Creating the Data Warehouse Schema - 3/4

qimia.io/en/blog/Creating-the-Data-Warehouse-Schema

Welcome back to our series about Data Engineering on MS Azure. In the previous blog's articles, we showed how to set up the infrastructure with Data Engineering on Azure - The Setup and how to pre-process some data with data factory with Basic ETL Processing with Azure Data Factory. This article will cover the creation and configuration of the Synapse DWH, as well as the data architecture and setup of an appropriate star schema.

Synapse SQL Pool

For our Data Warehousing, we use Azure Synapse Analytics(formerly SQL DW).

1. Concepts

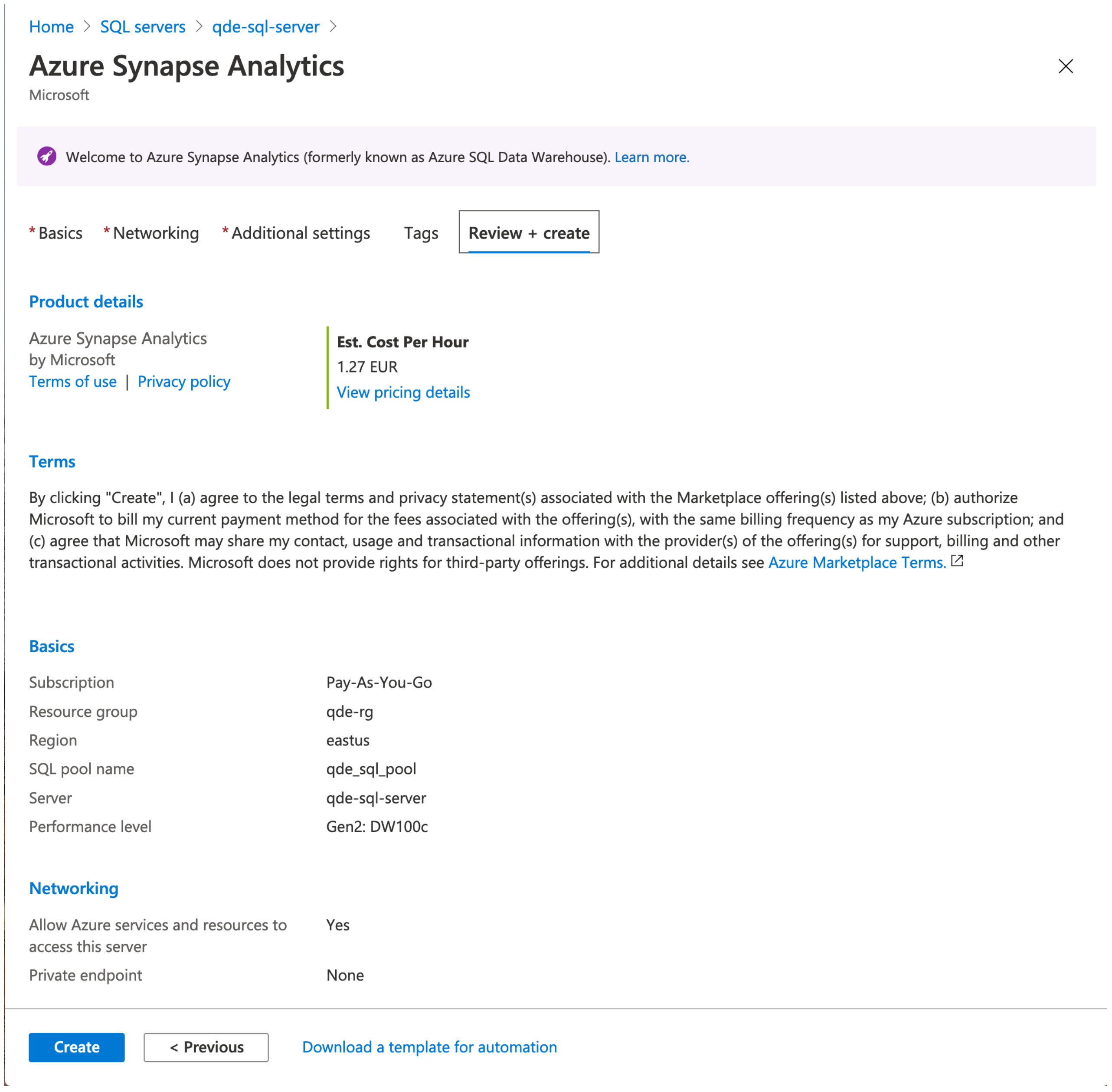
Synapse is the Data Warehousing platform of Azure. It has two versions, the standalone version that is just the SQL pool (called in the Azure Portal "Azure Synapse Analytics (formerly SQL DW)") and the "Synapse Data Analytics (workspaces preview)" integrating the DWH with various other Azure services such PowerBI or Azure Machine Learning, bringing together Spark, SQL, orchestration, and data ingestion. We use the simpler standalone version for this tutorial.

Synapse SQL Pool is Azure's data warehousing solution. It is distributed, scalable, and persistent. The processing power, memory, and pricing of the SQL pool depend on the Data Warehousing Units (DWU), ranging from 100 to 30000.

2. Resource Creation

Go to "Azure Synapse Analytics (formerly SQL DW)" from the Azure Portal and click "Add". Choose the Subscription and the Resource Group that you created in the previous sections and choose a name. We previously created an SQL server for our database, which you can select under "server".

To reduce costs, we chose the smallest instance "DW100c", which is enough for this task, but anything else works too. Our configuration looks like this:


Home > SQL servers > qde-sql-server >

Azure Synapse Analytics

Microsoft

Welcome to Azure Synapse Analytics (formerly known as Azure SQL Data Warehouse). Learn more.

Product details

Azure Synapse Analytics
by Microsoft
Terms of use | Privacy policy

Est. Cost Per Hour
1.27 EUR
View pricing details

Terms

By clicking "Create", I (a) agree to the legal terms and privacy statement(s) associated with the Marketplace offering(s) listed above; (b) authorize Microsoft to bill my current payment method for the fees associated with the offering(s), with the same billing frequency as my Azure subscription; and (c) agree that Microsoft may share my contact, usage and transactional information with the provider(s) of the offering(s) for support, billing and other transactional activities. Microsoft does not provide rights for third-party offerings. For additional details see Azure Marketplace Terms.

Basics

Subscription	Pay-As-You-Go
Resource group	qde-rg
Region	eastus
SQL pool name	qde_sql_pool
Server	qde-sql-server
Performance level	Gen2: DW100c

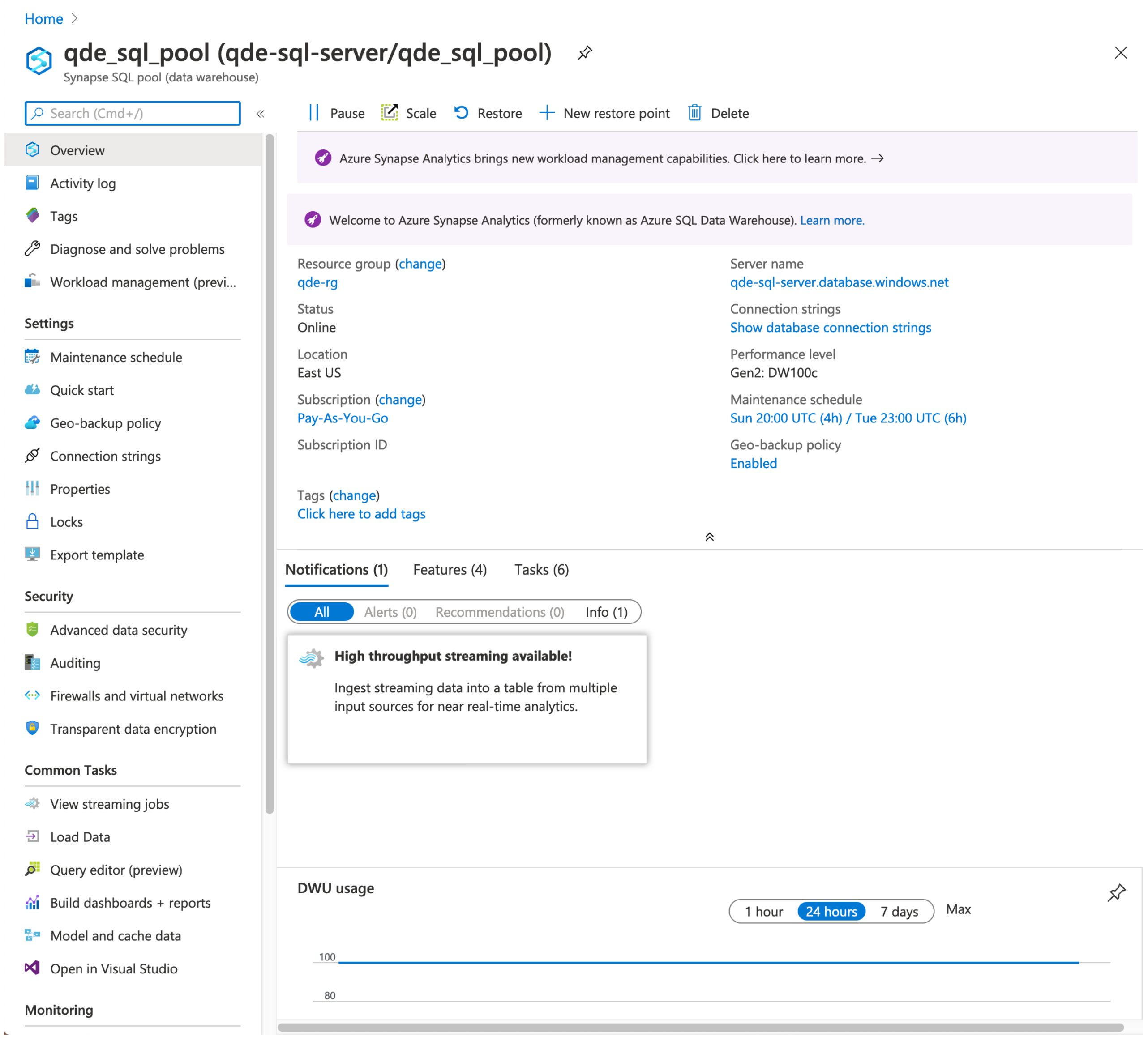
Networking

Allow Azure services and resources to access this server	Yes
Private endpoint	None

Azure Snaps Analytics

3. Database Setup

Search for the name you chose in the Azure Portal and go to the resource.


Home >

qde_sql_pool (qde-sql-server/qde_sql_pool)

Synapse SQL pool (data warehouse)

Search (Cmd+/)

Pause | Scale | Restore | New restore point | Delete

- Overview
- Activity log
- Tags
- Diagnose and solve problems
- Workload management (previ...

Settings

- Maintenance schedule
- Quick start
- Geo-backup policy
- Connection strings
- Properties
- Locks
- Export template

Security

- Advanced data security
- Auditing
- Firewalls and virtual networks
- Transparent data encryption

Common Tasks

- View streaming jobs
- Load Data
- Query editor (preview)
- Build dashboards + reports
- Model and cache data
- Open in Visual Studio

Monitoring

Azure Synapse Analytics brings new workload management capabilities. Click here to learn more. →

Welcome to Azure Synapse Analytics (formerly known as Azure SQL Data Warehouse). Learn more.

Resource group (change)
qde-rg

Status
Online

Location
East US

Subscription (change)
Pay-As-You-Go

Subscription ID

Tags (change)
Click here to add tags

Server name
qde-sql-server.database.windows.net

Connection strings
Show database connection strings

Performance level
Gen2: DW100c

Maintenance schedule
Sun 20:00 UTC (4h) / Tue 23:00 UTC (6h)

Geo-backup policy
Enabled

Notifications (1) | Features (4) | Tasks (6)

All | Alerts (0) | Recommendations (0) | Info (1)

High throughput streaming available!

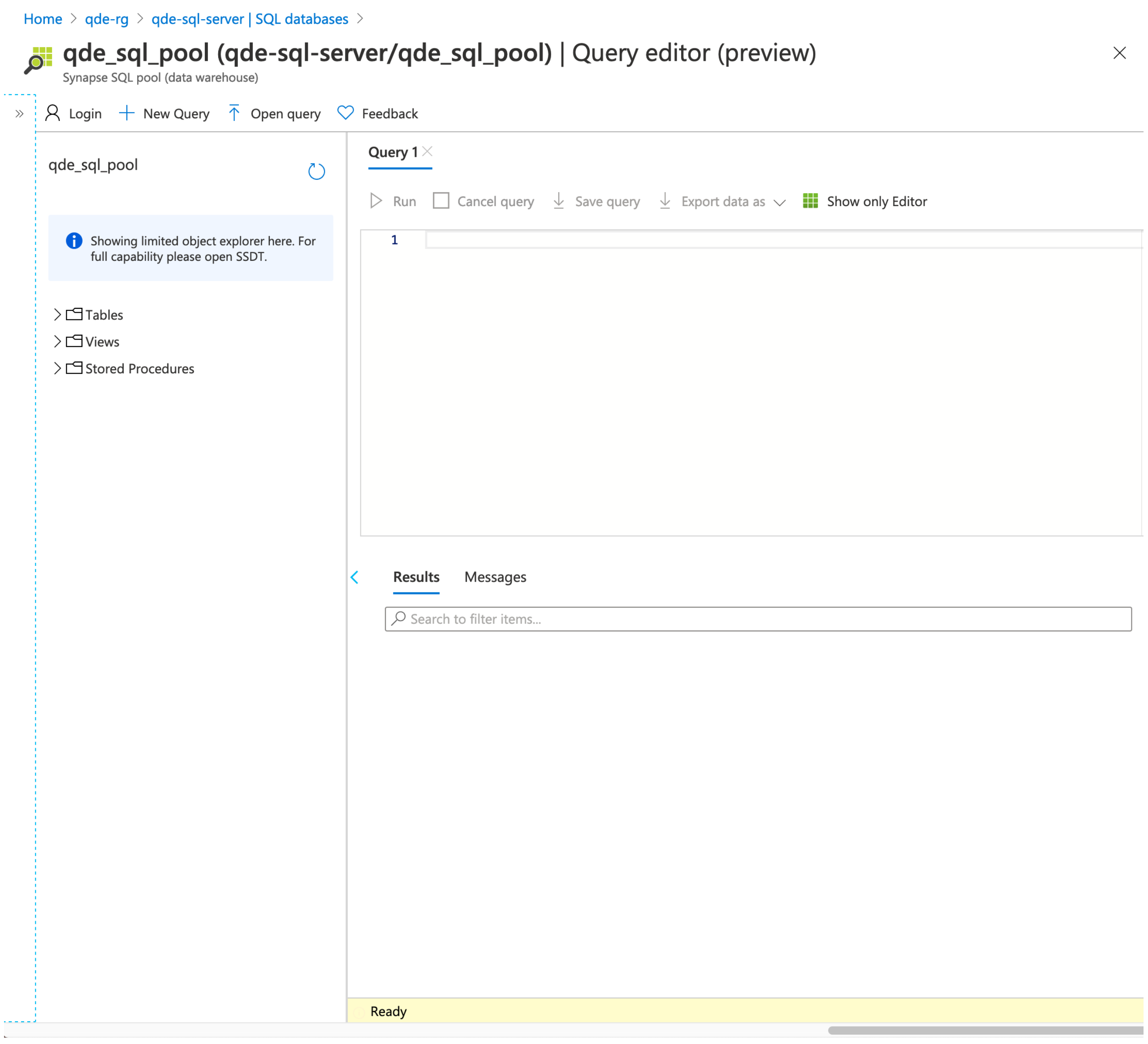
Ingest streaming data into a table from multiple input sources for near real-time analytics.

DWU usage

1 hour | 24 hours | 7 days | Max

SQL Pool Portal

On the left pane, go to "Query editor", this will take you to a Web IDE for your SQL Pool.


Home > qde-rg > qde-sql-server | SQL databases >
qde_sql_pool (qde-sql-server/qde_sql_pool) | Query editor (preview)
Synapse SQL pool (data warehouse)
Login
New Query
Open query
Feedback
qde_sql_pool
Showing limited object explorer here. For full capability please open SSDT.
Tables
Views
Stored Procedures
Query 1
Run
Cancel query
Save query
Export data as
Show only Editor
Results
Messages
Search to filter items...
Ready

Synapse UI
We can start by creating a schema called northwind by running CREATE SCHEMA NORTHWIND;

The first star schema we create is one answering the questions around the employees. In the master database, an employee can have another employee as their supervisor. We would like to answer different employee or supervisor specific business questions here. We define a supervisor as *an employee* who supervises at least one other employee. Thus we have an inheritance dependency here.

Every supervisor is an employee but not every employee is a supervisor. We design a star schema with only one dimension DimEmployee, which contains basic information found in the master database; FactEmployee which defines some employee-specific aggregated information; FactEmployeeMonthly, similar to FactEmployee but defines monthly statistics; and finally FactSupervisor, which only includes the employees who are also supervisors, containing supervisor-specific statistics.

All the tables in our DWH will include columns called key for defining the dependencies between the dimension and fact tables. We deliberately avoid using the ID columns found in the master database, as in the future a table may have multiple IDs for the same record due to updates in the data. Also, we want to differentiate the cross-table relationship between the master database and our DWH.

We define DimEmployee as follows:

```
create table [northwind].[DimEmployee](
    [key] bigint UNIQUE NOT ENFORCED,
    [employee_id] int,
    [last_name] nvarchar (20) NOT NULL,
    [first_name] nvarchar (10) NOT NULL,
    [birth_date] datetime NULL,
    [hire_date] datetime NULL,
    [address] nvarchar (60) NULL ,
    [city] nvarchar (15) NULL ,
    [region] nvarchar (15) NULL ,
    [postal_code] nvarchar (10) NULL ,
    [country] nvarchar (15) NULL,
    [created_ts] datetime
    )WITH(
         DISTRIBUTION = HASH ([key]),
    CLUSTERED COLUMNSTORE INDEX ORDER ([key])
    )
;

CREATE INDEX name_index ON [northwind].[DimEmployee] ([last_name]);
```

The columns we add here are the created_ts — the time the record was created and key, the unique identification of the row. In the second part we see options DISTRIBUTION and CLUSTERED COLUMNSTORE INDEX ORDER ([key]). These are the important part of a distributed database like Synapse.

3.1. SQL Pool Data Distribution

Synapse SQL Pool is a distributed platform that stores its data in individual nodes. There are several strategies for distributing the data across distributions. Regardless of the selected Performance Level, there are 60 total Distributions. For example, the Performance Level DW1500c, which has 3 Compute nodes, has 60/3=20 data distributions for each of its compute nodes. The data inserted into tables are kept in the following ways.

3.1.1. Hash Distribution

This is the distribution used above, where the data is distributed according to the hash value of a column. The modulo 60 of this hash value indicates the distribution to put the data on. This distribution has the advantage of high-efficiency when filtering or joining the data — if used correctly. For example, if two tables are hash distributed on the same join column, this operation would be colocated on the same distribution.

The hash column must be selected carefully to not cause data skewness. If the hash of the distribution column does not end up in a uniform distribution across the nodes, the data will be skewed. Some nodes get too much data to handle while some will be under-utilized.

3.1.2. Round-Robin Distribution

Each data point is distributed to the nodes in Round-robin fashion. Every new data point is inserted into the distributions in the circular order. This ensures the data is evenly distributed. The downside is the lack of logical data-dependent distribution which may cause too much shuffling with joins.

3.1.3. Replication

Replication is a method suitable for small tables; basically, each data point is saved in all the distributions. If the table is a small dimension table, this is very efficient when joining to the fact table. However, if the table is large, this may cause storage issues.

3.2. Data Indexing

An important difference of Synapse SQL Pool to other RDBMSes is its lack of primary and foreign key constraints. Although they exist, Synapse doesn't enforce them. This is due to the distributed nature of Synapse: every node in the cluster manages its own and enforcing these would require synchronization between the nodes. Since this is not feasible with such a system, the developer must ensure the primary keys are unique, and the foreign keys are set correctly.

In a typical RDBMS, the indexes are implemented with b-trees. Although these data structures are suitable for accessing individual rows, they are not as performant with aggregate operations and joins. Synapse introduces several ways to index data.

3.2.1. Clustered Columnstore Indexes

In this mode, the data is stored column-oriented. The data is stored in sections and keeps the range of its minimum and maximum values for each column. This way queries may be more efficient and can filter out out-of-range segments.

However, since there is no ordering by default, there will be overlaps between segment ranges. You can choose to order the data based on one or more columns, reducing the overlap between segments. The ordering also significantly improves the search over the data as binary search is of O(log n) complexity. Since the data is columnar and ordered, compression algorithms save a lot of space. If there's only a small amount of data, other indexes may be preferable, otherwise, it is safe to say that clustered columnstore indexes are the best choice.

Note: Unfortunately, ordering on string columns isn't supported; we recommend using hash values of the string columns instead.

3.2.2. Clustered vs Nonclustered Indexes

These are indexes for row oriented data. The clustered index describes the way the data is stored while nonclustered index is an external rowstore index referencing the rows in the original table.

3.2.3. Heap Index

Heap tables are mostly suggested when writing tables that are small and temporary. As no ordering is required, loading data into the heap is much faster. However, this means if the number of rows is too high, querying the table takes a lot of time. A common use case would be to quickly insert data to a heap and later changing the index to a clustered index. For example, a table that is used for staging can be used; : write everything once, read everything once, no other operations such as joins.

3.3. Back to the Schema

We created FactEmployee, FactSupervisor, and FactEmployeeMonthly as follows:

```
create table [northwind].[FactEmployee](
    [key] bigint,
    [employee_key] bigint, -- refers to the  records in the dim_employee table
    [employee_id] bigint,
    [total_distinct_territories] int,
    [total_distinct_regions] int,
    [num_orders_affiliated] int,
    [num_products_affiliated] int,
    [created_ts] datetime
)WITH(
         DISTRIBUTION = HASH ([employee_key]),
    CLUSTERED COLUMNSTORE INDEX ORDER ([employee_key])
    );

-- People who supervise at least one person.
create table [northwind].[FactSupervisor](
    [key] bigint,
    [employee_id] bigint,
    [employee_key] bigint, -- refers to the  records in the dim_employee table
    [num_employees_directly_supervised] int,
    [created_ts] datetime
)WITH(
         DISTRIBUTION = HASH ([employee_key]),
    CLUSTERED COLUMNSTORE INDEX ORDER ([employee_key])
    )
;


create table [northwind].[FactEmployeeMonthly](
    [key] bigint, -- hash(employee_id, year, month)
    [employee_key] bigint, -- refers to the  records in the dim_employee table
    [employee_id] bigint,
    [year] int,
    [month] int,
    [total_products] int,
    [total_distinct_products] int,
    [total_orders] int,
    [created_ts] datetime
)WITH(
     DISTRIBUTION = HASH ([employee_key]),
    CLUSTERED COLUMNSTORE INDEX ORDER ([employee_key])
    )
;
```

This can be thought of as polymorphism in SQL. For example, FactEmployee, FactSupervisor, and FactEmployeeMonthly tables refer to a single row on DimEmployee, inheriting all the properties of a basic employee. All the fact tables we defined to have the same distribution and indexes. Let's take FactSupervisor as an example. We define the distribution as the Hash of the employee_key which is used to join on DimEmployee(key). Since DimEmployee is also distributed on the same column, this means the join between the two will be colocated on the same node. To further improve the join efficiency, we added CLUSTERED COLUMNSTORE INDEX ORDER ([key]) on the DimEmployee. This means when left joining between the fact and dimension table, the records will be searched much faster.

Moreover, we added UNIQUE NOT ENFORCED constraint on the key column of DimEmployee. The uniqueness property of the column is not enforced, the developer must ensure the values are unique. However, Synapse engine can use this constraint to optimize the queries when searching over the data.

To DimEmployee, we also added a nonclustered index with CREATE INDEX name_index ON [northwind].[DimEmployee] ([last_name]). In most business cases, the employee stats may be searched by filtering on the name. This allows speeding-up of the search on the name column of the dimension. The Fact tables which refer to the dimension table can then easily be filtered as they have a clustered columnstore index on the employee_key.

3.3.1. FactCustomer

We have another table called FactCustomer without any dimension tables.

```
CREATE TABLE [northwind].[FactCustomer](
    [key] bigint,
    [customer_id] nvarchar(100),
    [company_name] nvarchar(100),
    [contact_name] nvarchar(100),
    [contact_title] nvarchar(100),
    [address] nvarchar(100),
    [city] nvarchar(100),
    [region] nvarchar(100),
    [country] nvarchar(100),
    [phone] nvarchar(100),
    [fax] nvarchar(100),
    [total_orders] int,
    [total_raw_price] float,
    [total_discount] float,
    [average_discount] float,
    [created_ts] datetime
)WITH(
     DISTRIBUTION = ROUND_ROBIN,
    CLUSTERED COLUMNSTORE INDEX
    )
;
CREATE INDEX company_name_index     ON [northwind].[FactCustomer]
([company_name]);
CREATE INDEX contact_name_index     ON [northwind].[FactCustomer]
([contact_name]);
CREATE INDEX contact_title_index    ON [northwind].[FactCustomer]
([contact_title]);
```

This table isn't likely to be joined with other tables, so we don't need to add a columnstore index ordering on any specific column. We neither need a key to distribute on, so we use ROUND_ROBIN distribution, which will ensure evenness among nodes. Since it might be necessary to search through data based on company_name, contact_name, or contact_title, we created additional indexes for speeding up search over these specific fields. These indexes are nonclustered, external to the data, and won't change the order or distribution of it. However, every time data is inserted into this table, these indexes will be automatically updated; increasing the insert time.

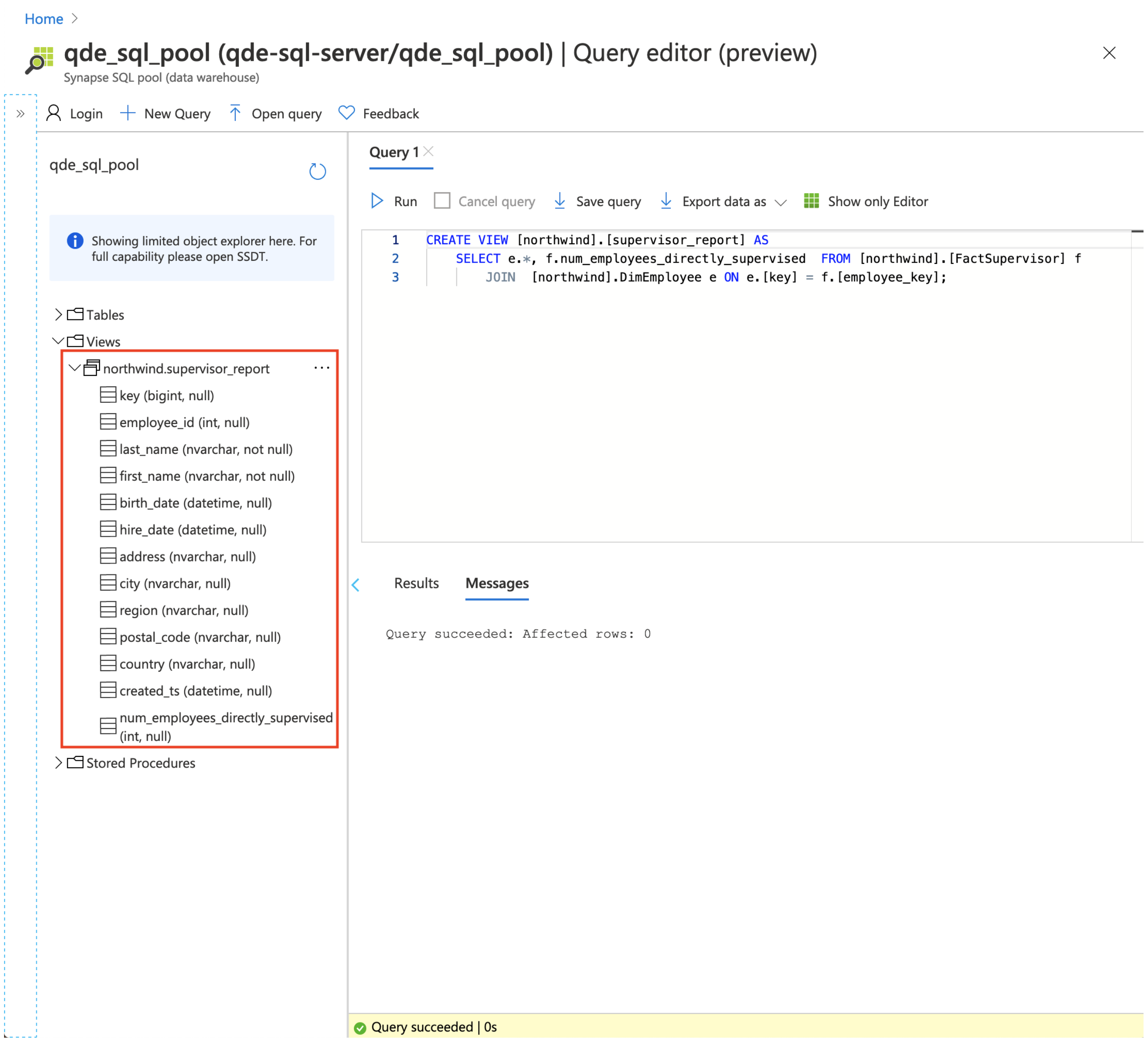
3.4. Preparing Business Report Views

To be able to take advantage of the star schemas, the fact and dimension tables must be joined together to show data. However, the user may not want to join the tables every time they want to query the data. We can use views to automate this for convenience.

Let's say we want to get a report on the Supervisors; then we can use the FactSupervisor and its relation `DimEmployee`` We create the views as follows;

```
CREATE VIEW [northwind].[supervisor_report] AS
    SELECT e.*, f.num_employees_directly_supervised  FROM [northwind].
[FactSupervisor] f
        JOIN  [northwind].DimEmployee e ON e.[key] = f.[employee_key];
```

We can see the view we created as follows:


Home >
qde_sql_pool (qde-sql-server/qde_sql_pool) | Query editor (preview)
Synapse SQL pool (data warehouse)
Login
New Query
Open query
Feedback
qde_sql_pool
Showing limited object explorer here. For full capability please open SSDT.
Tables
Views
Stored Procedures
Query 1
Run
Cancel query
Save query
Export data as
Show only Editor
Results
Messages

```
northwind.supervisor_report
    key (bigint, null)
    employee_id (int, null)
    last_name (nvarchar, not null)
    first_name (nvarchar, not null)
    birth_date (datetime, null)
    hire_date (datetime, null)
    address (nvarchar, null)
    city (nvarchar, null)
    region (nvarchar, null)
    postal_code (nvarchar, null)
    country (nvarchar, null)
    created_ts (datetime, null)
    num_employees_directly_supervised (int, null)
```

```
CREATE VIEW [northwind].[supervisor_report] AS
    SELECT e.*, f.num_employees_directly_supervised  FROM [northwind].[FactSupervisor] f
        JOIN  [northwind].DimEmployee e ON e.[key] = f.[employee_key];
```

```
Query succeeded: Affected rows: 0
```

Query succeeded | 0s

SQL Pool View

Anytime a view is retrieved, the underlying query is run again and the result is returned. This mechanism allows the separation of the complexity of the joins or mappings from external mechanisms.

One disadvantage of this mechanism is that the result of the view is not cached. Every time the view is retrieved, the query must be re-run. For caching, materialized views can be used as follows:

```
CREATE MATERIALIZED VIEW [northwind].[supervisor_report] AS
    SELECT e.*, f.num_employees_directly_supervised  FROM [northwind].
[FactSupervisor] f
        JOIN  [northwind].DimEmployee e ON e.[key] = f.[employee_key];
```

It works by saving the result of the query when it is created, rebuilt, or the underlying data is changed. Similar to the tables on the SQL Pool, the materialized views can be customized by their distribution, partitioning, and indexing. Contrary to the other Cloud Data Warehouses like Amazon Redshift, the materialized views on Synapse don't need to be refreshed manually with the new data coming in, this is done automatically by Synapse as the underlying data changes.

However, materialized tables may occasionally be slower than directly running the queries, as the columnstore indexes must be scanned for incremental changes. To re-optimize, the REBUILD command must be used to recreate the materialized view with the new data.

4. Ingesting Data from External Sources

It's typical for a DWH to copy data from an external location such as a datalake to a persistent table. This can be done with the COPY command. To put it into practice let's do a toy example by copying a parquet file we have on our datalake to Synapse.

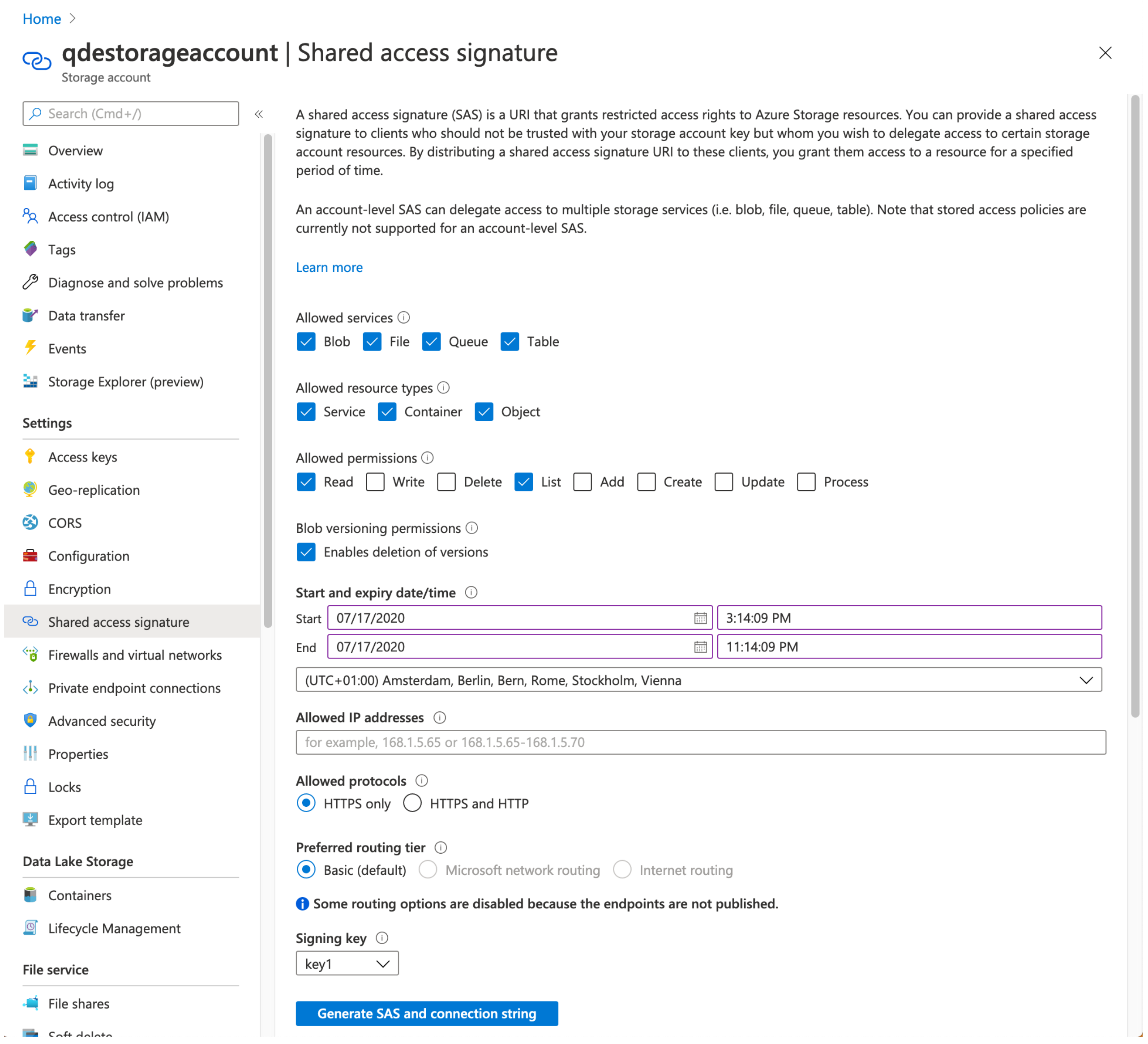
To access a datalake, you will need credentials. We recommend either using a Storage Account Key or a Shared Access Key. Storage Account Key is the master credentials for our storage account. You can grab the value used in the Setup.

```
CREATE TABLE [northwind].[temp_category] (
    [CustomerID] varchar(200),
    [CompanyName] varchar(200),
    [ContactName] varchar(200),
    [ContactTitle] varchar(200),
    [Address] varchar(200),
    [City] varchar(200),
    [Region] varchar(200),
    [PostalCode] varchar(200),
    [Country] varchar(200),
    [Phone] varchar(200),
    [Fax] varchar(200)
)


COPY INTO [northwind].[temp_category](
    [CustomerID],
    [CompanyName],
    [ContactName],
    [ContactTitle],
    [Address],
    [City],
    [Region],
    [PostalCode],
    [Country],
    [Phone],
    [Fax]
)FROM
'https://qdestorageaccount.blob.core.windows.net/northwind/prepared/customers/year=

WITH (FILE_TYPE= 'PARQUET',
    CREDENTIAL=(IDENTITY = 'Storage Account Key',
    SECRET = '<YOUR_STORAGE_ACCOUNT_KEY>')
    )
```

An alternative is to use Shared Access Signature. That is a permission model where you can restrict the permissions on different levels. Moreover, you can also restrict whether from when and until when it can be used. Let's create a read only Shared Access Signature for our Storage Account. Go to your storage account -> Access Keys -> Copy SAS Token. The configuration can be set as follows:



Home >

qdestorageaccount | Shared access signature

Storage account

Search (Cmd+/)

- Overview
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
- Data transfer
- Events
- Storage Explorer (preview)

Settings

- Access keys
- Geo-replication
- CORS
- Configuration
- Encryption
- Shared access signature
- Firewalls and virtual networks
- Private endpoint connections
- Advanced security
- Properties
- Locks
- Export template

Data Lake Storage

- Containers
- Lifecycle Management

File service

- File shares

A shared access signature (SAS) is a URI that grants restricted access rights to Azure Storage resources. You can provide a shared access signature to clients who should not be trusted with your storage account key but whom you wish to delegate access to certain storage account resources. By distributing a shared access signature URI to these clients, you grant them access to a resource for a specified period of time.

An account-level SAS can delegate access to multiple storage services (i.e. blob, file, queue, table). Note that stored access policies are currently not supported for an account-level SAS.

Learn more

Allowed services
✓ Blob ✓ File ✓ Queue ✓ Table

Allowed resource types
✓ Service ✓ Container ✓ Object

Allowed permissions
✓ Read ☐ Write ☐ Delete ✓ List ☐ Add ☐ Create ☐ Update ☐ Process

Blob versioning permissions
✓ Enables deletion of versions

Start and expiry date/time

Start	07/17/2020	3:14:09 PM
End	07/17/2020	11:14:09 PM

(UTC+01:00) Amsterdam, Berlin, Bern, Rome, Stockholm, Vienna

Allowed IP addresses
for example, 168.1.5.65 or 168.1.5.65-168.1.5.70

Allowed protocols
[✓ HTTPS only] HTTPS and HTTP

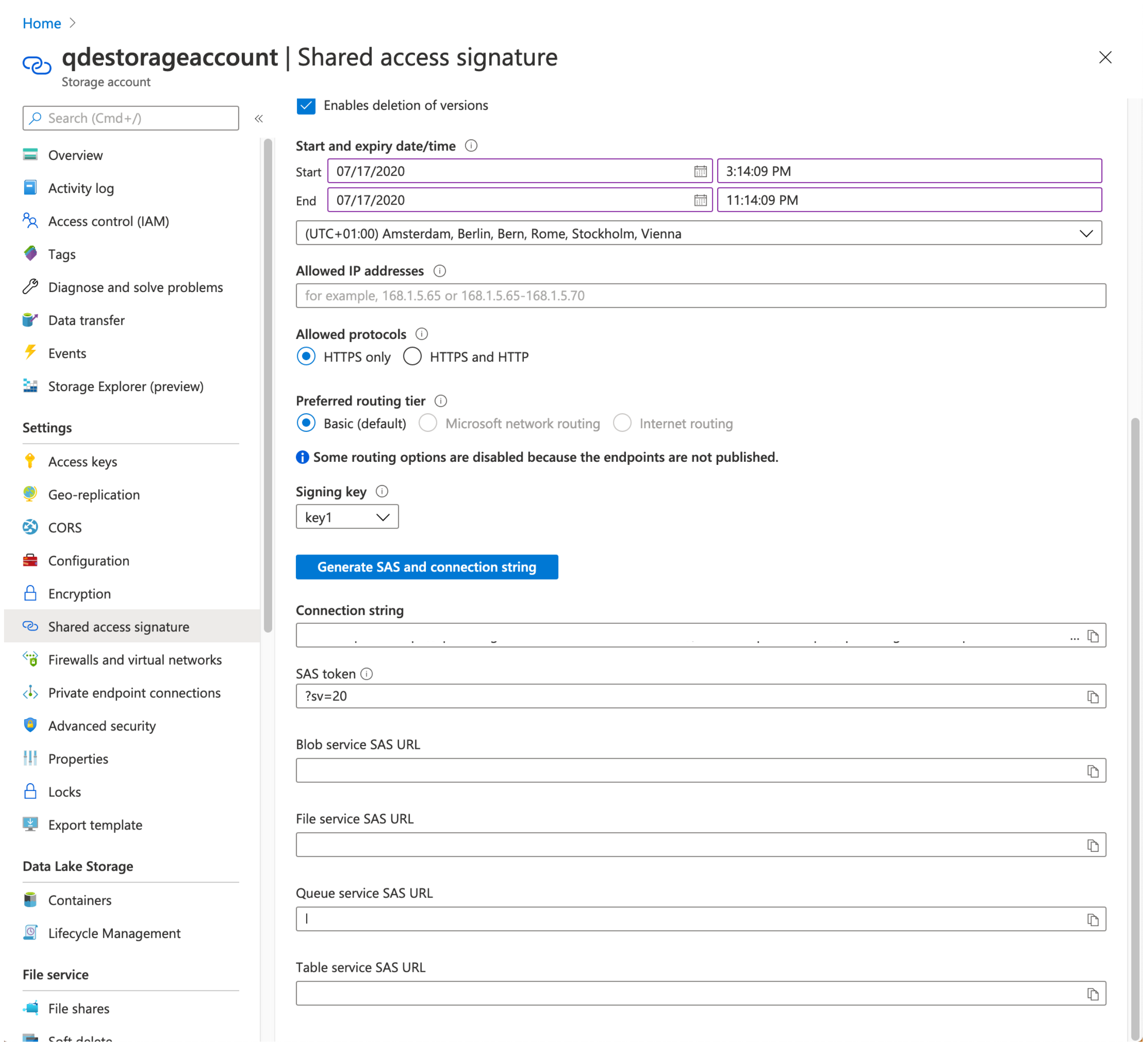
Preferred routing tier
[✓ Basic (default)] Microsoft network routing Internet routing

Some routing options are disabled because the endpoints are not published.

Signing key
key1

Generate SAS and connection string

SAS I
Copy from 'SAS token' below:



qdestorageaccount | Shared access signature

Storage account

[✓] Enables deletion of versions

Start and expiry date/time

Start 07/17/2020 3:14:09 PM

End 07/17/2020 11:14:09 PM

(UTC+01:00) Amsterdam, Berlin, Bern, Rome, Stockholm, Vienna

Allowed IP addresses

for example, 168.1.5.65 or 168.1.5.65-168.1.5.70

Allowed protocols

[✓ HTTPS only] HTTPS and HTTP

Preferred routing tier

[✓ Basic (default)] Microsoft network routing Internet routing

Some routing options are disabled because the endpoints are not published.

Signing key

key1

Connection string

SAS token

?sv=20

Blob service SAS URL

File service SAS URL

Queue service SAS URL

Table service SAS URL

SAS II

You must drop the leading '?' from the token as it may cause permission problems. The copy command can be run with the credentials:

```
with (FILE_TYPE= 'PARQUET',
    CREDENTIAL=(IDENTITY = 'SHARED ACCESS SIGNATURE',
    SECRET = 'YOUR_SAS_TOKEN_WITHOUT_THE_LEADING_?')
    )
```

5. Creating External Tables

Synapse also has the ability to dynamically read the data stored on the data lake. Let's create an external table on the same Parquet file we used earlier. This time the credentials must be stored by Synapse securely. For that, we run CREATE MASTER KEY to create a key to store our secrets encrypted. You must log in as the admin user to run these commands.

5.1. Parquet File Format

First, we create a Parquet file format. This can also be an ORC or CSV parser, but we prefer Parquet for simplicity.

```
CREATE EXTERNAL FILE FORMAT ParquetFormat
    WITH (
    FORMAT_TYPE = PARQUET);
```

5.2. Credentials

Second, the credentials we want to store in Synapse are created. This will be encrypted by the Master KEY of the database. We use the same storage account key as above.

```
CREATE DATABASE SCOPED CREDENTIAL datalake_sak
    WITH IDENTITY = 'Storage Account Key',
    SECRET = '<YOUR_STORAGE_ACCOUNT_KEY>';
```

5.3. Data Source

Then we create a data source. This only needs the credentials and the root location. We provided our container 'northwind' under the storage account 'qdestorageaccount'.

```
CREATE EXTERNAL DATA SOURCE datalakesource
    WITH (
    -- abfss is used for datalake gen2
    LOCATION = 'abfss://northwind@qdestorageaccount.dfs.core.windows.net' ,
    -- Use the credential we just created
    CREDENTIAL = [datalake_sak],
    TYPE=HADOOP) ;
```

5.4. External Table

Finally, we create the external table. We specify our data source which automatically will use the credentials. The file format is specified as the Parquet format we just created. The location is given relative to the root of the data lake container.

```
 CREATE EXTERNAL TABLE [northwind].[ext_category](
      [CustomerID] varchar(200),
      [CompanyName] varchar(200),
      [ContactName] varchar(200),
      [ContactTitle] varchar(200),
      [Address] varchar(200),
      [City] varchar(200),
      [Region] varchar(200),
      [PostalCode] varchar(200),
      [Country] varchar(200),
      [Phone] varchar(200),
      [Fax] varchar(200)
  )
      WITH
          (
          -- The path should be provided relative to the container root in the
DATA_SOURCE
          LOCATION =
'prepared/customers/year=1996/month=7/day=16/dbo.Customers.parquet',
          DATA_SOURCE = [datalakesource],
          FILE_FORMAT = [ParquetFormat]
          )
  ;
```

6. Controlling the Costs

Using Synapse with smaller data (less than ca. 1 TB) doesn't need to be the most efficient. The data is always stored in 60 distributions which can bring unnecessary parallelism overhead with little data and only one compute node.

Also, when you create the cluster, 1TB of space is automatically allocated for you. This means, even if you use 100GB out of 1TB, you pay for 1TB per hour (€ 0,16). When your capacity exceeds 1TB, the storage will automatically be updated to 2TB; then you'll pay 2*(€0,16) per hour. Bear in mind that the database automatically creates snapshots, which are saved in the same storage.

Computing is separate from the storage, so you can scale independently from the storage. When you join tables, the data will be sent between the compute nodes. This will cause network charges. You can reduce the costs by distributing your data in a way that reduces the shuffling.

If you don't need provisioned resources, you can also use the serverless SQL-on-demand of Synapse Workspace. You will be charged by the TB of data that you process (€5/TB). Your tables can only be external though. Unfortunately, the external tables can't discover the Hive partitioning. Use Views with OPENROWSET function instead.

Sources

- Azure Docs: Synapse Best Practices
- Azure Docs: Synapse Indexing

- Azure Docs: Synapse Distributing
- Azure Docs: Synapse Views
- Azure Docs: Synapse Copy statement
- Azure Docs: Synapse External Tables
- MS Azure Synapse Pricing
- Azure Docs: Synapse Architecture

For the next part of the tutorial click here.