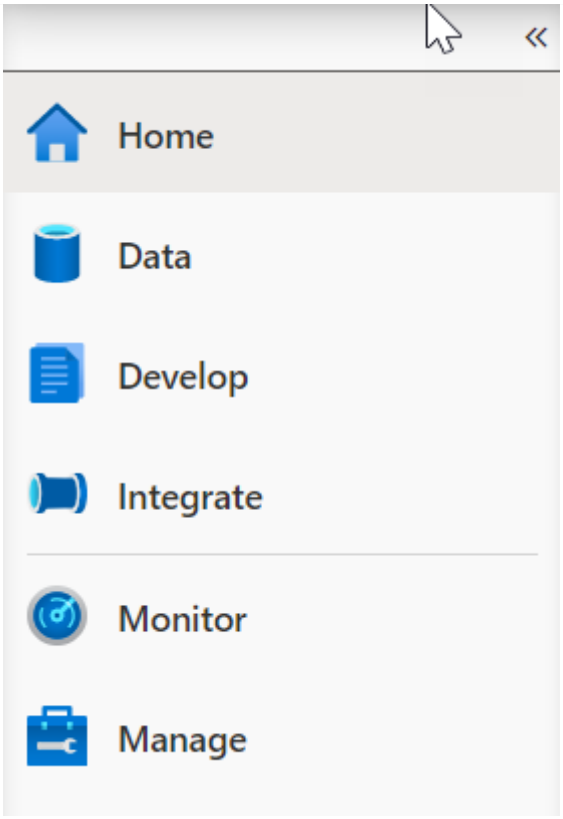
View Table Chart Export results

Search

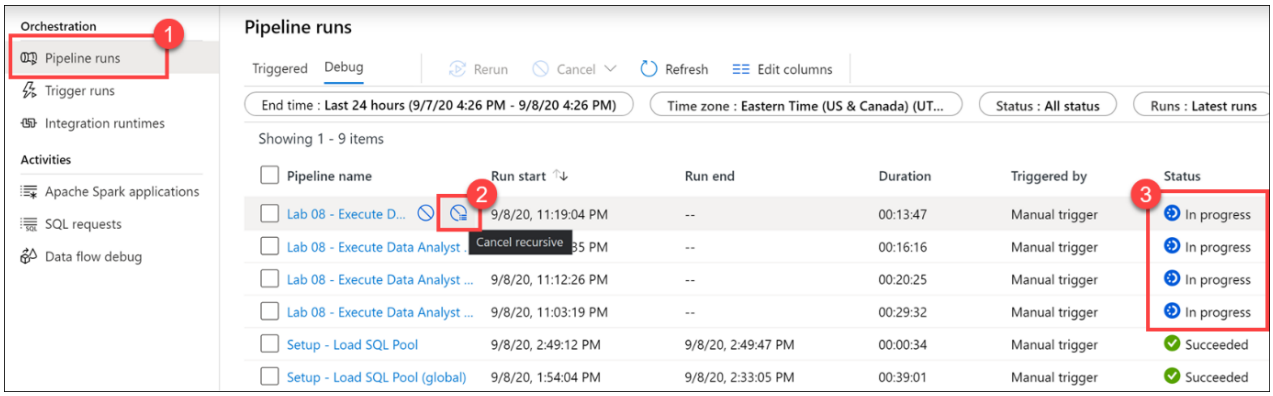
Login_name	Status	Importance
asa.sql.workload01	Suspended	high
asa.sql.workload01	Suspended	high
asa.sql.workload02	Suspended	normal
asa.sql.workload01	Suspended	high
asa.sql.workload01	Suspended	high
asa.sql.workload02	Suspended	normal
asa.sql.workload01	Suspended	high
asa.sql.workload01	Suspended	high
asa.sql.workload01	Suspended	high
asa.sql.workload01	Suspended	high
asa.sql.workload02	Suspended	normal

Notice that the queries executed by the `asa.sql.workload01` user have a **high** importance.

15. Select the **Monitor** hub.



16. Select **Pipeline runs (1)**, and then select **Cancel recursive (2)** for each running Lab 08 pipelines, marked **In progress (3)**. This will help speed up the remaining tasks.



Reserve resources for specific workloads through workload isolation

Workload isolation means resources are reserved, exclusively, for a workload group. Workload groups are containers for a set of requests and are the basis for how workload management, including workload isolation, is configured on a system. A simple workload management configuration can manage data loads and user queries.

In the absence of workload isolation, requests operate in the shared pool of resources. Access to resources in the shared pool is not guaranteed and is assigned on an importance basis.

Given the workload requirements provided by Tailwind Traders, you decide to create a new workload group called `CEODemo` to reserve resources for queries executed by the CEO.

Let's start by experimenting with different parameters.

1. In the query window, replace the script with the following:

   SQL

   ```
   IF NOT EXISTS (SELECT * FROM sys.workload_management_workload_groups
   where name = 'CEODemo')
   BEGIN
       Create WORKLOAD GROUP CEODemo WITH
       ( MIN_PERCENTAGE_RESOURCE = 50        -- integer value
       ,REQUEST_MIN_RESOURCE_GRANT_PERCENT = 25 --
       ,CAP_PERCENTAGE_RESOURCE = 100
       )
   END
   ```

   The script creates a workload group called `CEODemo` to reserve resources exclusively for the workload group. In this example, a workload group with a `MIN_PERCENTAGE_RESOURCE` set to 50% and `REQUEST_MIN_RESOURCE_GRANT_PERCENT` set to 25% is guaranteed 2 concurrency.

2. Select **Run** from the toolbar menu to execute the SQL command.

3. In the query window, replace the script with the following to create a Workload Classifier called `CEODreamDemo` that assigns a workload group and importance to incoming requests:

   SQL

   ```
   IF NOT EXISTS (SELECT * FROM sys.workload_management_workload_classi-
   fiers where  name = 'CEODreamDemo')
   BEGIN
       Create Workload Classifier CEODreamDemo with
       ( Workload_Group ='CEODemo',MemberName='asa.sql.workload02',IMPOR-
   TANCE = BELOW_NORMAL);
   END
   ```

   This script sets the Importance to **BELOW_NORMAL** for the `asa.sql.workload02` user, through the new `CEODreamDemo` Workload Classifier.

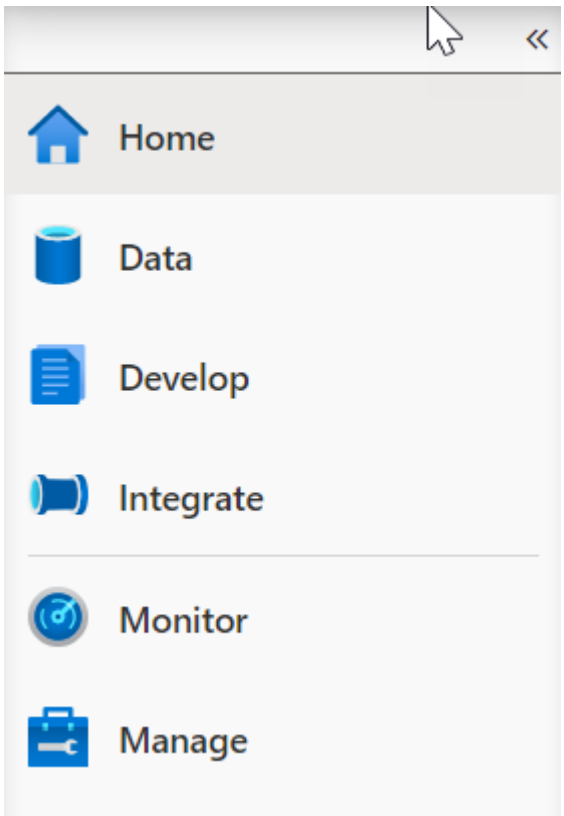
4. Select **Run** from the toolbar menu to execute the SQL command.

5. In the query window, replace the script with the following to confirm that there are no active queries being run by `asa.sql.workload02` (suspended queries are OK):

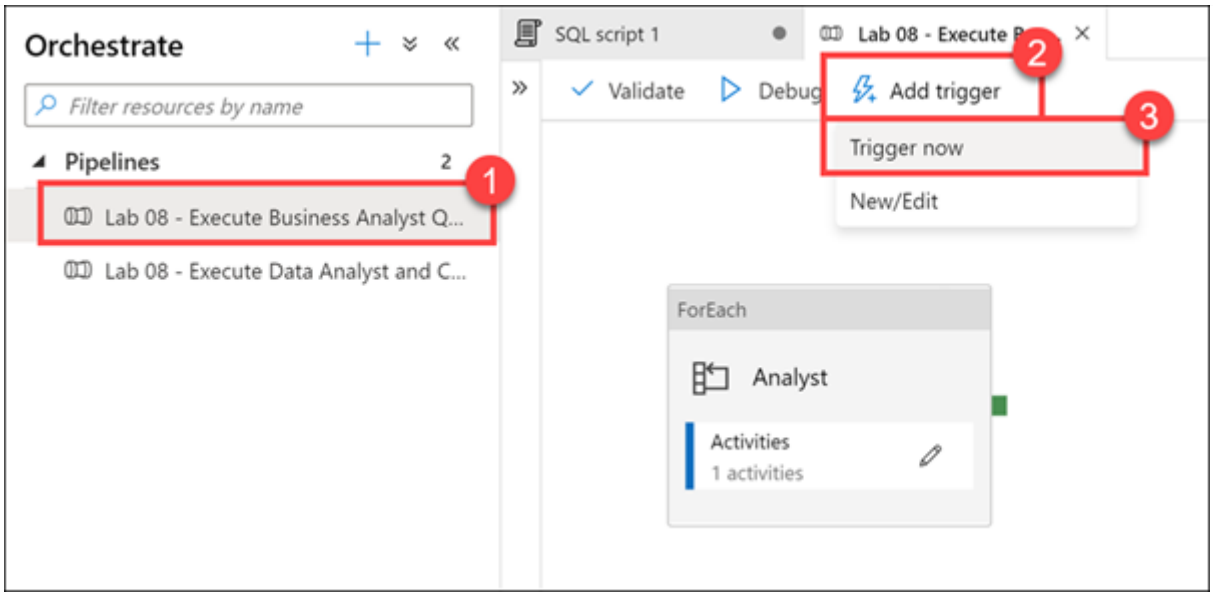
```
SQL

SELECT s.login_name, r.[Status], r.Importance, submit_time,
start_time ,s.session_id FROM sys.dm_pdw_exec_sessions s
JOIN sys.dm_pdw_exec_requests r ON s.session_id = r.session_id
WHERE s.login_name IN ('asa.sql.workload02') and Importance
is not NULL AND r.[status] in ('Running','Suspended')
ORDER BY submit_time, status
```

6. Select **Run** from the toolbar menu to execute the SQL command.

7. Select the **Integrate** hub.



8. Select the **Lab 08 - Execute Business Analyst Queries** Pipeline **(1)**, which will run / trigger `asa.sql.workload02` queries. Select **Add trigger (2)**, then **Trigger now (3)**. In the dialog that appears, select **OK**.

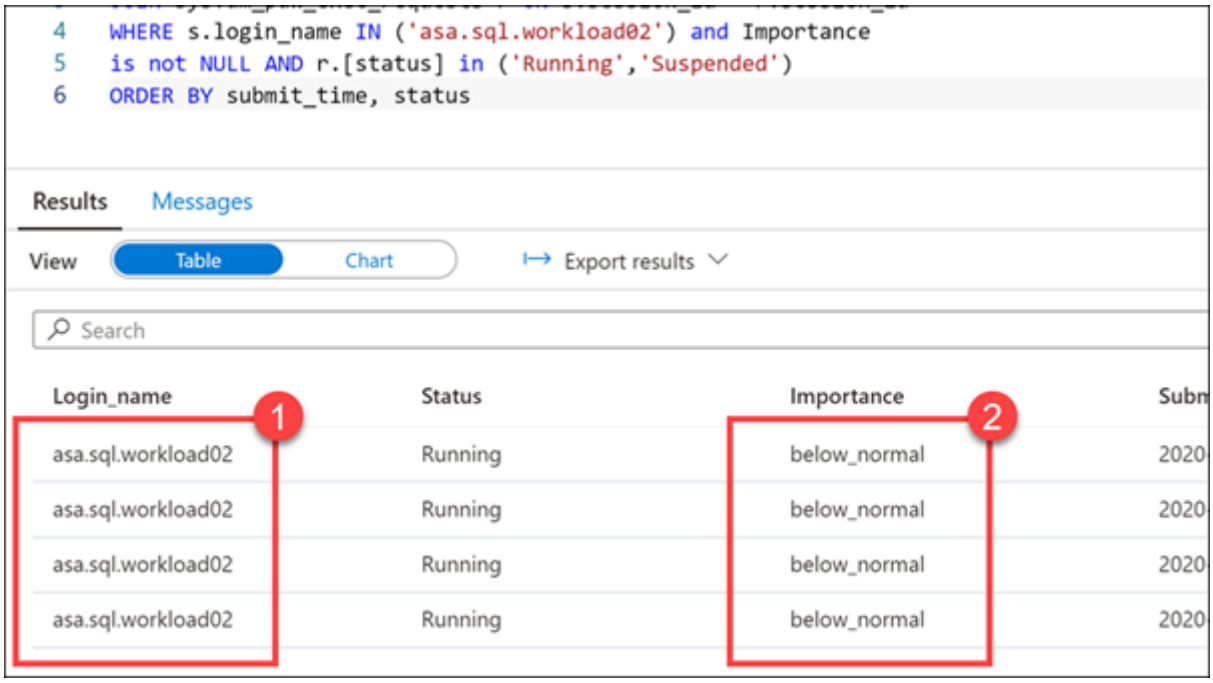


9. In the query window, replace the script with the following to see what happened to all the `asa.sql.workload02` queries we just triggered as they flood the system:

```
SQL

SELECT s.login_name, r.[Status], r.Importance, submit_time,
start_time ,s.session_id FROM sys.dm_pdw_exec_sessions s
JOIN sys.dm_pdw_exec_requests r ON s.session_id = r.session_id
WHERE s.login_name IN ('asa.sql.workload02') and Importance
is not NULL AND r.[status] in ('Running','Suspended')
ORDER BY submit_time, status
```

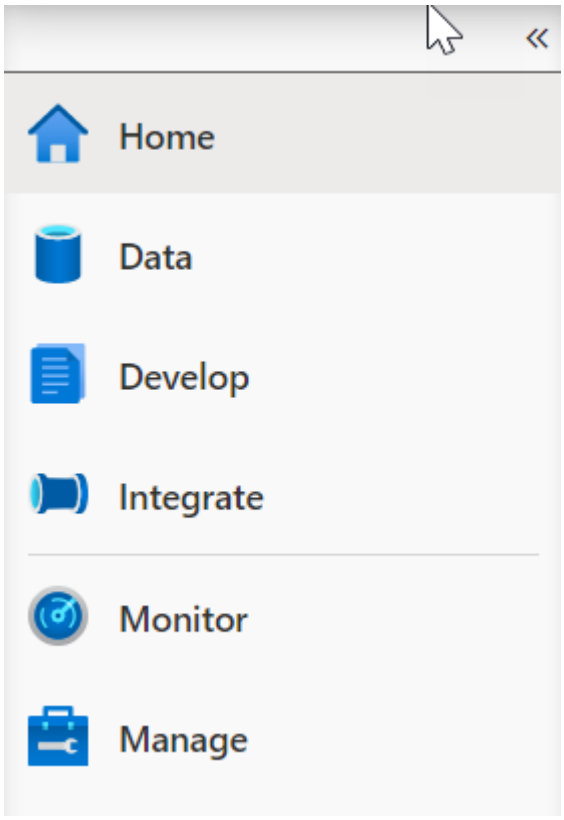
10. Select **Run** from the toolbar menu to execute the SQL command.

You should see an output similar to the following that shows the importance for each session set to `below_normal`:

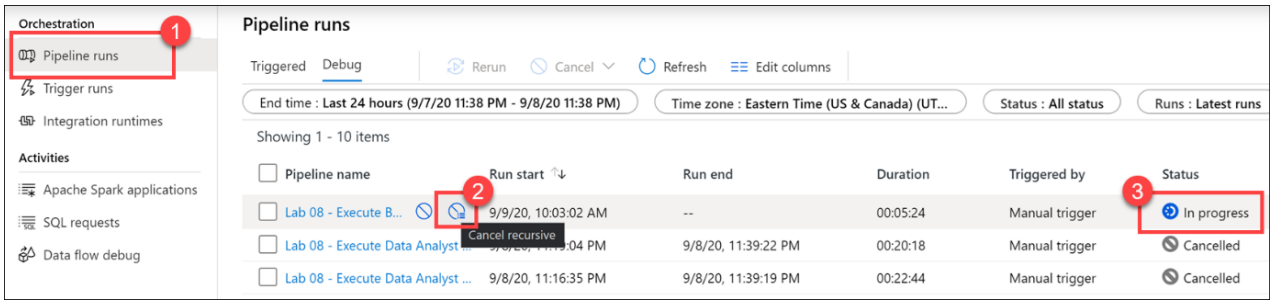


Notice that the running scripts are executed by the `asa.sql.workload02` user **(1)** with an Importance level of **below_normal (2)**. We have successfully configured the business analyst queries to execute at a lower importance than the CEO queries. We can also see that the `CEODreamDemo` Workload Classifier works as expected.

11. Select the **Monitor** hub.



12. Select **Pipeline runs (1)**, and then select **Cancel recursive (2)** for each running Lab 08 pipelines, marked **In progress (3)**. This will help speed up the remaining tasks.



13. Return to the query window under the **Develop** hub. In the query window, replace the script with the following to set 3.25% minimum resources per request:

SQL

```
IF  EXISTS (SELECT * FROM sys.workload_management_workload_classifiers
where group_name = 'CEODemo')
BEGIN
    Drop Workload Classifier CEODreamDemo
    DROP WORKLOAD GROUP CEODemo
    --- Creates a workload group 'CEODemo'.
        Create  WORKLOAD GROUP CEODemo WITH
    (MIN_PERCENTAGE_RESOURCE = 26 -- integer value
        ,REQUEST_MIN_RESOURCE_GRANT_PERCENT = 3.25 -- factor of 26
(guaranteed more than 4 concurrencies)
    ,CAP_PERCENTAGE_RESOURCE = 100
    )
    --- Creates a workload Classifier 'CEODreamDemo'.
    Create Workload Classifier CEODreamDemo with
    (Workload_Group ='CEODemo',MemberName='asa.sql.workload02',IMPOR-
TANCE = BELOW_NORMAL);
END
```

> **Note**
>
> Configuring workload containment implicitly defines a maximum level of concurrency. With a CAP_PERCENTAGE_RESOURCE set to 60% and a REQUEST_MIN_RESOURCE_GRANT_PERCENT set to 1%, up to a 60-concurrency level is allowed for the workload group. Consider the method included below for determining the maximum concurrency: [Max Concurrency] = [CAP_PERCENTAGE_RESOURCE] / [REQUEST_MIN_RESOURCE_GRANT_PERCENT]

14. Select **Run** from the toolbar menu to execute the SQL command.

15. Let's flood the system again and see what happens for `asa.sql.workload02`. To do this, we will run an Azure Synapse Pipeline which triggers queries. Select the `Integrate` Tab. **Run** the **Lab 08 - Execute Business Analyst Queries** Pipeline, which will run / trigger `asa.sql.workload02` queries.

16. In the query window, replace the script with the following to see what happened to all of the `asa.sql.workload02` queries we just triggered as they flood the system:

```
SQL

SELECT s.login_name, r.[Status], r.Importance, submit_time,
start_time ,s.session_id FROM sys.dm_pdw_exec_sessions s
JOIN sys.dm_pdw_exec_requests r ON s.session_id = r.session_id
WHERE s.login_name IN ('asa.sql.workload02') and Importance
is  not NULL AND r.[status] in ('Running','Suspended')
ORDER BY submit_time, status
```

17. Select **Run** from the toolbar menu to execute the SQL command.

After several moments (up to a minute), we should see several concurrent executions by the `asa.sql.workload02` user running at **below_normal** importance. We have validated that the modified Workload Group and Workload Classifier works as expected.

```
4   WHERE s.login_name IN ('asa.sql.workload02') and Importance
5   is  not NULL AND r.[status] in ('Running','Suspended')
6   ORDER BY submit_time, status
```

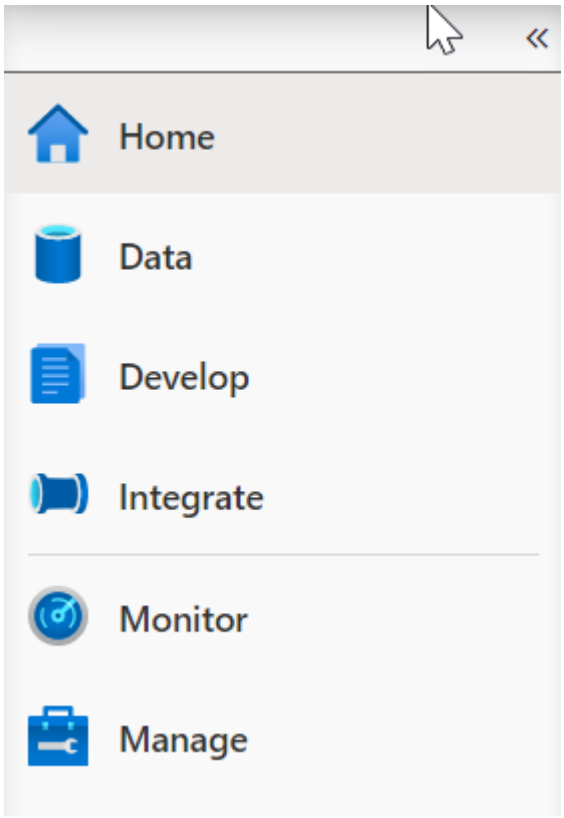
Results | Messages

View: Table | Chart — Export results

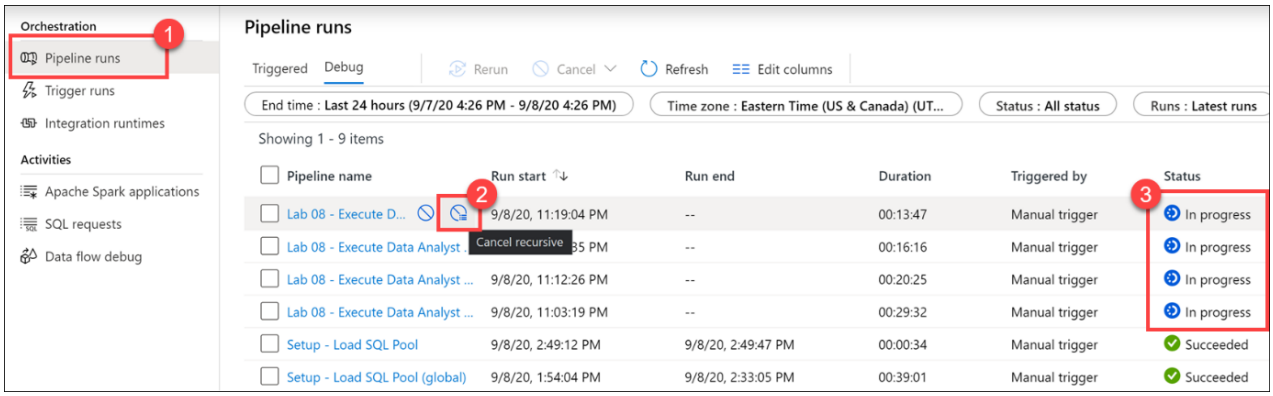
Search

Login_name	Status	Importance
asa.sql.workload02	Running	below_normal
asa.sql.workload02	Running	below_normal
asa.sql.workload02	Running	below_normal
asa.sql.workload02	Running	below_normal
asa.sql.workload02	Running	below_normal

18. Select the **Monitor** hub.



19. Select **Pipeline runs (1)**, and then select **Cancel recursive (2)** for each running Lab 08 pipelines, marked **In progress (3)**. This will help speed up the remaining tasks.



Next unit: Use Azure Advisor to review recommendations

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

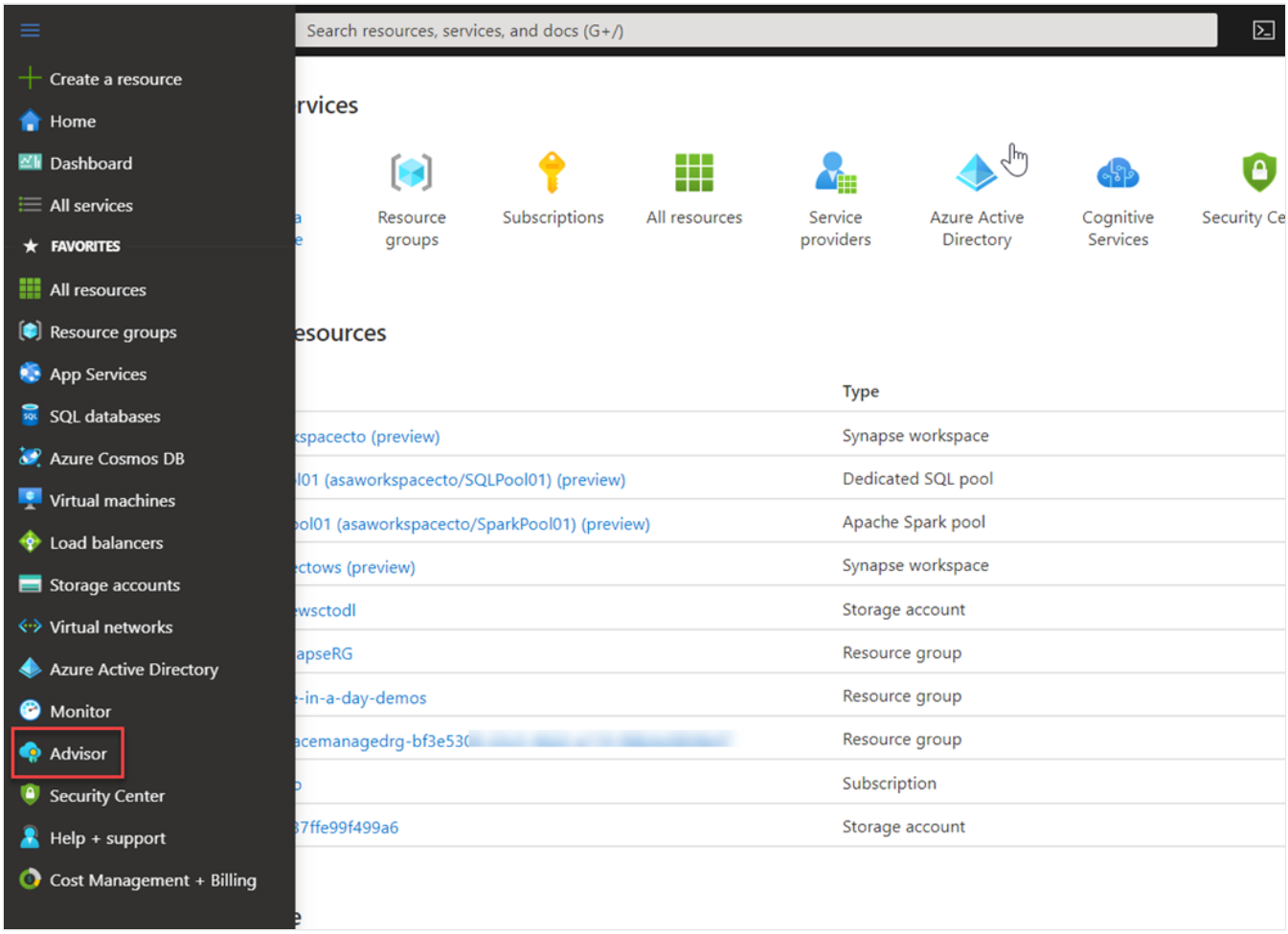
100 XP

Use Azure Advisor to review recommendations

9 minutes

Azure Advisor provides you with personalized messages that provide information on best practices to optimize the setup of your Azure services. It analyzes your resource configuration and usage telemetry and then recommends solutions that can help you improve the cost effectiveness, performance, Reliability (formerly called High availability), and security of your Azure resources.

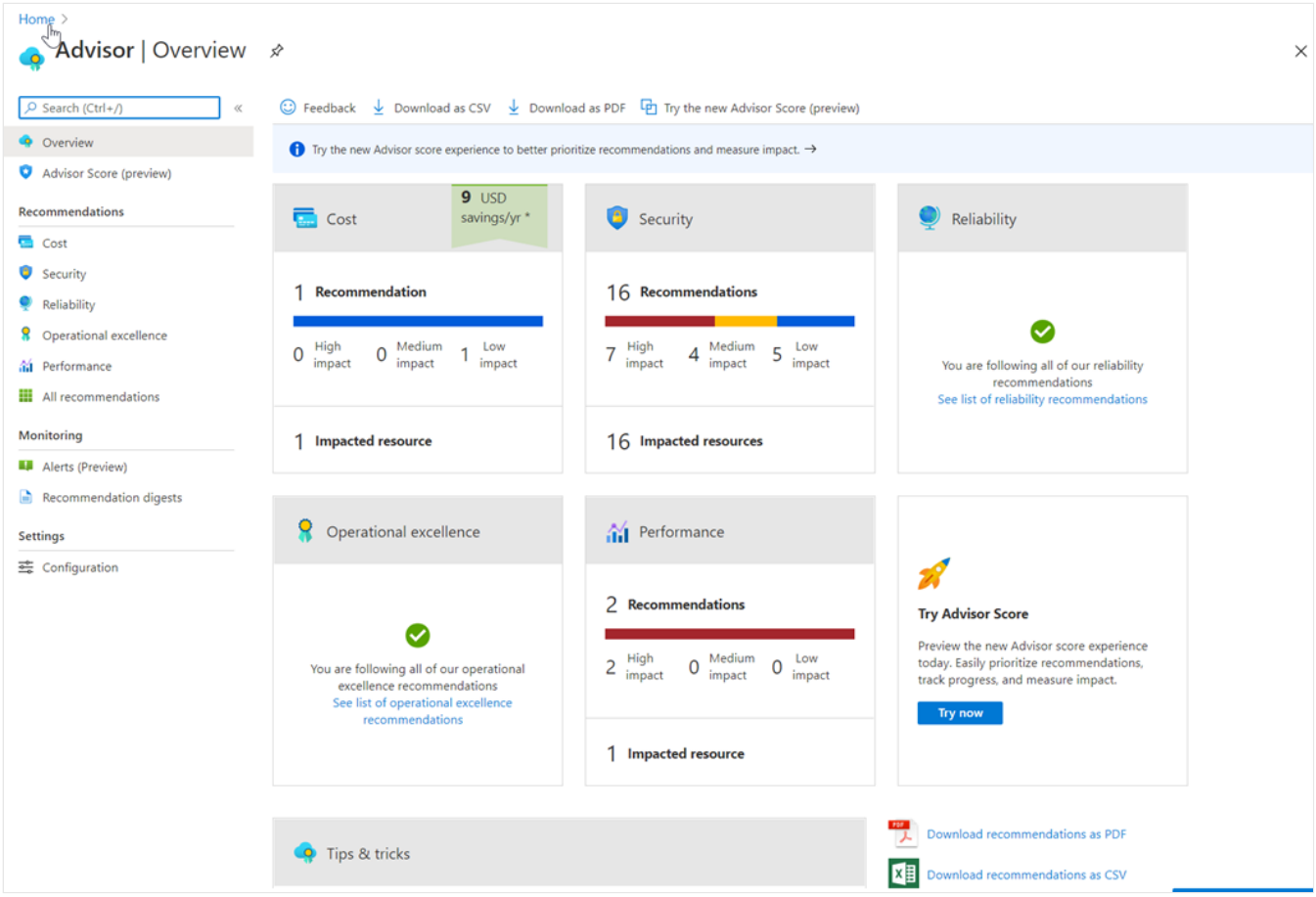
The Advisor may appear when you log into the Azure portal, but you can also access the Advisor by selecting Advisor in the navigation menu.



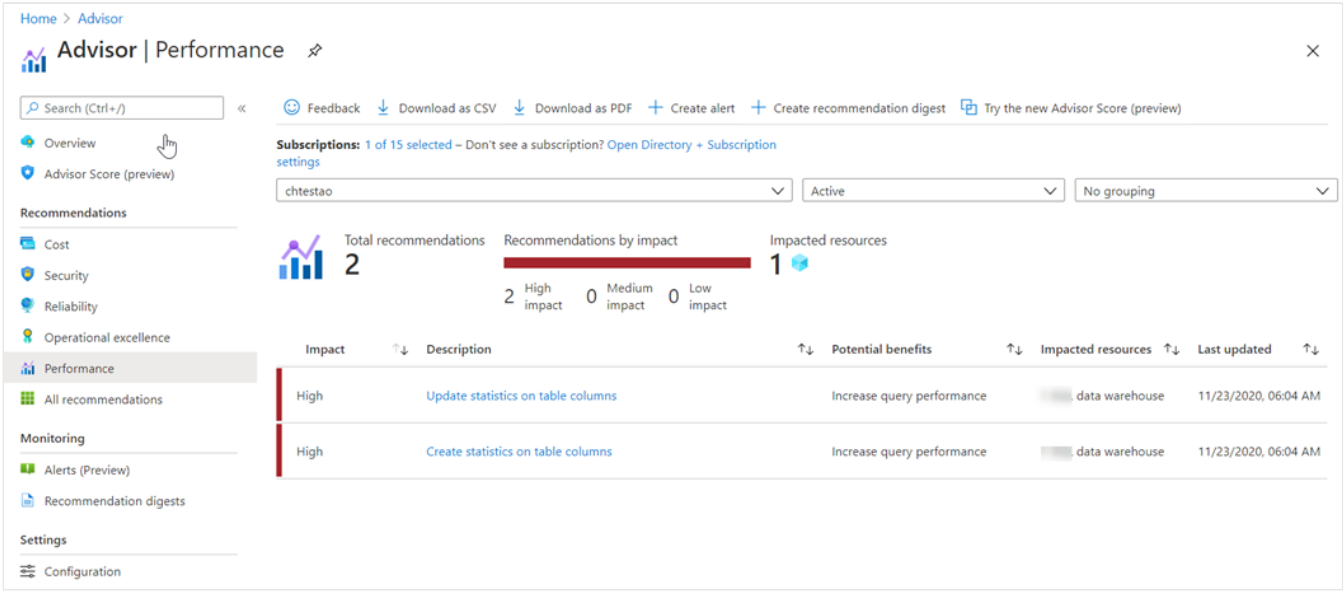
On accessing Advisor, a dashboard is presented that provides recommendations in the following areas:

- Cost
- Security

- Reliability
- Operational excellence
- Performance

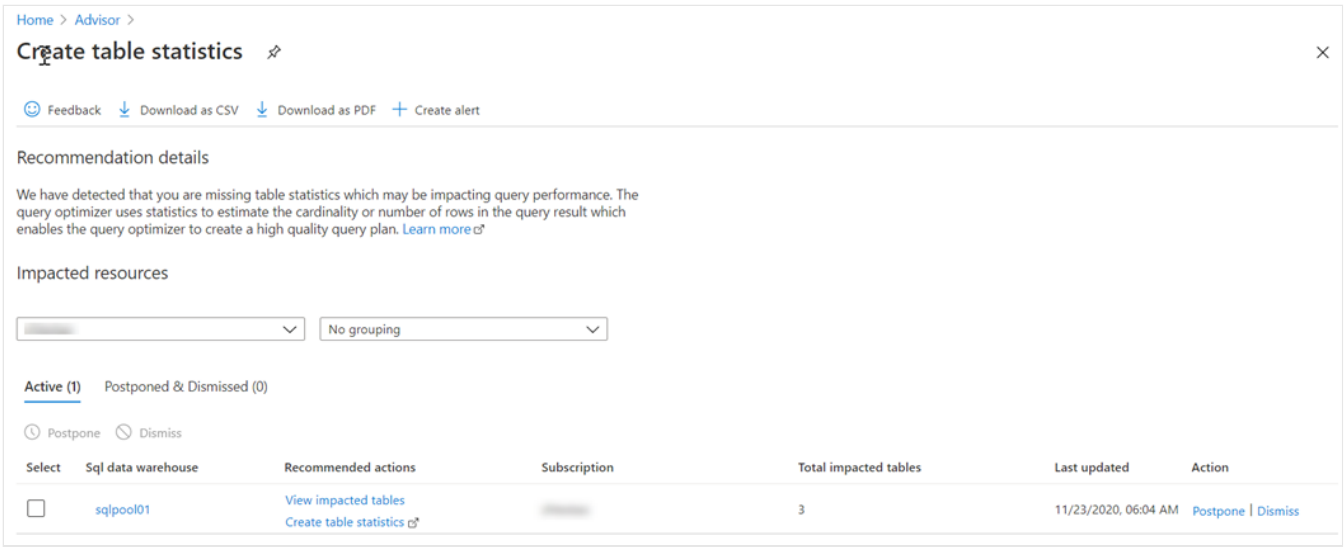


You can click on any of the dashboard items for more information. In the following example, the performance dashboard item is showing more information on two high impact items in Azure Synapse Analytics.



You can also click on each item to get even more information that can help you resolve the issue. In the following example, this is the information that is shown when clicking on the

Create statistics on table columns recommendation.

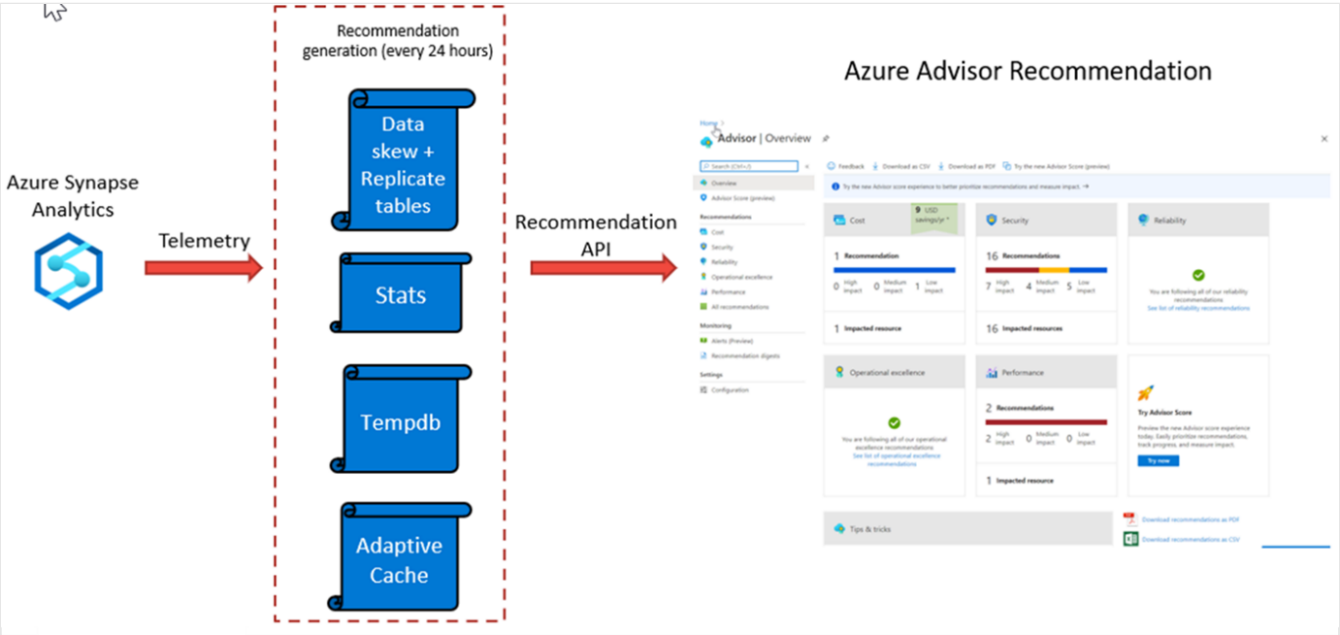


In this screen, you can click on the **view impacted tables** to see which tables are being impacted specifically, and there are also links to the help in the Azure documentation that you can use to get more understanding of the issue.

How Azure Synapse Analytics works with Azure Advisor

Azure Advisor recommendations are free, and the recommendations are based on telemetry data that is generated by Azure Synapse Analytics. The telemetry data that is captured by Azure Synapse Analytics include

- Data Skew and replicated table information.
- Column statistics data.
- TempDB utilization data.
- Adaptive Cache.



Azure Advisor recommendations are checked every 24 hours, as the recommendation API is queried against the telemetry generated from with Azure Synapse Analytics, and the recommendation dashboards are then updated to reflect the information that the telemetry has generated. This can then be viewed in the Azure Advisor dashboard.

Next unit: Use dynamic management views to identify and troubleshoot query performance

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Use dynamic management views to identify and troubleshoot query performance

8 minutes

Dynamic Management Views provide a programmatic experience for monitoring the Azure Synapse Analytics SQL pool activity by using the Transact-SQL language. The views that are provided, not only enable you to troubleshoot and identify performance bottlenecks with the workloads working on your system, but they are also used by other services such as Azure Advisor to provide recommendations about Azure Synapse Analytics.

There are over 90 Dynamic Management Views that can queried against dedicated SQL pools to retrieve information about the following areas of the service:

- Connection information and activity
- SQL execution requests and queries
- Index and statistics information
- Resource blocking and locking activity
- Data movement service activity
- Errors

The following is an example of monitoring query execution of the Azure Synapse Analytics SQL pools. The first step involves checking the connections against the server first, before checking the query execution activity.

Monitoring connections

All logins to your data warehouse are logged to sys.dm_pdw_exec_sessions. The session_id is the primary key and is assigned sequentially for each new logon.

SQL

```
-- Other Active Connections
SELECT * FROM sys.dm_pdw_exec_sessions where status <> 'Closed' and ses-
sion_id <> session_id();
```

Monitor query execution

All queries executed on SQL pool are logged to sys.dm_pdw_exec_requests. The request_id uniquely identifies each query and is the primary key for this DMV. The request_id is assigned sequentially for each new query and is prefixed with QID, which stands for query ID. Querying this DMV for a given session_id shows all queries for a given logon.

Step 1

The first step is to identify the query you want to investigate

SQL

```
-- Monitor active queries
SELECT *
FROM sys.dm_pdw_exec_requests
WHERE status not in ('Completed','Failed','Cancelled')
  AND session_id <> session_id()
ORDER BY submit_time DESC;

-- Find top 10 queries longest running queries
SELECT TOP 10 *
FROM sys.dm_pdw_exec_requests
ORDER BY total_elapsed_time DESC;
```

From the preceding query results, **note the Request ID** of the query that you would like to investigate.

Queries in the **Suspended** state can be queued due to a large number of active running queries. These queries also appear in the sys.dm_pdw_waits waits query with a type of UserConcurrencyResourceType. For information on concurrency limits, see Memory and concurrency limits or Resource classes for workload management. Queries can also wait for other reasons such as for object locks. If your query is waiting for a resource, see Investigating queries waiting for resources further down in this article.

To simplify the lookup of a query in the sys.dm_pdw_exec_requests table, use LABEL to assign a comment to your query, which can be looked up in the sys.dm_pdw_exec_requests view.

SQL

```
-- Query with Label
SELECT *
FROM sys.tables
OPTION (LABEL = 'My Query')
;
```

```
-- Find a query with the Label 'My Query'
-- Use brackets when querying the label column, as it it a key word
SELECT  *
FROM    sys.dm_pdw_exec_requests
WHERE   [label] = 'My Query';
```

Step 2

Use the Request ID to retrieve the queries distributed SQL (DSQL) plan from sys.dm_pdw_request_steps

SQL

```
-- Find the distributed query plan steps for a specific query.
-- Replace request_id with value from Step 1.

SELECT * FROM sys.dm_pdw_request_steps
WHERE request_id = 'QID####'
ORDER BY step_index;
```

When a DSQL plan is taking longer than expected, the cause can be a complex plan with many DSQL steps or just one step taking a long time. If the plan is many steps with several move operations, consider optimizing your table distributions to reduce data movement.

The Table distribution article explains why data must be moved to solve a query. The article also explains some distribution strategies to minimize data movement.

To investigate further details about a single step, the operation_type column of the long-running query step and note the **Step Index**:

- Proceed with Step 3 for **SQL operations**: OnOperation, RemoteOperation, ReturnOperation.
- Proceed with Step 4 for **Data Movement operations**: ShuffleMoveOperation, BroadcastMoveOperation, TrimMoveOperation, PartitionMoveOperation, MoveOperation, CopyOperation.

Step 3

Use the Request ID and the Step Index to retrieve details from sys.dm_pdw_sql_requests, which contains execution information of the query step on all of the distributed databases.

SQL

```
-- Find the distribution run times for a SQL step.
-- Replace request_id and step_index with values from Step 1 and 3.

SELECT * FROM sys.dm_pdw_sql_requests
WHERE request_id = 'QID####' AND step_index = 2;
```

When the query step is running, DBCC PDW_SHOWEXECUTIONPLAN can be used to retrieve the SQL Server estimated plan from the SQL Server plan cache for the step running on a particular distribution.

SQL

```
-- Find the SQL Server execution plan for a query running on a specific SQL
pool or control node.
-- Replace distribution_id and spid with values from previous query.

DBCC PDW_SHOWEXECUTIONPLAN(1, 78);
```

Step 4

Use the Request ID and the Step Index to retrieve information about a data movement step running on each distribution from sys.dm_pdw_dms_workers.

SQL

```
-- Find information about all the workers completing a Data Movement Step.
-- Replace request_id and step_index with values from Step 1 and 3.

SELECT * FROM sys.dm_pdw_dms_workers
WHERE request_id = 'QID####' AND step_index = 2;
```

- Check the total_elapsed_time column to see if a particular distribution is taking longer than others for data movement.
- For the long-running distribution, check the rows_processed column to see if the number of rows being moved from that distribution is larger than others. If so, this finding might indicate skew of your underlying data. One cause for data skew is distributing on a column with many NULL values (whose rows will all land in the same distribution). Prevent slow queries by avoiding distribution on these types of columns or filtering your query to eliminate NULLs when possible.

If the query is running, you can use DBCC PDW_SHOWEXECUTIONPLAN to retrieve the SQL Server estimated plan from the SQL Server plan cache for the currently running SQL Step within a particular distribution.

SQL

```
-- Find the SQL Server estimated plan for a query running on a specific SQL
pool Compute or control node.
-- Replace distribution_id and spid with values from previous query.

DBCC PDW_SHOWEXECUTIONPLAN(55, 238);
```

Dynamic Management Views (DMV) only contains 10,000 rows of data. On heavily utilized systems this means that data held in this table may be lost with hours, or even minutes as data is managed in a first in, first out system. As a result you can potentially lose meaningful information that can help you diagnose query performance issues on your system. In this situation, you should use the Query Store.

You can also monitor additional aspects of Azure Synapse SQL pools including:

- Monitoring waits
- Monitoring tempdb
- Monitoring memory
- Monitoring transaction log
- Monitoring PolyBase

You can view information about monitoring these areas here

Next unit: Knowledge check

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

200 XP

Knowledge check

3 minutes

1. Which ALTER DATABASE statement parameter allows a dedicated SQL pool to scale? *

- SCALE.
  - ✗ Incorrect. SCALE is not a valid ALTER DATABASE parameter
- MODIFY
  - ✓ Correct. MODIFY is used to scale a dedicated SQL pool.
- CHANGE.

2. Which workload management feature influences the order in which a request gets access to resources? *

- Workload classification.
  - ✗ Incorrect. Workload classification allows workload policies to be applied to requests through assigning resource classes and importance..
- Workload importance.
  - ✓ Correct. Workload importance influences the order in which a request gets access to resources. On a busy system, a request with higher importance has first access to resources.
- Workload isolation.

3. Which Dynamic Management View enables the view of the active connections against a dedicated SQL pool? *

- sys.dm_pdw_exec_requests.
  - ✓ Correct. sys.dm_pdw_exec_requests enables you to view the active connections against a dedicated SQL pool
- sys.dm_pdw_dms_workers.

DBCC PDW_SHOWEXECUTIONPLAN.

✕ Incorrect. DBCC PDW_SHOWEXECUTIONPLAN Shows an execution plan of a query, and is not a Dynamic Management View.

Next unit: Summary

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆