# Data Engineering on Azure - The Setup - 1/4

qimia.io/en/blog/data-engineering-microsoft-azure-set-up

## The Setup Introduction

In recent years, Machine Learning has received a lot of attention due to the progress made in research. Many companies would like to leverage the opportunities offered by these techniques or perform Business Analytics but struggle to do so, because their data is locked up in silos like in OLTP databases containing operational data. Historically, there have been two main approaches to utilize data for analytical and Machine Learning purposes:

1. With a Data Warehouse
2. With a Data Lake

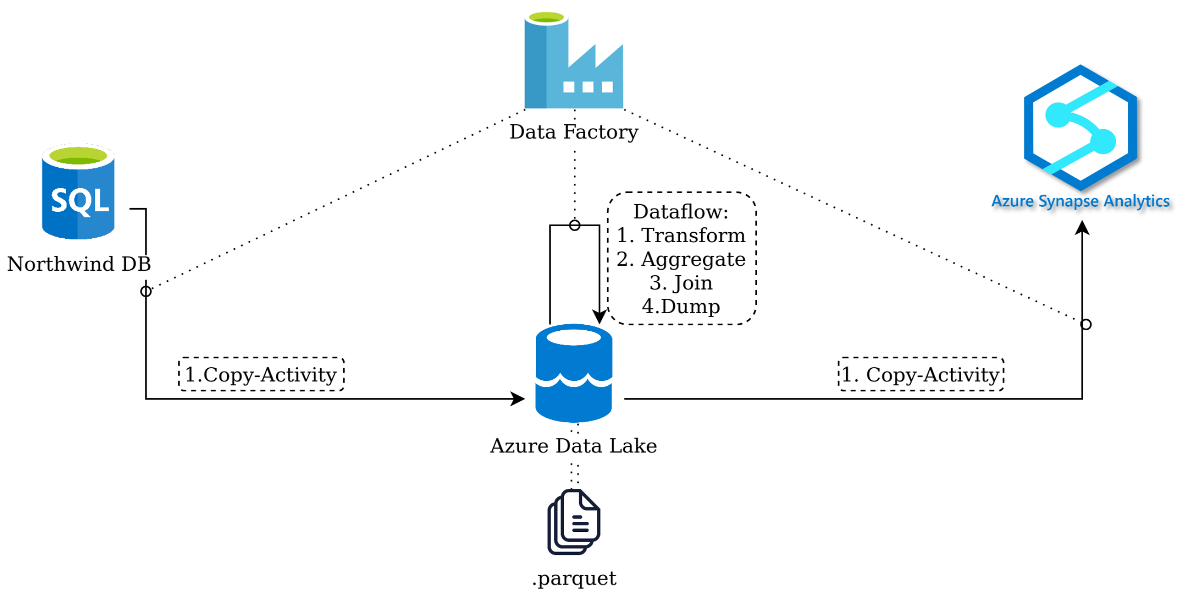
Access to the data is granted by utilizing ETL (Data Warehouse) or ELT (Data Lake) processes. While the concept of a Data Lake is mostly about avoiding to transfer data and keeping it where it is, Data Warehouses focus on transforming and storing data in a way that enables lightning-fast analytical queries. Setting up the appropriate Data Architecture is absolutely crucial to the success of not only classical analytical solutions but also Machine Learning Algorithms. With this short introduction, we would like to welcome you to our tutorial series about Data Engineering on MS Azure!

## Target

In this tutorial, we are going to teach you how to:

- Set up the appropriate resources on Azure and manage their access to each other
- Build pipelines in Data Factory that can read, transform, partition, and write data
- Set up appropriate star schemas in the Azure Data Warehouse Solution 'Synapse'

The series will eventually lead us to the following setup:



Data Engineering on Azure Schema
The *tutorial* will consist of *four parts*, building on each other:

1. Setup and Introduction to the project

2. Load the Northwind Database, batch and dump the data to a Data Lake

3. Design the appropriate star schema for our data based on assumed access patterns

4. Transform, aggregate, join and dump the data to the Data Lake & copy the data to Synapse on a daily basis

This is the first part of our series and the focus of it will be the setup of our starting resources including their most important configuration options. We will prepare for our first pipeline that will transfer data from our SQL database to the Azure Data Lake Storage. We are going to use the well-known Northwind Database.

**If you are not interested in setting up the components by yourself, check out our Azure CLI script on Github. Be aware that you will still have to put the Storage Account Access Key into the Key Vault as a Secret.**

## Setting up the Resources

Before we get started with our first pipeline, we will set up these few things:

1. Resource Group

2. Azure Data Lake Storage Gen 2 (ADLS G2)

3. SQL Database and Database

4. Key Vault

## 1. Resource Group

In Azure, resources are organized into Resource Groups. We will set up a new resource group, where we will store all our resources. To do so, go to your Azure Portal and search for Resource Groups and add a new one.

We will call ours 'qde_rg' for Qimia Data Engineering resource group. We are going to use that prefix throughout the whole tutorial. Make sure to use a region that contains all the services that we need. We are going to use **US East**.

## 2. Data Lake Storage

Next, we will create the Data Lake Storage Gen 2. In Azure, this storage utilizes Blob Storage which runs on storage accounts, so we need to create a new storage account first. From your resource group, add a new resource and search for the storage account. When creating a blob storage, you will be confronted with configuration options.

### 2.1. Read/Write Performance

To reduce the costs, we will go for the **Standard Performance tier**, **General purpose V2 storage**, **LRS,** and **Cool access tier**. Be aware that you can opt for the Premium Performance tier, blob storage, and hot access tier if you want to speed up reading and writing.

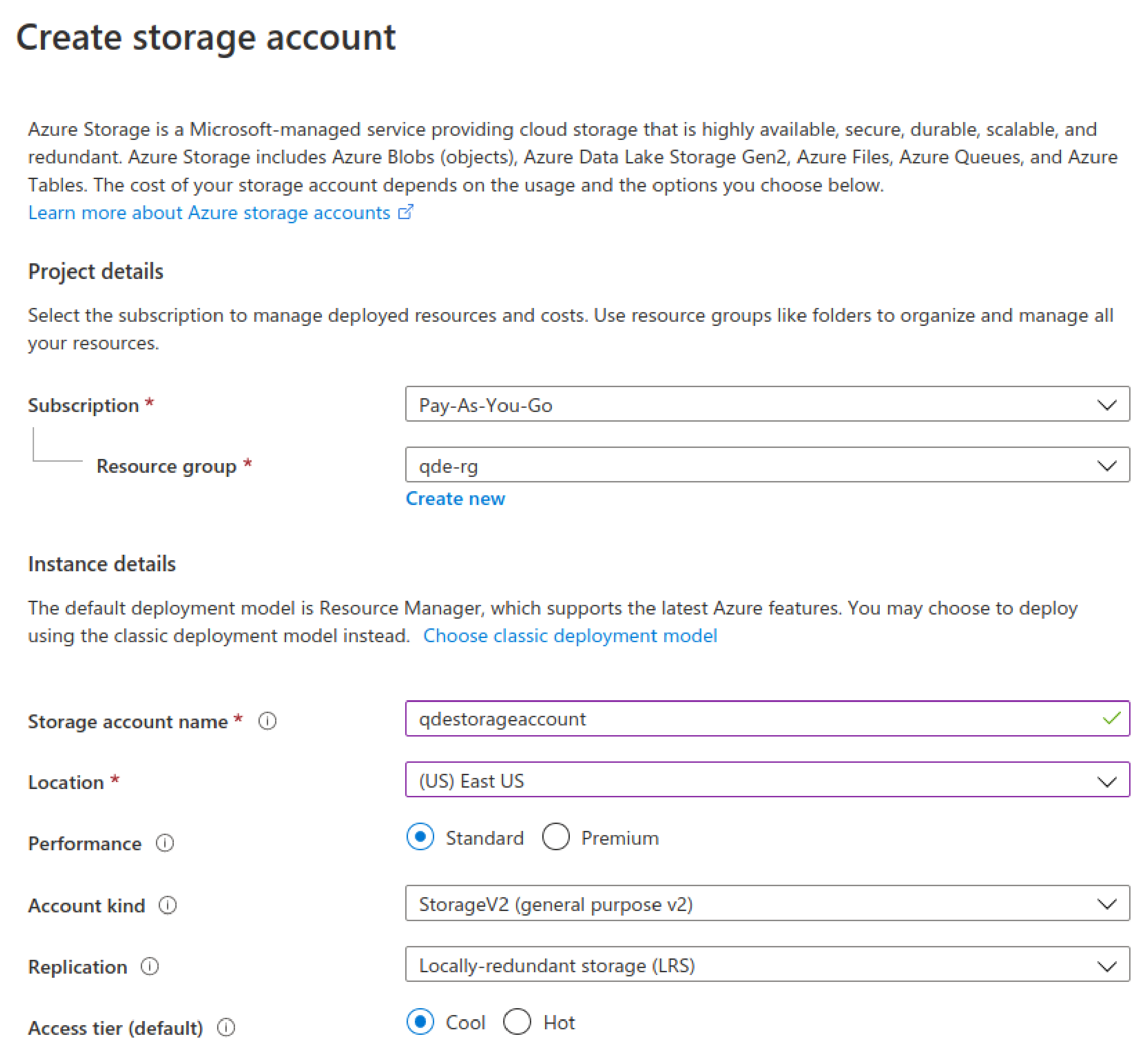
### 2.2. Durability & Security

Choosing another replication option can be useful to improve on the durability of your storage in cases of disasters. When opting for replication to other regions, be aware that reading from the asynchronous replicas is disabled by default. Enabling soft delete is another option to improve the durability in the case of unintentionally deleted files. We can go without it for our case.

### 2.3. Hierarchical Namespaces

We will require secure transfer and disable public access to blobs for security reasons. Make sure to check the option for hierarchical namespaces, which will turn this storage account into the Data Lake storage.

Apart from some additional options for Access Management, the hierarchical namespace is the main advantage of the Data Lake Storage over casual Blob Storage.



## Create storage account

Azure Storage is a Microsoft-managed service providing cloud storage that is highly available, secure, durable, scalable, and redundant. Azure Storage includes Azure Blobs (objects), Azure Data Lake Storage Gen2, Azure Files, Azure Queues, and Azure Tables. The cost of your storage account depends on the usage and the options you choose below.
Learn more about Azure storage accounts

**Project details**

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

| | |
|---|---|
| Subscription * | Pay-As-You-Go |
| Resource group * | qde-rg |
| | Create new |

**Instance details**

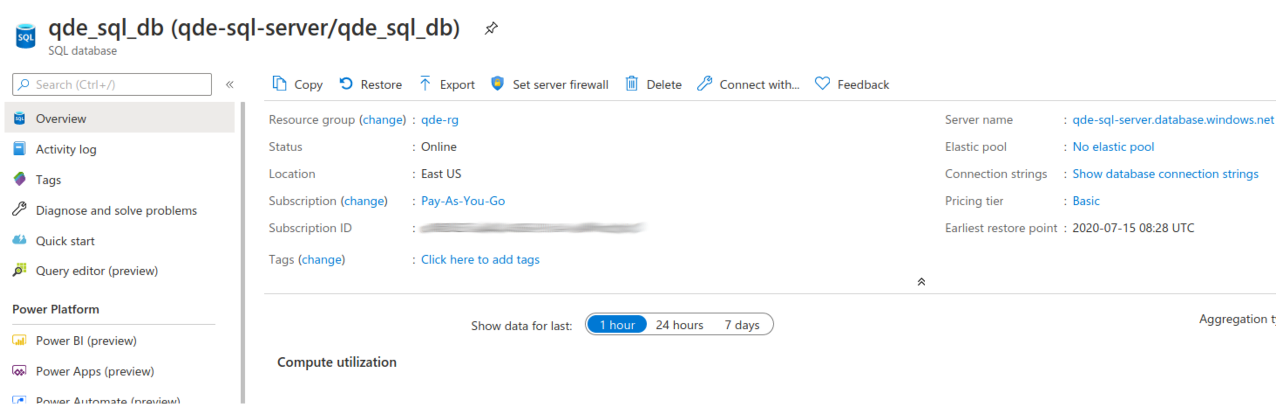
The default deployment model is Resource Manager, which supports the latest Azure features. You may choose to deploy using the classic deployment model instead. Choose classic deployment model

| | |
|---|---|
| Storage account name * | qdestorageaccount |
| Location * | (US) East US |
| Performance | [✓ Standard] Premium |
| Account kind | StorageV2 (general purpose v2) |
| Replication | Locally-redundant storage (LRS) |
| Access tier (default) | [✓ Cool] Hot |

Azure Data Lake Storage - Create Storage Account

## 3. SQL Server and Database

Our tutorial will start with data stored in a database on an SQL Server. Go to your resource group and add the SQL Database to it. You will need the password, so remember it or write it down. In the configuration, create a new SQL Server for the database to run on. A basic database with 2GB storage will be enough for our use case.



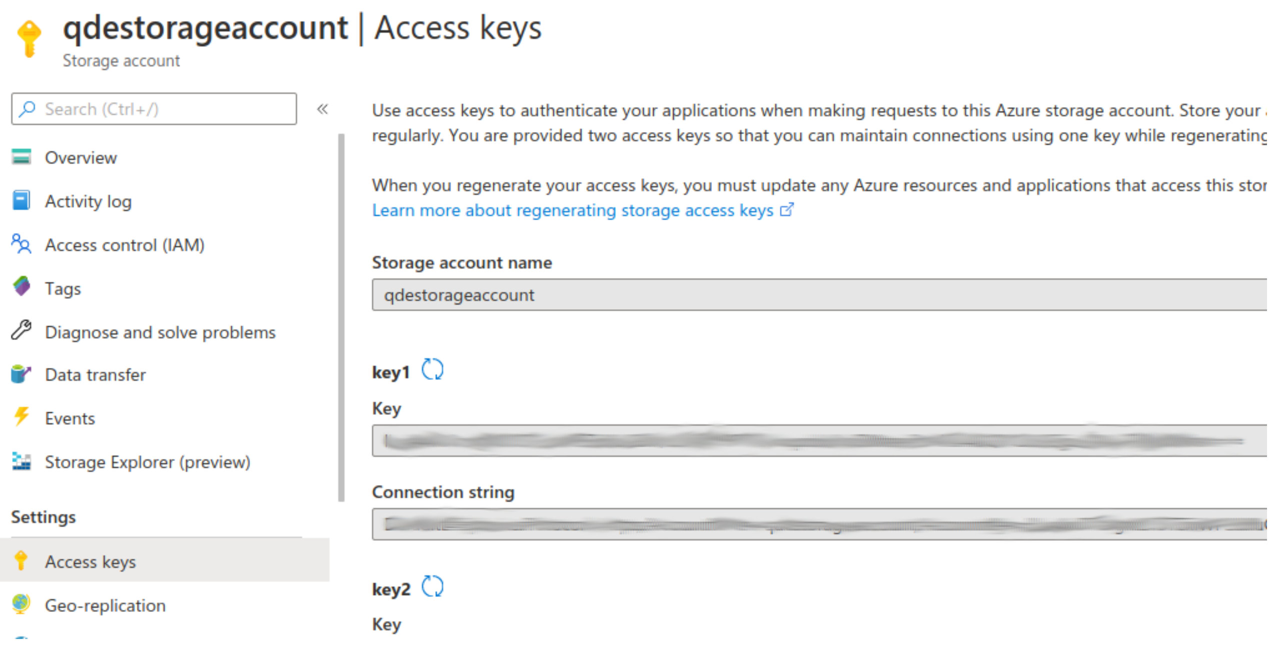
qde_sql_db (qde-sql-server/qde_sql_db)
SQL database

| | | | |
|---|---|---|---|
| Resource group (change) | qde-rg | Server name | qde-sql-server.database.windows.net |
| Status | Online | Elastic pool | No elastic pool |
| Location | East US | Connection strings | Show database connection strings |
| Subscription (change) | Pay-As-You-Go | Pricing tier | Basic |
| Subscription ID | [redacted] | Earliest restore point | 2020-07-15 08:28 UTC |
| Tags (change) | Click here to add tags | | |

Data Engineering on Azure - SQL Server and Database

## 4. Key Vault

There are a few options to control access to blob storage and databases. For storage, you can use Access Keys, Shared Access Signatures, and the Azure Active Directory. Shared Access Signatures are a good option if you want to restrict access to individual objects or to a specific time period. Access with Azure AD works in combination with Role-Based Access Control (RBAC) and is evaluated when you are personally trying to access the storage. Also, it can be a good tool to grant access to applications over a service principal. For our case, we will grant access with the access keys. Those are basically master keys to the storage account, that will enable you to perform nearly all operations on the storage account, except special things like mounting data to databricks. Be aware that RBAC trumps the rights granted by Access Keys.



Use access keys to authenticate your applications when making requests to this Azure storage account. Store your
regularly. You are provided two access keys so that you can maintain connections using one key while regenerating

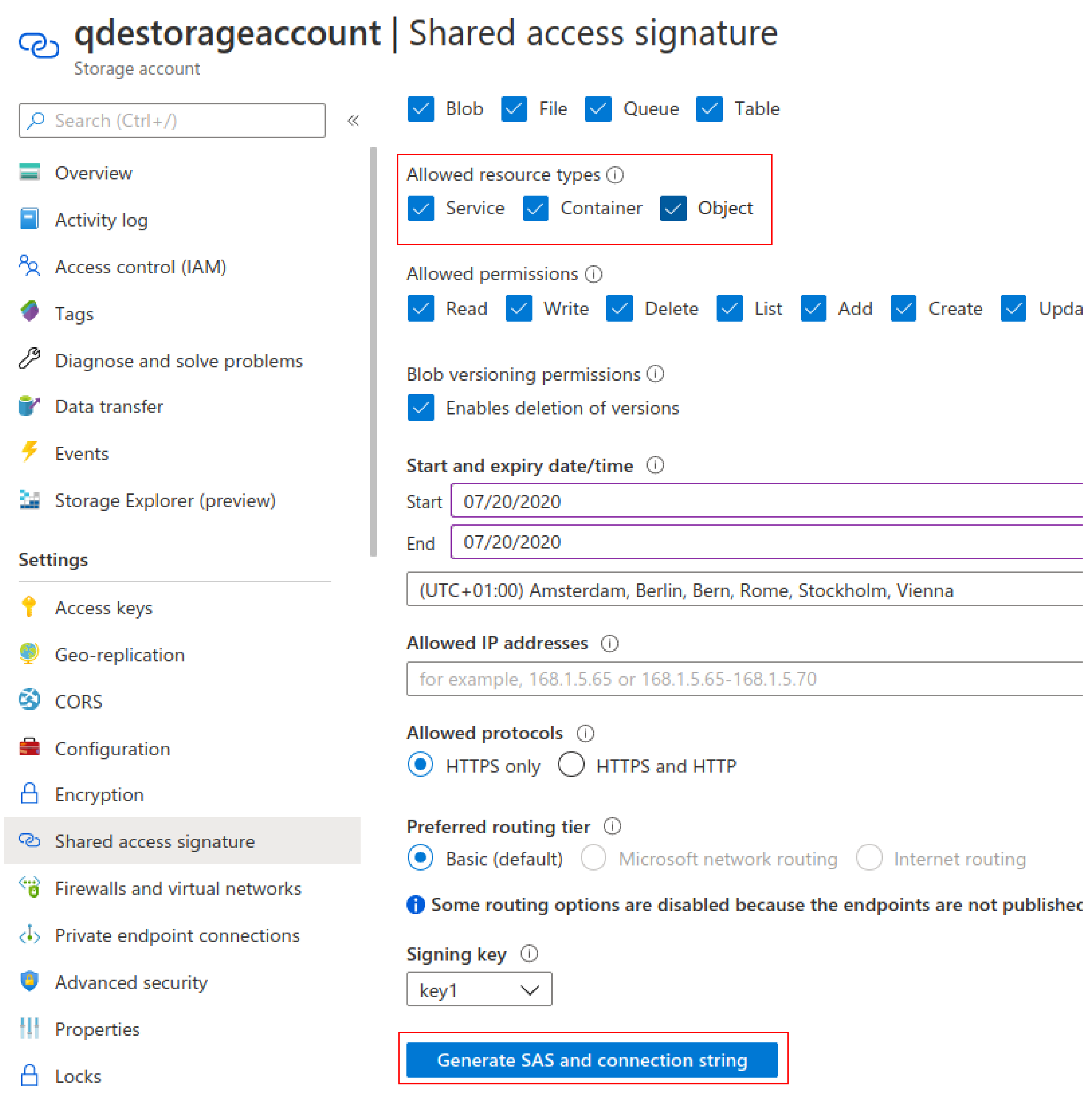
When you regenerate your access keys, you must update any Azure resources and applications that access this stor
Learn more about regenerating storage access keys

Storage account name
qdestorageaccount

key1
Key
[redacted]
Connection string
[redacted]

key2
Key

Data Engineering on Azure - Setting up the Resources - Access Keys
As a more secure alternative, you can also generate a Shared Access Signature on the whole storage account or for a subset of components. That signature can include different permissions like 'Read' or 'Write' and the validity can be restricted to a timeframe.


qdestorageaccount | Shared access signature
Storage account

Search (Ctrl+/)

- 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

[✓ Blob] [✓ File] [✓ Queue] [✓ Table]

Allowed resource types ⓘ
[✓ Service] [✓ Container] [✓ Object]

Allowed permissions ⓘ
[✓ Read] [✓ Write] [✓ Delete] [✓ List] [✓ Add] [✓ Create] [✓ Upda

Blob versioning permissions ⓘ
[✓ Enables deletion of versions]

Start and expiry date/time ⓘ
Start 07/20/2020
End 07/20/2020
(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 publishec

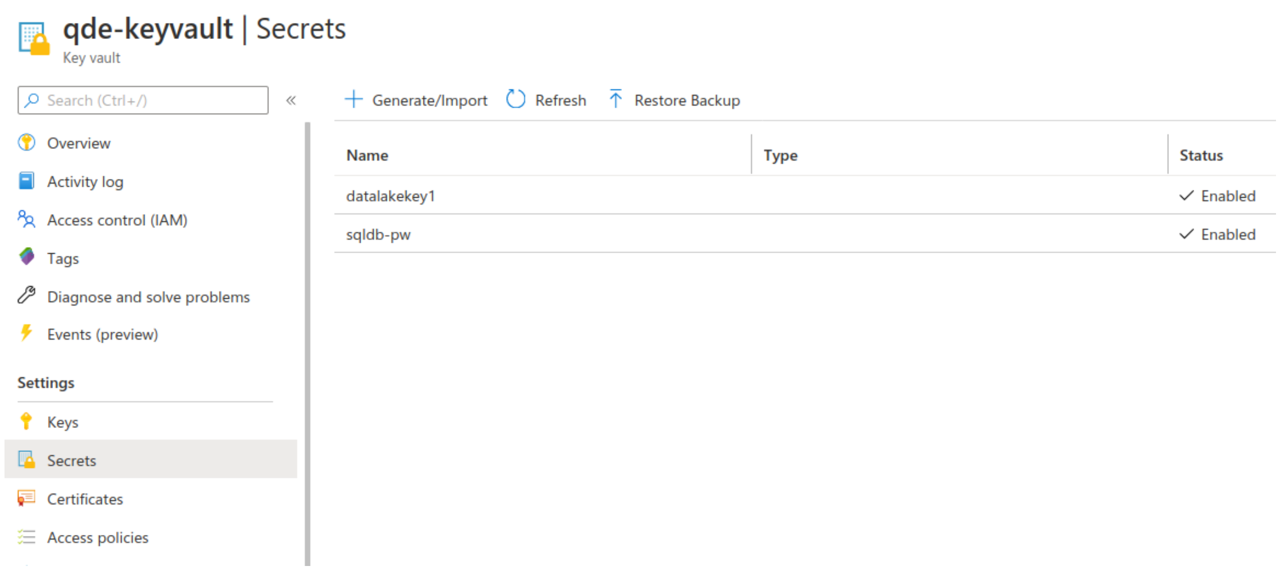
Signing key ⓘ
key1

Generate SAS and connection string

Data Engineering on Microsoft Azure - Shared Access Signature
Access to the database can be granted for Azure AD identities and database users. We will use the database user credentials, more specifically the password. Since we do not want to enter our raw access data to connect to the resources, we will make use of a key vault. Go to your resource group and add a key vault to it. Once the key vault is up and running, open it and switch to the secrets tab. Generate/Import the two Secrets:

1. Containing the Access Key of your Storage Account

2. Containing your database user password



| Name | Type | Status |
|---|---|---|
| datalakekey1 | | ✓ Enabled |
| sqldb-pw | | ✓ Enabled |

Data Engineering on Microsoft Azure - Key Vault - Secrets

## Wrapping It Up

Now that we have setup the necessary resources, we will be able to build a pipeline that will transfer data from our SQL DB to the Data Lake storage. This concludes part 1 of our tutorial. See you on part 2 Basic ETL Processing with Azure Data Factory.

### Sources

- Azure Docs: Resource Groups
- Azure Docs: Introduction to ADLS G2
- Azure Docs: Introduction to Blob Storage
- Azure Docs: Authorizing Access to Blob Storage
- Azure Docs: Key Vault Concepts
- Azure Docs: Create Azure SQL Database

For the next part of the tutorial click here.

# Basic ETL Processing with Azure Data Factory - 2/4

qimia.io/en/blog/Basic-ETL-Processing-with-Azure-Data-Factory

Welcome back to the second article of our Azure ETL series. In the last article, we saw how to create resources in Azure.

In this episode, we will show you how to set up an Azure Data Factory (ADF), how to handle costs within ADF, and how to process a sample dataset. As described in the introduction, we use the Northwind data, load it to an MS SQL database and dump it from there to Azure Data Lake storage in a daily running procedure using ADF.

## 1. Azure Data Factory

Azure Data Factory is a cloud-based ETL and data integration service to create workflows for moving and transforming data. With Data Factory you can create scheduled workflows (pipelines) in a code-free manner. Each graphically programmed block in the pipeline represents a piece of JSON-code that can also be coded manually or ingested from a GIT repository instead of dragging and dropping. The basic workflow in Data Factory is structured in pipelines. The data in- and outputs (here sources and sinks) are defined as so called datasets. Each Data Factory usually consists of one or more pipelines that contain and execute activities.

The concepts of datasets, pipelines, and activities are explained later in this article. Any changes to a Data Factory have to be published in order to be saved for the next session unless you have enabled version control! Also, keep in mind that nearly everything that you do in ADF has a price tag:

Not only the execution of a workflow but also building a pipeline, monitoring runs, or just storing the code. However, most of it is quite cheap, such that mainly processing time matters:

- Runtime costs are \$0.25/DIU-hour[1] for data movement and \$0.005/hour per activity
- Runtime of a data flow costs \$0.193-\$0.343 per vCore-hour depending on the types of cores
- Each piece of runtime is calculated per single activity and rounded up to a minute! => It is more cost-effective to have few complex activities with long lasting execution than many simple ones.

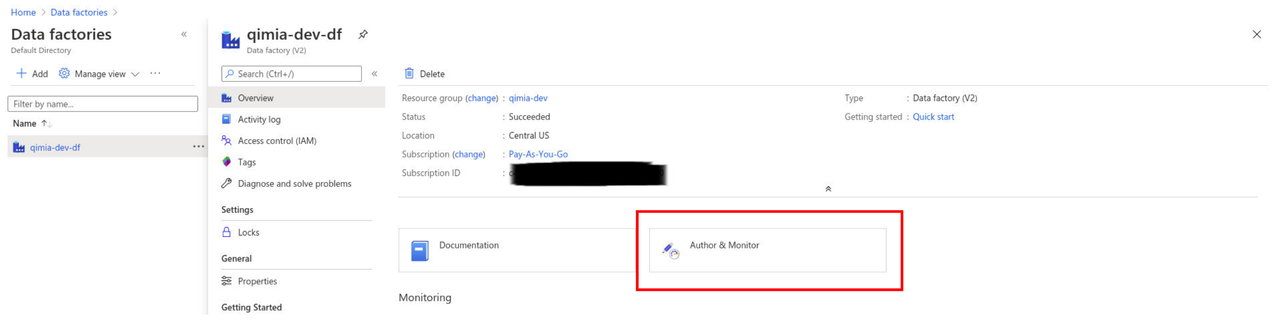
[1] DIU-hour = Data Integration Unit hour, essentially compute nodes per hour

### 1.1. Creating an Azure Data Factory

In order to create a Data Factory resource, go to the Azure Portal. Search for „data factories“ and create/add one. Choose a globally unique name. Choose the **Version 2**.

Place the Data Factory in the resource group you created and choose the same region (=location) as for the storage and resource group that you created in the first episode. You don't have to enable GIT for now. However, it is a best practice to include some version control. You can also use the Azure DevOps repo for it.

Every publishing you do on ADF will also be pushed to that repo. However, saving your code in the repo does not automatically publish it. We provide the complete code for our ADF as JSONs on our Github. You can add the JSONs to your repo and deploy a Data Factory including the resource from the repo. Now, everything is in place for getting started with the ETL preparation. You can go to your Data Factory and enter it by clicking "Author & Monitor".



Resource group (change) : qimia-dev
Status : Succeeded
Location : Central US
Subscription (change) : Pay-As-You-Go
Subscription ID : [redacted]
Type : Data factory (V2)
Getting started : Quick start

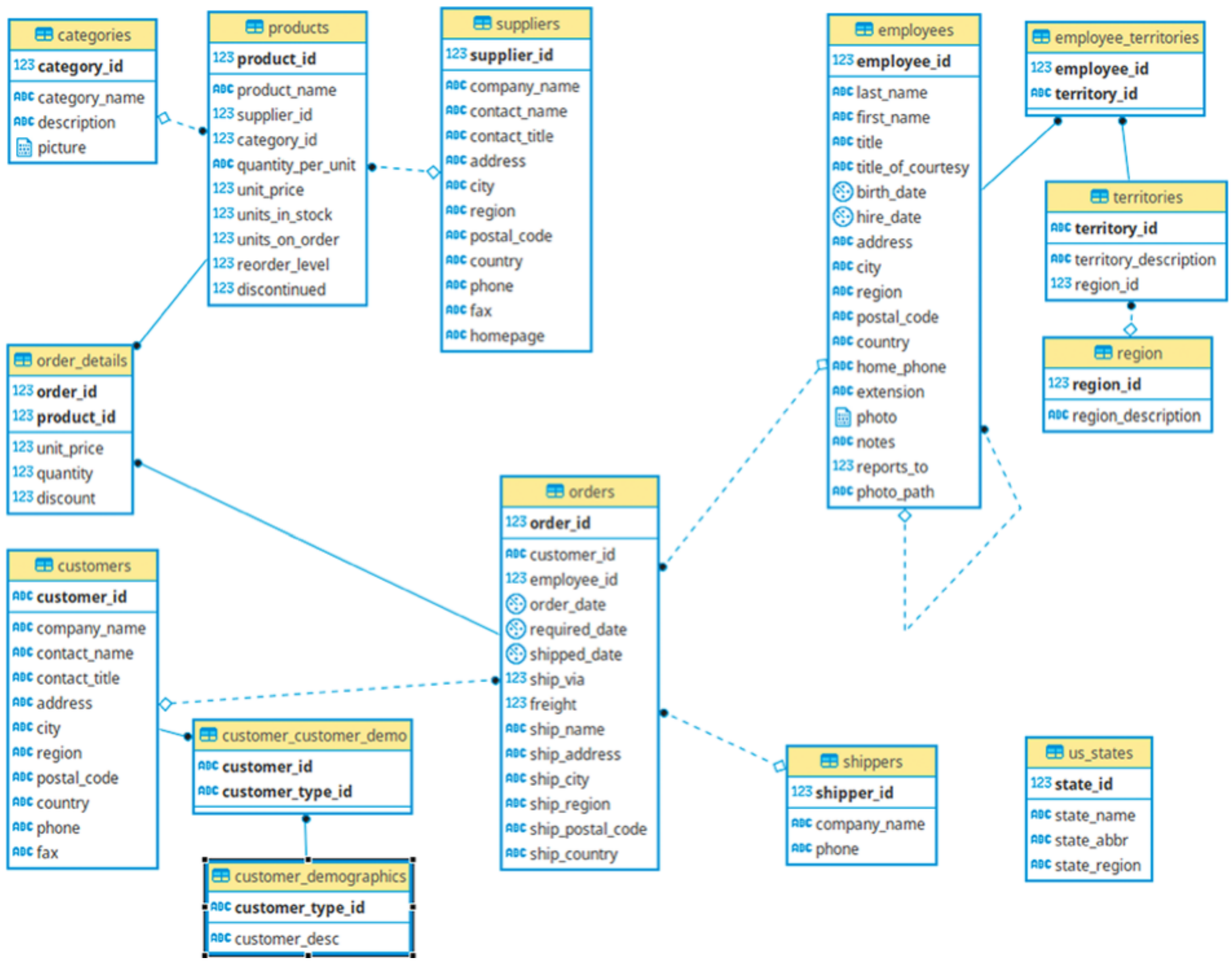
Creating an Azure Data Factory

## 2. Basic ETL Example

In this section, we will create an easy pipeline that already contains a lot of tools in ADF. Furthermore, we explain the basic ideas of it, such as linked services, datasets, activities, and pipelines.

### 2.0. Preparation

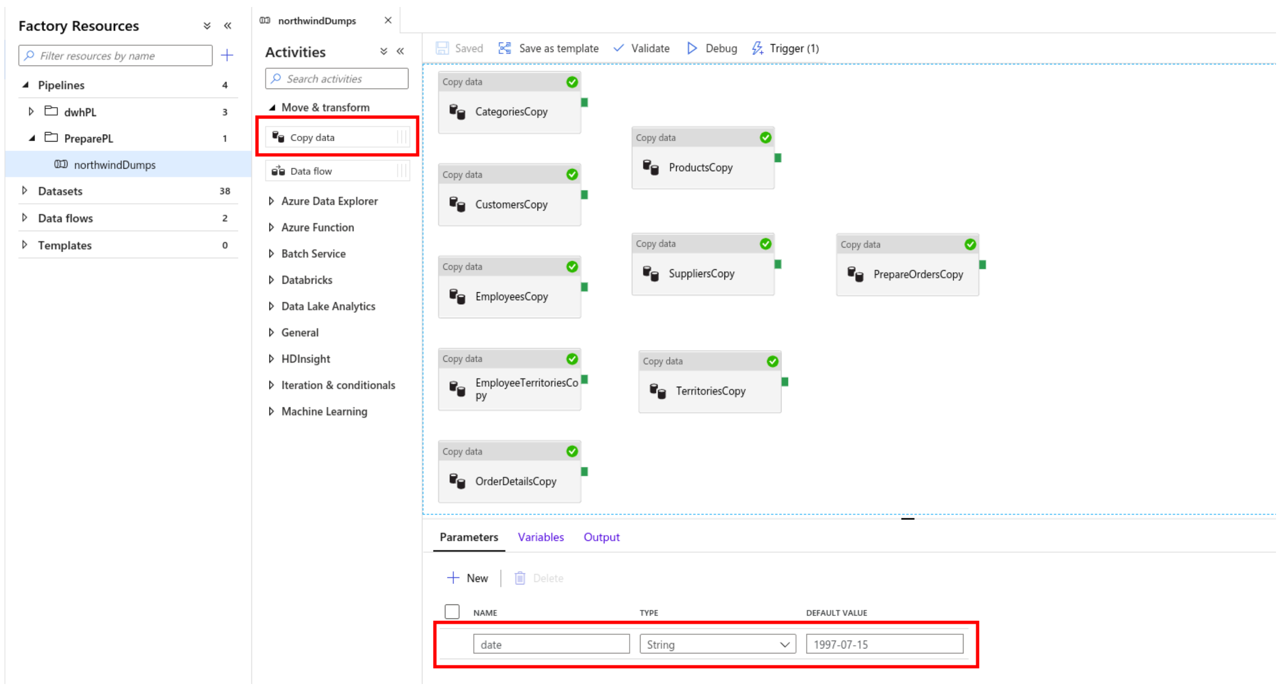
As mentioned before, we will inject the data into the Azure SQL-Server. The data itself can be found here as an SQL file containing CREATE and INSERT statements. Creating and entering an Azure SQL DB is quite similar to creating the Synapse DWH, which will be done in part 3. If you are not sure about it, you can check it out in our next article Creating the Data Warehouse Schema. The Northwind data is organized in an OLTP-like data schema:



Northwind Data Schema
Northwind Data Schema Image Source

## 2.1. The Pipeline

In Data Factory a pipeline is a group of activities, each of them performing pieces of the workflow such as copy, transform or verify data. These activities are brought together in a DAG-like graphical programming interface. In order to control the workflow, a pipeline has two other basic features: Triggers and Parameters/Variables. Furthermore, the pipeline can change the workflow, if a failure occurs. In our scenario, we just create one pipeline.



| NAME | TYPE | DEFAULT VALUE |
|---|---|---|
| date | String | 1997-07-15 |

Basic ETL Example - The Pipeline

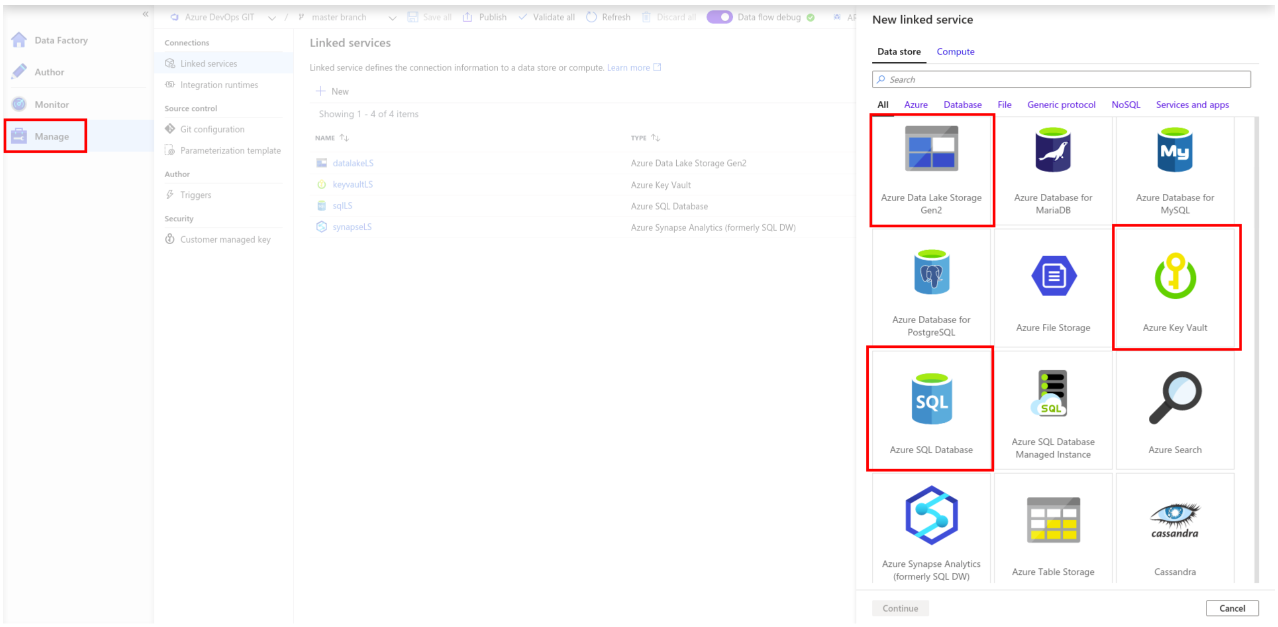
The copy activities in the preparation pipeline do not have any dependencies. They are all executed in parallel. Here we just put them next to each other for a better overview.

It reads partitions from the orders table together with all other needed tables and dumps them to ADLS. This pipeline just contains nine copy-activities. One for each table that we use for the DWH model: Categories, Customers, Employees, Employee-Territories, Order Details, Products, Suppliers, Territories, and Orders. Furthermore, it includes one Parameter "date". This parameter can later be used for scheduling. Each copy activity has a Source and a Sink. For both, you need to create a dataset, which is tied to linked services.

### 2.1.1. Linked Services and Datasets

The Linked Service (LS) is essentially a connection string to the data that has to be processed together with the runtime that should be used for it. A dataset is basically a view or a representation of data, which is used by activity within a pipeline. In order to create an LS, you need to switch to the "Manage"-tab in your ADF. Here we need three linked services:

One for the Key Vault, one for the storage, and one for the SQL database. Go to the Manage-tab and create the linked services. Choose the according tiles. In this example, we provide the access key to the storage via Key Vault. This is the more secure way as is suggested by Azure.



Linked services

Linked service defines the connection information to a data store or compute. Learn more

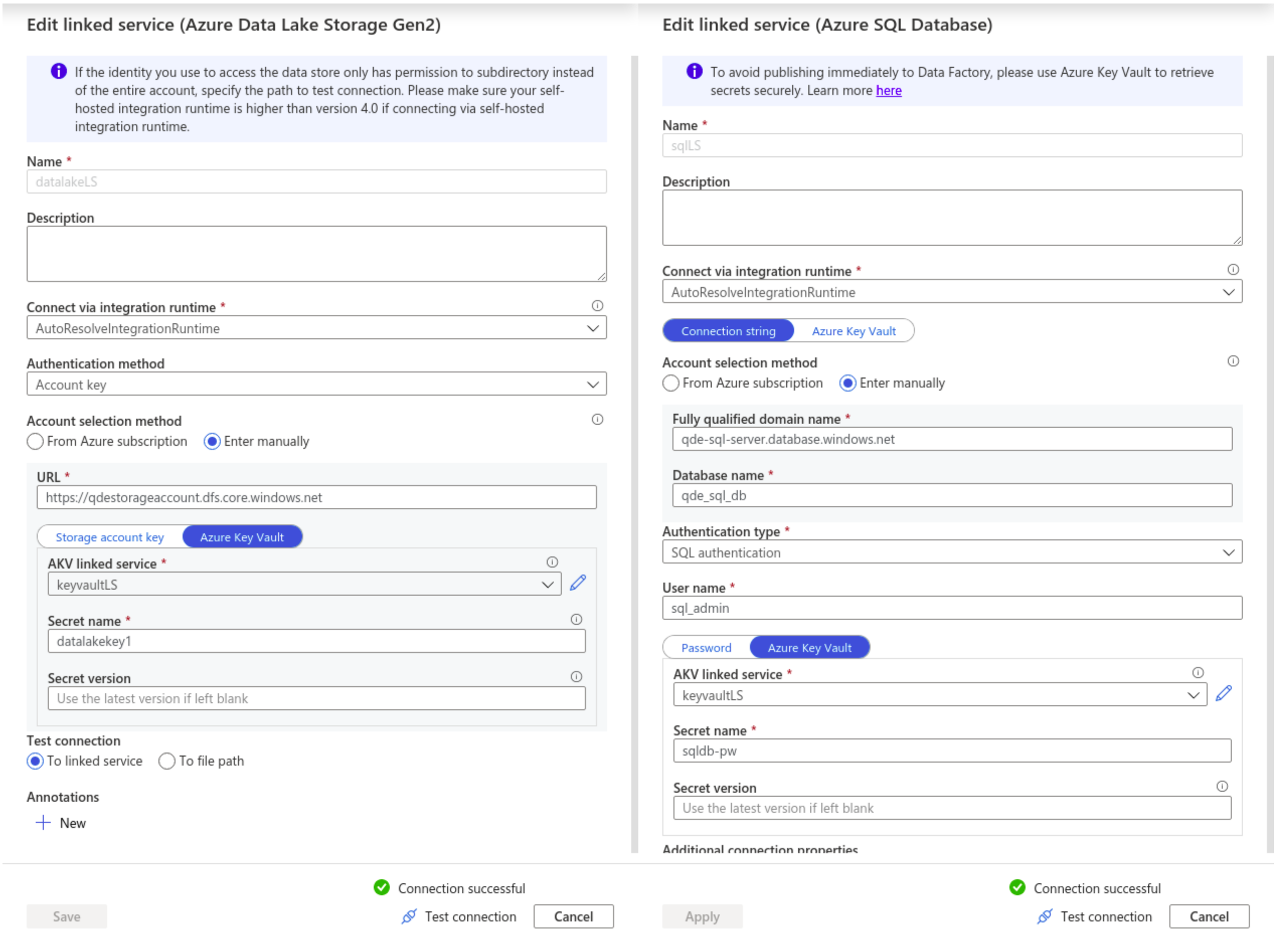
+ New

Showing 1 - 4 of 4 items

| NAME | TYPE |
|---|---|
| datalakeLS | Azure Data Lake Storage Gen2 |
| keyvaultLS | Azure Key Vault |
| sqlLS | Azure SQL Database |
| synapseLS | Azure Synapse Analytics (formerly SQL DW) |

Azure Data Factory - The Pipeline - Linked Services and Datasets I
Create the Key Vault linked service first. You will be asked to grant Data Factory service access to the Key Vault. Copy the object ID and click that link. You will be redirected to a page in the Key Vault, where you can add access policies.

Add an access policy by giving it the needed secret permissions and your object ID. After adding the policy, make sure to hit "save". For the other Services, you can use the key vault secrets for accessing the data. You can always test the connection. Make sure that all services are up and running when you test the connection to them.



Edit linked service (Azure Data Lake Storage Gen2)

If the identity you use to access the data store only has permission to subdirectory instead of the entire account, specify the path to test connection. Please make sure your self-hosted integration runtime is higher than version 4.0 if connecting via self-hosted integration runtime.

Name *: datalakeLS

Description:

Connect via integration runtime *: AutoResolveIntegrationRuntime

Authentication method: Account key

Account selection method: From Azure subscription [✓ Enter manually]

URL *: https://qdestorageaccount.dfs.core.windows.net

Storage account key [✓ Azure Key Vault]

AKV linked service *: keyvaultLS

Secret name *: datalakekey1

Secret version: Use the latest version if left blank

Test connection: [✓ To linked service] To file path

Annotations

+ New

Connection successful

Save | Test connection | Cancel

Edit linked service (Azure SQL Database)

To avoid publishing immediately to Data Factory, please use Azure Key Vault to retrieve secrets securely. Learn more here

Name *: sqlLS

Description:

Connect via integration runtime *: AutoResolveIntegrationRuntime

[✓ Connection string] Azure Key Vault

Account selection method: From Azure subscription [✓ Enter manually]

Fully qualified domain name *: qde-sql-server.database.windows.net

Database name *: qde_sql_db

Authentication type *: SQL authentication

User name *: sql_admin

Password [✓ Azure Key Vault]

AKV linked service *: keyvaultLS

Secret name *: sqldb-pw

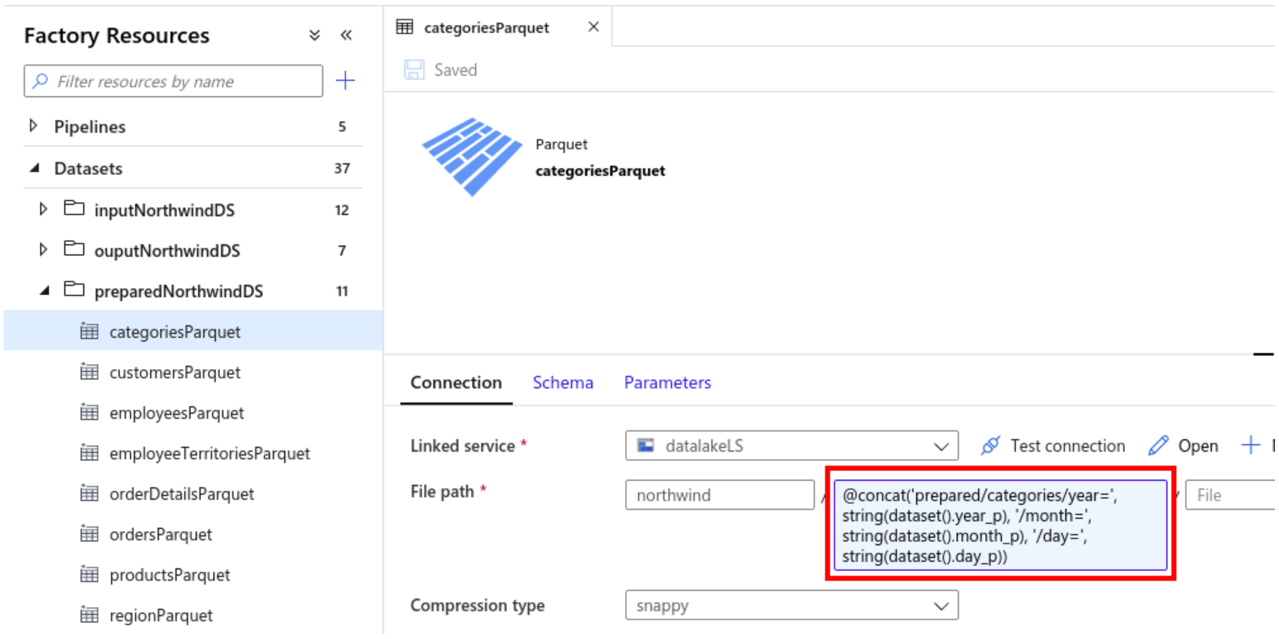
Secret version: Use the latest version if left blank

Additional connection properties

Connection successful

Apply | Test connection | Cancel

Storage Linked Services - SQL Linked Services
You probably need many Data Factoryresources, especially datasets. Therefore, make sure to put them in a nice structure. Work with folders. Give them some meaningful name. We create all datasets that represent the Data in the SQL DB tables. Furthermore, we create all datasets that represent the data in the destination (ADLS2 - parquet file). Here we define the destination Parquet dynamically. The destination folder is parametrized. This will create a kind of data partition.


Factory Resources

Filter resources by name

- Pipelines 5
- Datasets 37
  - inputNorthwindDS 12
  - ouputNorthwindDS 7
  - preparedNorthwindDS 11
    - categoriesParquet
    - customersParquet
    - employeesParquet
    - employeeTerritoriesParquet
    - orderDetailsParquet
    - ordersParquet
    - productsParquet
    - regionParquet

categoriesParquet

Saved

Parquet
categoriesParquet

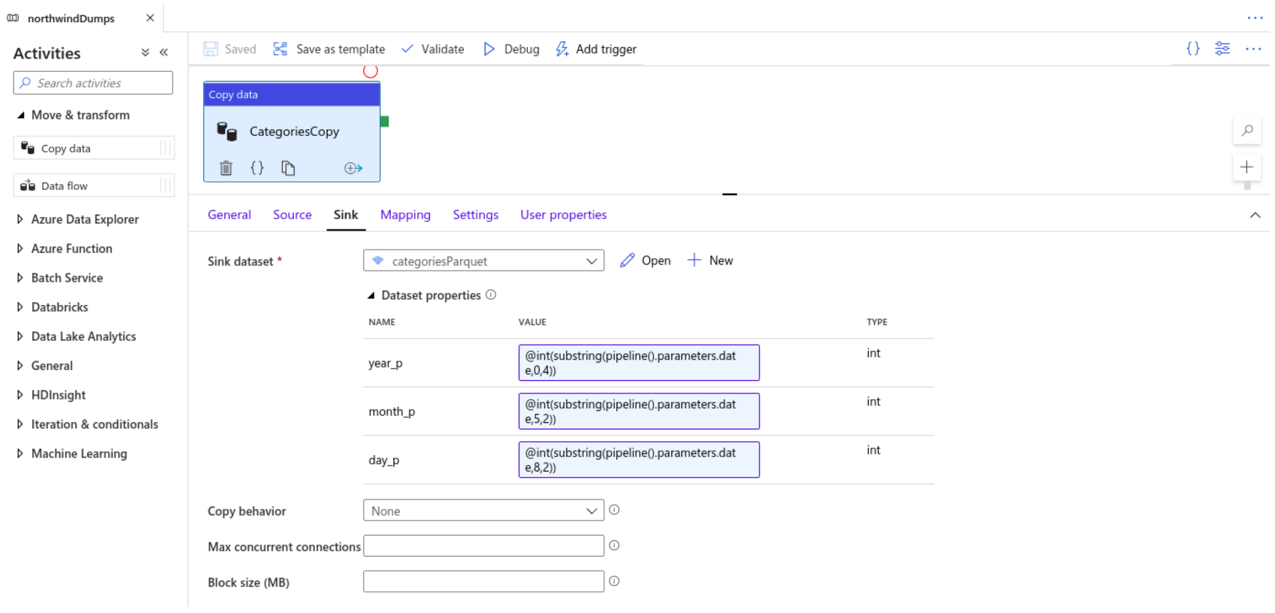
Connection | Schema | Parameters

| | |
|---|---|
| Linked service * | datalakeLS |
| File path * | northwind / @concat('prepared/categories/year=', string(dataset().year_p), '/month=', string(dataset().month_p), '/day=', string(dataset().day_p)) / File |
| Compression type | snappy |

Basic ETL Processing with Azure Data Factory - The Pipeline - Linked Services and Datasets

## 2.1.2. The Copy Activities

The copy activity copies data from a source to a sink dataset. Furthermore, it allows basic mapping (no conditional mapping) such as renaming or typecasting. There are various data sources allowed such as Azure Blob storage, Amazon Redshift, or S3, whereas sinks are more or less limited to the Azure storage systems such as Azure SQL, Azure Synapse, or Azure Blob Storage. In our scenario, we specify a copy activity for each table with the according to Source and Sink datasets. In the Sink dataset, we specify the dataset properties to include the year, month, day parameters.



Sink dataset * categoriesParquet

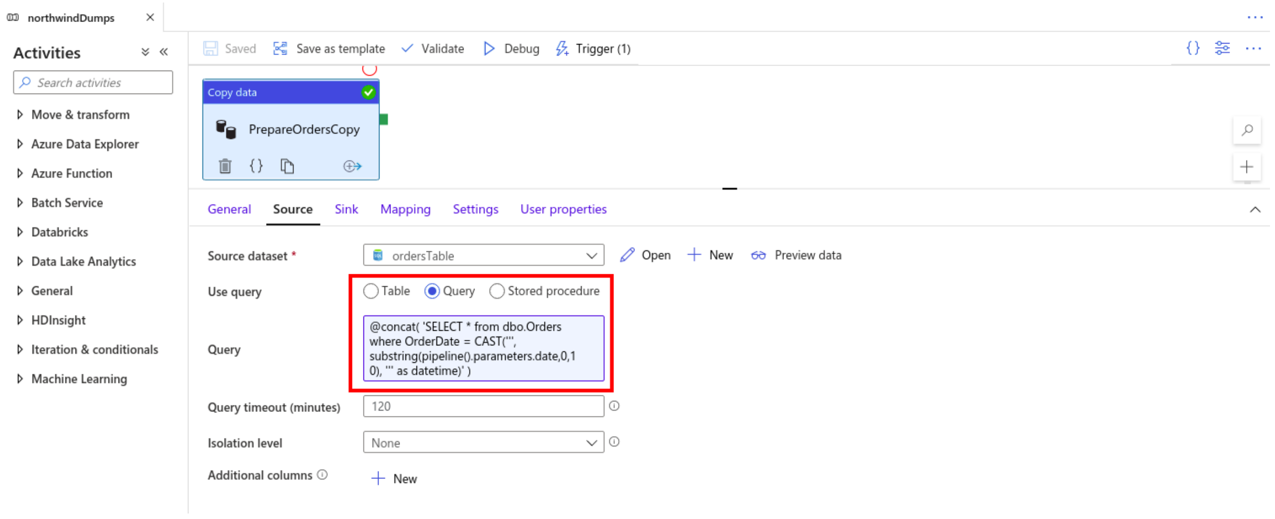
Dataset properties

| NAME | VALUE | TYPE |
|---|---|---|
| year_p | @int(substring(pipeline().parameters.date,0,4)) | int |
| month_p | @int(substring(pipeline().parameters.date,5,2)) | int |
| day_p | @int(substring(pipeline().parameters.date,8,2)) | int |

| | |
|---|---|
| Copy behavior | None |
| Max concurrent connections | |
| Block size (MB) | |

The Copy Activities - I

However, when creating the Orders, we want to read only the data from specific order date. Therefore, we need to enter a query in the Source tab. As we need to parametrize the date, we need to add some dynamic content to the query. Here we need to escape a single quote. In Azures's dynamic expression builder this is done by another single quote.



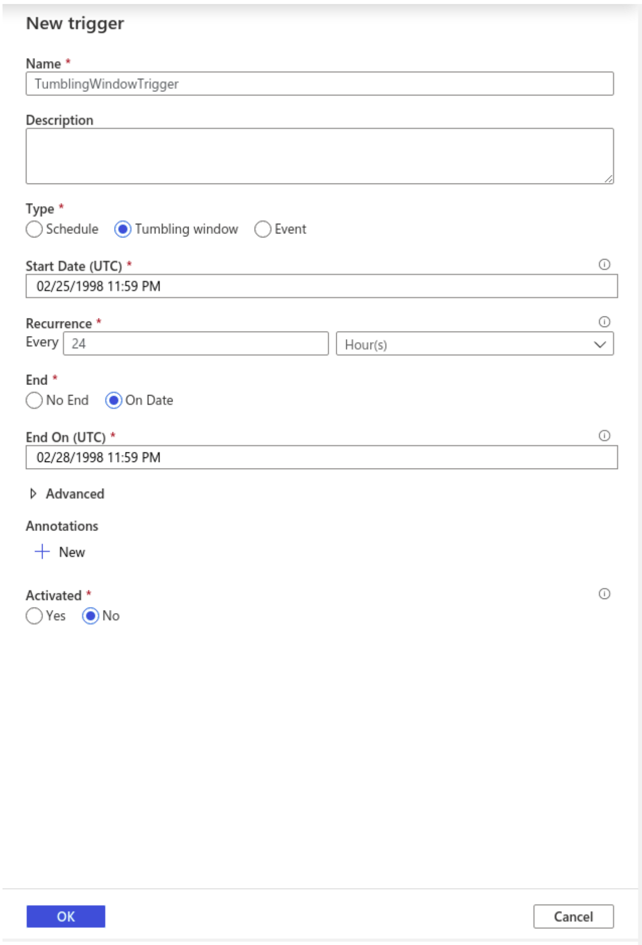
| | |
|---|---|
| Source dataset * | ordersTable |
| Use query | Table [✓ Query] Stored procedure |
| Query | @concat( 'SELECT * from dbo.Orders where OrderDate = CAST(''', substring(pipeline().parameters.date,0,10), ''' as datetime)' ) |
| Query timeout (minutes) | 120 |
| Isolation level | None |
| Additional columns | New |

The Copy Activities II

### 2.1.3. The Trigger

In our scenario, we want to do the dumping on a daily basis. However, as the data originates from 1996-1998, we need a backfilling mechanism. Therefore, we choose a so-called Tumbling Window (TW) Trigger.

You can find a more detailed discussion on the pros and cons of that trigger in our final article Building an Azure Data Factory Pipeline for Professional Data Warehousing. In order to create a trigger, go to your pipeline and click on the Trigger tab. Add a new Tumbling Window trigger and choose the appropriate start and end date. You can either activate it directly, which will cause that it triggers some runs directly after publishing, or you can set it to inactive and activate it later in your "Manage" tab. If you defined the parameter "date" on the pipeline, you will be asked what value that parameter should have. Here you can enter the @trigger().outputs.windowStartTime as a dynamic content.


New trigger

Name *
TumblingWindowTrigger

Description

Type *
Schedule [✓ Tumbling window] Event

Start Date (UTC) *
02/25/1998 11:59 PM

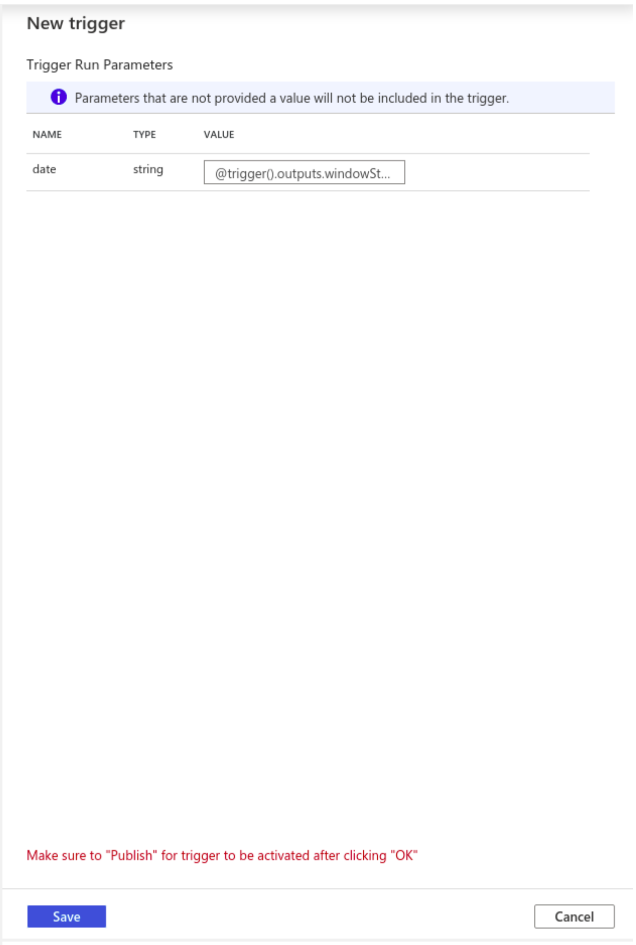
Recurrence *
Every 24 Hour(s)

End *
No End [✓ On Date]

End On (UTC) *
02/28/1998 11:59 PM

Activated *
Yes [✓ No]

Setting up the TW Trigger


New trigger

Trigger Run Parameters

Parameters that are not provided a value will not be included in the trigger.

| NAME | TYPE | VALUE |
|---|---|---|
| date | string | @trigger().outputs.windowSt... |

Make sure to "Publish" for trigger to be activated after clicking "OK"

Specify the pipeline parameter

Basic ETL Processing with Azure Data Factory - The Trigger
If you publish these last changes you can go to the monitor tab and you will recognize that the pipeline will be triggered for the days you specified in the trigger. This completes our episode on basic processing with ADF.

## Wrapping It Up

Summing it up, we have created a Data Factory together with a pipeline that contains several copy activities. We saw how to create linked services and datasets that we need for the processing. We also saw how to dynamically schedule a trigger and parametrize the pipeline in terms of which data should be read and where it is stored. In our next episode, we will learn how to set up a Synapse data warehouse with a schema that is suitable for business analytics. The last episode will present the Data Factory pipeline, which aggregates the data that is going to the data warehouse.

### Sources

- Azure Docs: Data Factory overview page

- Azure Docs: Data Factory Copy Activity
- MS Azure Data Pipeline Pricing
- Azure Docs: Data Factory Pricing examples
- Azure Docs: Concepts of Linked Services
- Azure Docs: Concepts of Datasets

For the next part of the tutorial click here.

# 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

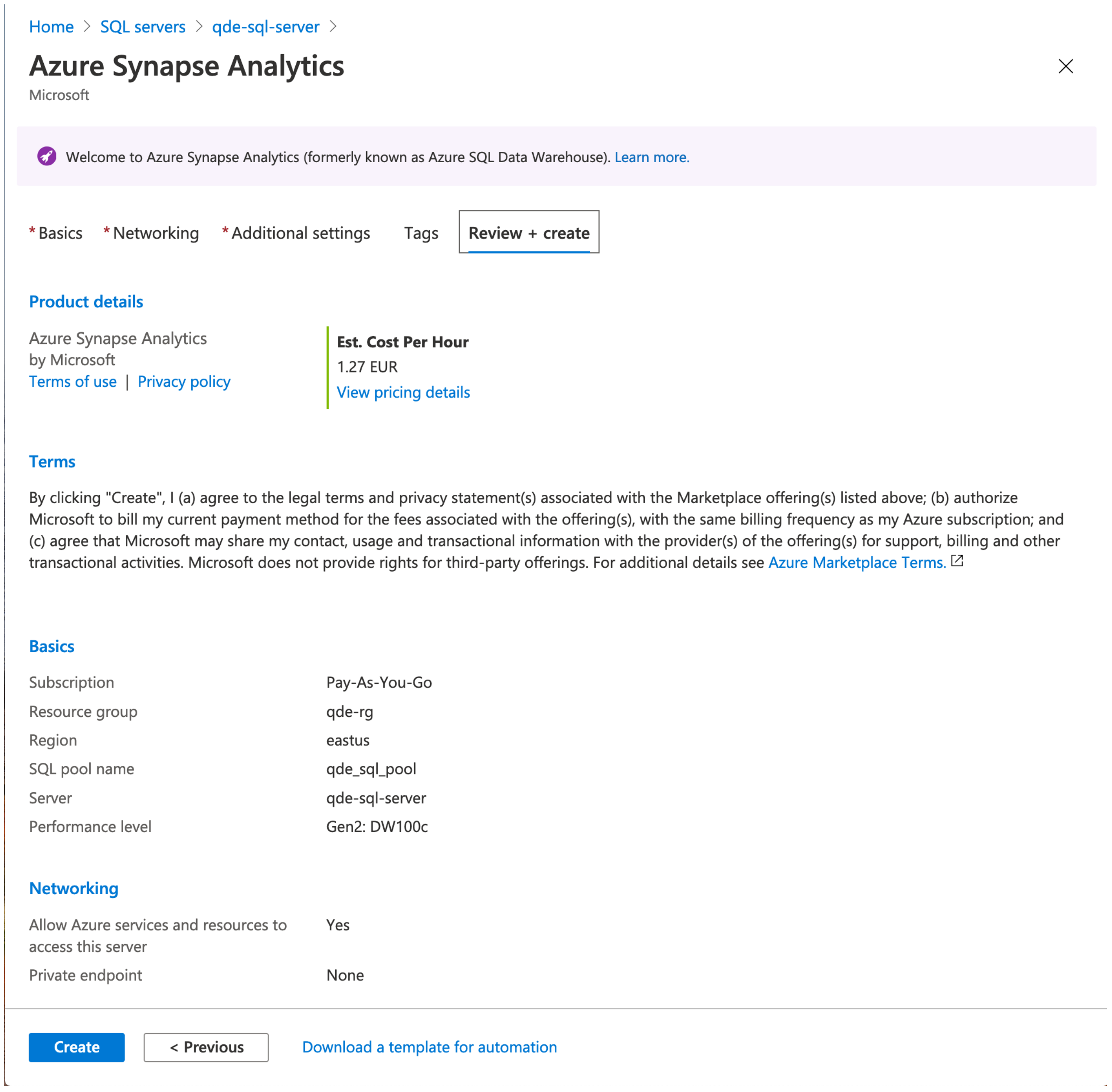
Synapse is the Data Warehousing platform of Azure. It has two versions, the standalone version that is just the SQL pool (called in the Azure Portal "Azure Synapse Analytics (formerly SQL DW)") and the "Synapse Data Analytics (workspaces preview)" integrating the DWH with various other Azure services 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 |

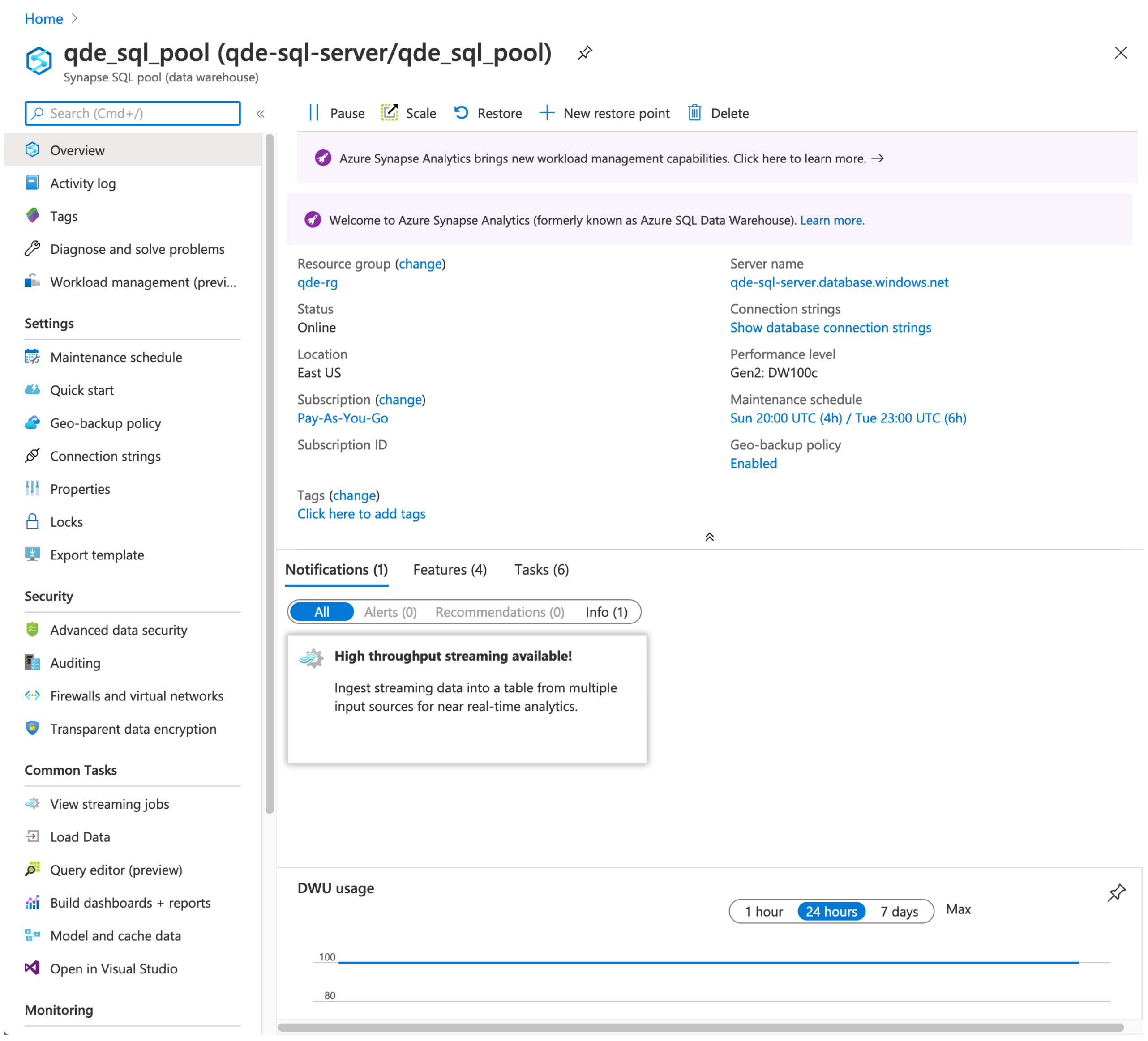
**Networking**

| | |
|---|---|
| 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.



qde_sql_pool (qde-sql-server/qde_sql_pool)

Synapse SQL pool (data warehouse)

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.

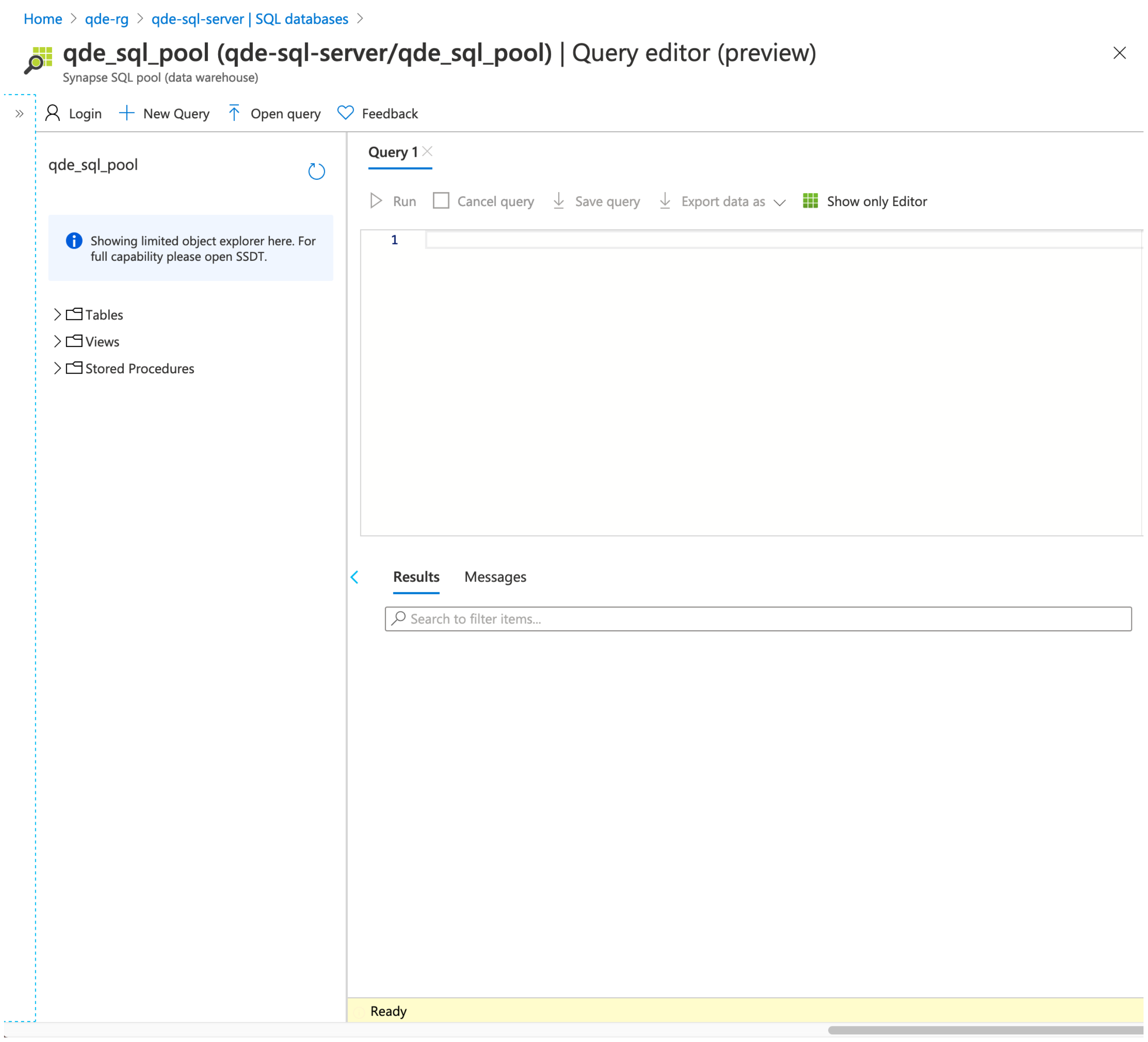
| | |
|---|---|
| Resource group (change)<br>qde-rg | Server name<br>qde-sql-server.database.windows.net |
| Status<br>Online | Connection strings<br>Show database connection strings |
| Location<br>East US | Performance level<br>Gen2: DW100c |
| Subscription (change)<br>Pay-As-You-Go | Maintenance schedule<br>Sun 20:00 UTC (4h) / Tue 23:00 UTC (6h) |
| Subscription ID | Geo-backup policy<br>Enabled |
| Tags (change)<br>Click here to add tags | |

**High throughput streaming available!**

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

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.

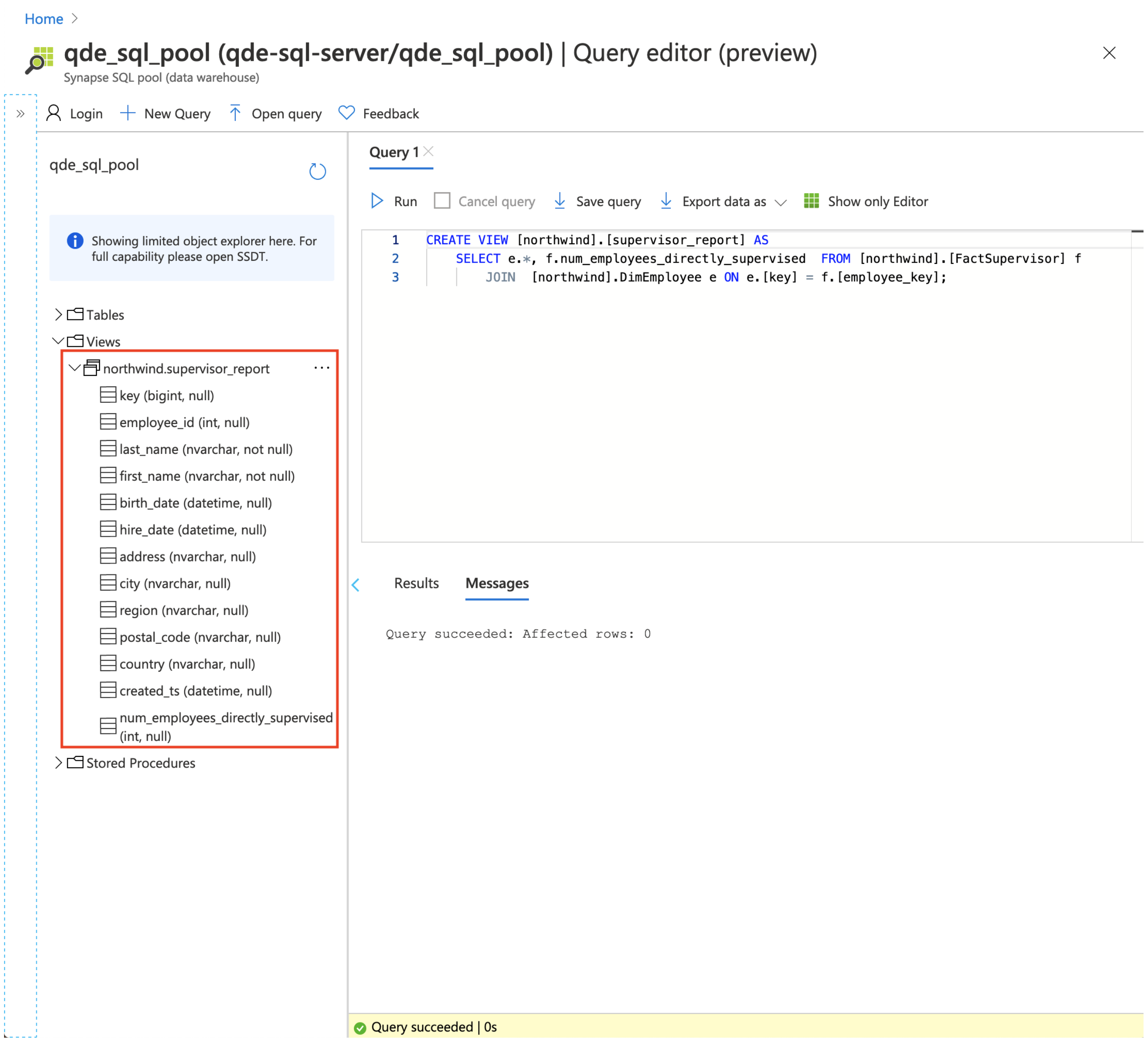
## 3.4. Preparing Business Report Views

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
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)
Stored Procedures
Query 1
Run
Cancel query
Save query
Export data as
Show only Editor
Results
Messages
Query succeeded | 0s

```
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
```

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.

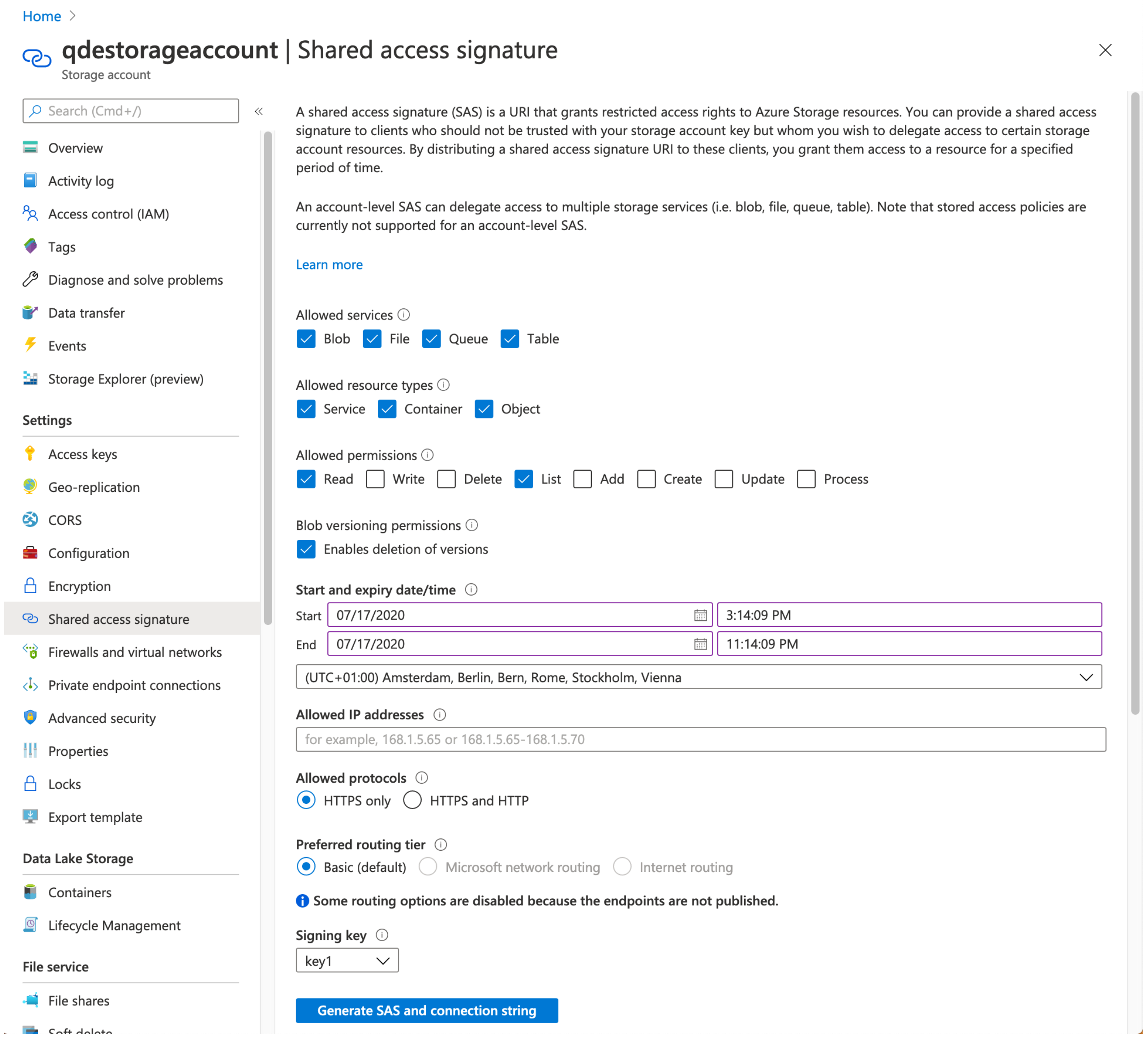
To access a datalake, you will need credentials. We recommend either using a Storage Account Key or a Shared Access Key. Storage Account Key is the master credentials for our storage account. You can grab the value used in the Setup.

```
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:



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.

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

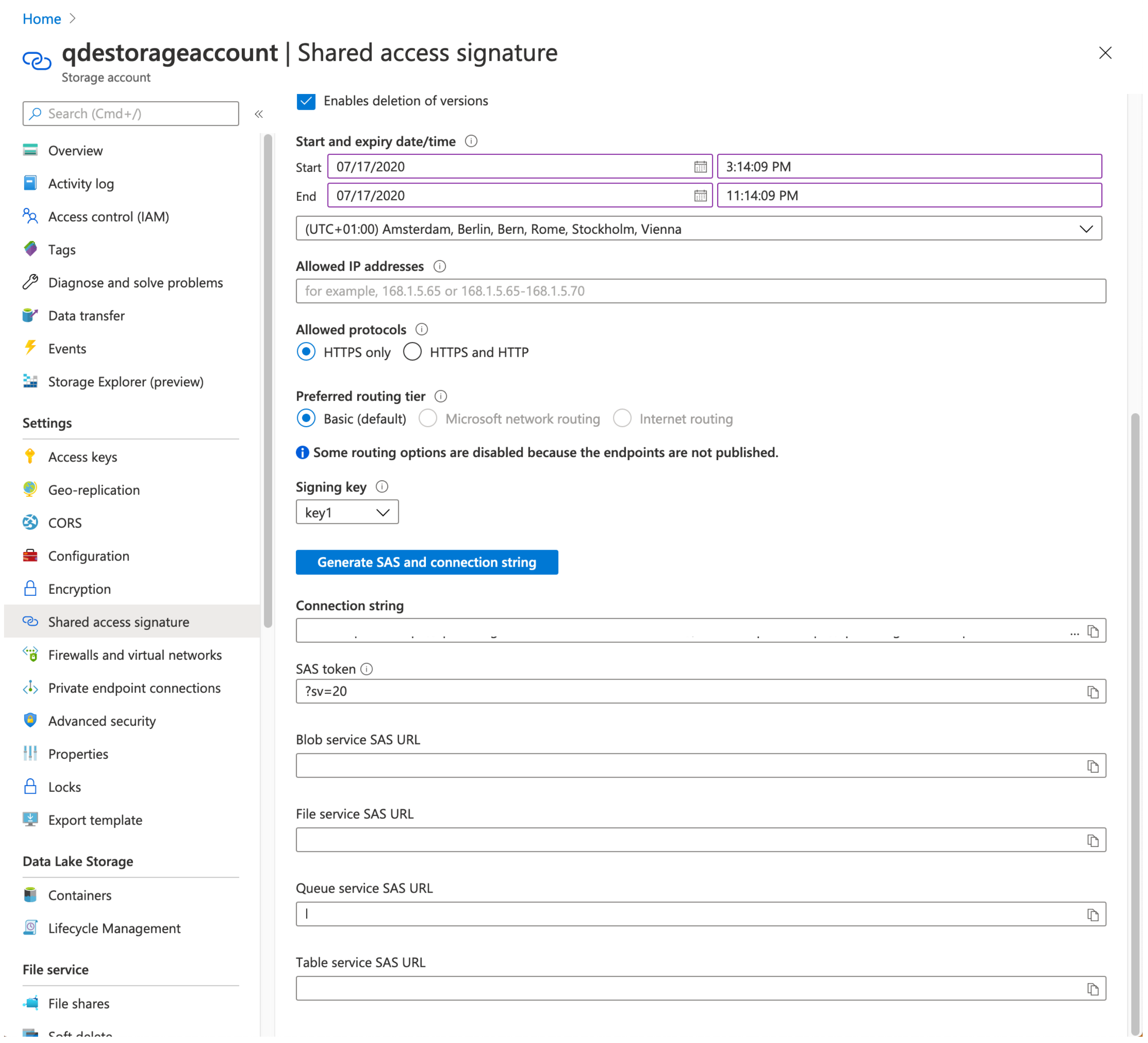
Allowed protocols: [✓ HTTPS only] HTTPS and HTTP

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

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

Signing key: key1

SAS I
Copy from 'SAS token' below:


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

[✓] 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**

**Connection string**

SAS token

?sv=20

Blob service SAS URL

File service SAS URL

Queue service SAS URL

l

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.

# Building an Azure Data Factory Pipeline for Professional Data Warehousing - 4/4

qimia.io/en/blog/Building-an-Azure-Data-Factory-Pipeline-for-Professional-Data-Warehousing

Welcome back to our series about Data Engineering on MS Azure. In this article, we describe the construction of an Azure Data Factory pipeline that prepares data for a data warehouse that is supposed to be used for business analytics. In the previous blog's articles, we showed how to set up the infrastructure with Data Engineering on Azure - The Setup.

How to pre-process some data with data factory and which structure we choose for the data warehouse. This article focuses on technical aspects and best practices of data factory data flow, e.g. reading (partitioned) data, hashing, mapping, joining, filtering, and triggering.

## 1. The Basic Idea

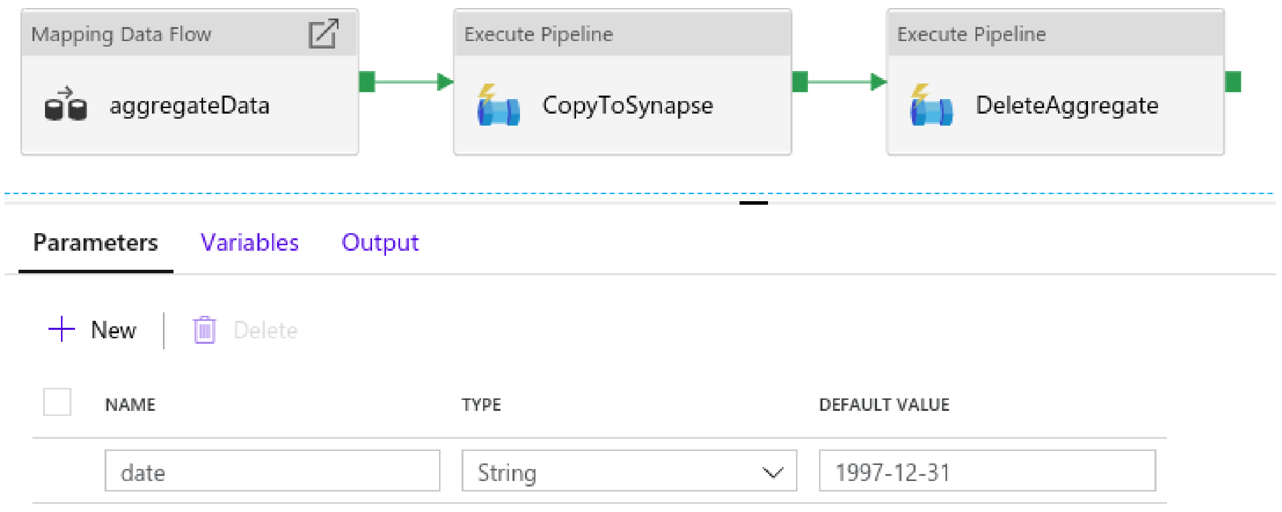
The basic idea of our ETL pipeline is to load the up-to-date Dimension tables on a monthly basis, add a technical key to them, dump them to Azure Data Lake Storage (ADLS), and copy them from ADLS to the DWH.

Accordingly, we also build monthly aggregates in Fact tables, which are connected to Dimension tables by foreign keys. Then we dump these dimension tables as well and copy them to the DWH.

## 2. The Pipeline Overview

For the whole data processing, we introduced one main pipeline (dwhPL) that calls a data flow for aggregating all data and dumping it to ADLS. The pipeline subsequently calls another pipeline that copies all dumped data to the Synapse DWH.

As the last step, we delete all aggregated and dumped data from the ADLS in order to reduce our storage capacity and thus save money. Another way of doing so would be using lifecycle management policies on our storage account. The details of these copy/delete pipelines are not covered here.



| NAME | TYPE | DEFAULT VALUE |
|---|---|---|
| date | String | 1997-12-31 |

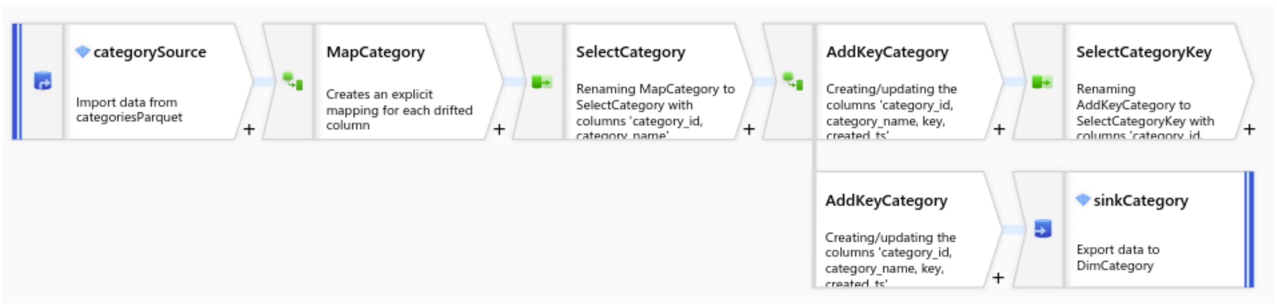
The Pipeline Overview
We will set a pipeline parameter "date" of type string that we will initialize with an arbitrary default value like '1997-12-31'. The idea of that parameter is to hand over a trigger parameter, namely the scheduled execution timestamp. This parameter is used in the aggregation, where we only read data from that particular month. We will come back to the details on how to hand over the trigger parameter later in this article.

## 3. The Aggregation

The aggregation of data takes place in the previously mentioned data flow. You can find the whole data flow JSON in our repo. This section does not mention each and every step of the data flow, but gives an overview and mentions some general techniques. One important feature of our data flow is the parameters, which we use for controlling the data flow execution. Since we have to read the data in monthly batches, we added the parameters month_p and year_p as integers, which will be derived from the date parameter of our pipeline.

### 3.1. The Dimension Tables

There are two general types of tables in our DWH-schema: the dimension tables and the fact tables. In our case, the dimension tables are easier to create as they just contain the incoming information together with a key and timestamp.


categorySource
Import data from categoriesParquet
MapCategory
Creates an explicit mapping for each drifted column
SelectCategory
Renaming MapCategory to SelectCategory with columns 'category_id,
AddKeyCategory
Creating/updating the columns 'category_id, category_name, key,
SelectCategoryKey
Renaming AddKeyCategory to SelectCategoryKey with
AddKeyCategory
Creating/updating the columns 'category_id, category_name, key,
sinkCategory
Export data to DimCategory

The Dimension Tables
Here we use the following elements for creating the Dimension tables:

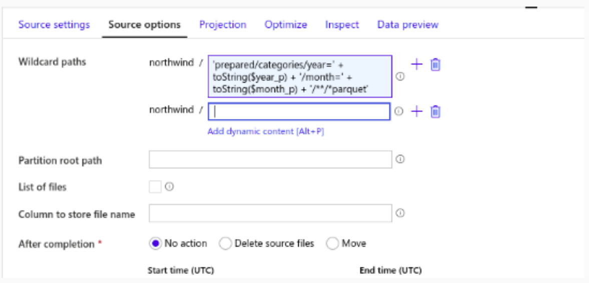
- Read the data from storage
- Map every column (with correct type)
- Select the needed columns
- Add a technical key and timestamp
- Split the stream:

  a) Select ID and key for joining it later to fact tables b) Write it to storage

### 3.1.1. Reading Data

Our data sources are parquet files. We provide a wildcard path to our parquet file since we want to read all months data from the year and month that we are processing in the current run.

It is also possible to add more than one path. After each parquet source, we add a mapping. This is a best practice to ensure that a certain schema is ingested. You lose a bit of flexibility with it, but you gain control and reliability. Data factory provides an automated mapping, which should be used to avoid typing every column manually. In order to do so, go to the data preview tab and press the "Map Drifted" button (this is only possible if you choose to Allow schema drift).



Wildcard paths: northwind / 'prepared/categories/year=' + toString($year_p) + '/month=' + toString($month_p) + '/**/*parquet'

northwind /

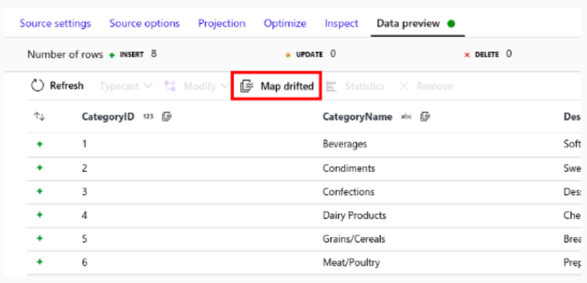
Partition root path:

List of files:

Column to store file name:

After completion *: [✓ No action] Delete source files Move

Source: Wildcard paths



Number of rows INSERT 8 UPDATE 0 DELETE 0

| | CategoryID | CategoryName | Des |
|---|---|---|---|
| + | 1 | Beverages | Soft |
| + | 2 | Condiments | Swe |
| + | 3 | Confections | Des |
| + | 4 | Dairy Products | Che |
| + | 5 | Grains/Cereals | Brea |
| + | 6 | Meat/Poultry | Prep |

Data preview and map drifted

Reading Data

### 3.1.2. Select Columns

It is a best practice to select a few columns as you need as early as possible in the stream. Furthermore, it is a good practice to have another select stage as the last step before sinking your data. This will grant more flexibility for cases where the column names might be changed or the exact same data is reused with slightly different naming at another point in the Data Flow.

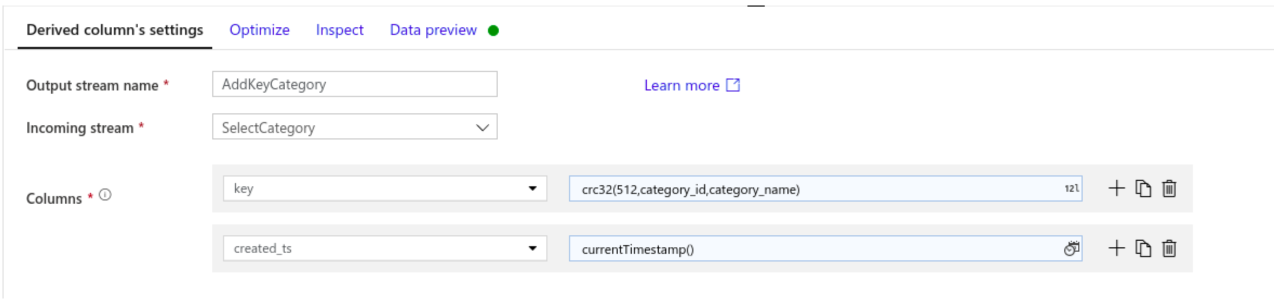
In the case of our dimension processing, we have very short streams. It would not make sense to have a selection at the beginning and one at the end. Therefore we have just one.

### 3.1.3. Adding Columns

It is a best practice to add some kind of timestamp to your data. One can either create an "insert_ts" or a "created_ts". Depending on your choice you would create that column content either in your destination DWH when inserting or you would create it in the data factory when creating the data.

We chose to create it during the ETL process. Furthermore, it is a best practice to have a technical key in the Dimension tables, which can be used for the joining of fact tables later and which guarantees a more balanced distribution, if you distribute your data on that key in the DWH.

In this case, we create the key in the dimension streamand join that key to the fact tables later. We create the key with the hash function crc32 and use the currentTimestamp for the created_ts.



| | | |
|---|---|---|
| Output stream name * | AddKeyCategory | |
| Incoming stream * | SelectCategory | |
| Columns * | key | crc32(512,category_id,category_name) |
| | created_ts | currentTimestamp() |

Adding Columns

### 3.1.4. Splitting and Sinking

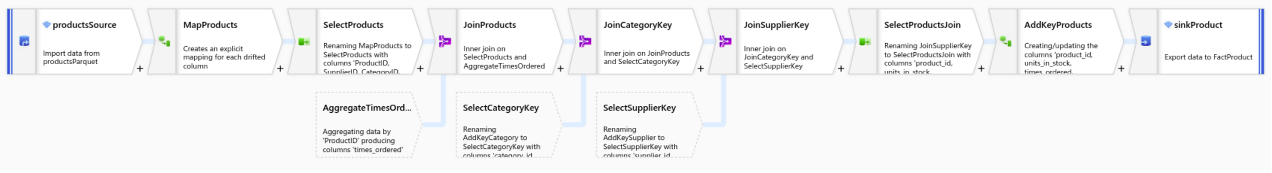
Here we split the stream with the "New Branch"-option. One of the branches is used to just select the previously defined key and the ID. We use that branch to join the key to the fact tables later. The second branch is writing the data to storage as parquet files.

## 3.2. The Fact Tables

The purpose of the fact tables is basically the aggregation of information that is used for data analytics in the data warehouse. Furthermore, each Fact table needs the keys from the associated dimension tables. Therefore we need two additional ingredients besides the general streaming structure:

- Join of aggregated values
- Join of technical keys for referencing to dimensions

The rest is similar (Read, Map, Select, Add key and timestamp, Write).


productsSource
Import data from productsParquet
MapProducts
Creates an explicit mapping for each drifted column
SelectProducts
Renaming MapProducts to SelectProducts with columns 'ProductID, SupplierID, CategoryID,
JoinProducts
Inner join on SelectProducts and AggregateTimesOrdered
JoinCategoryKey
Inner join on JoinProducts and SelectCategoryKey
JoinSupplierKey
Inner join on JoinCategoryKey and SelectSupplierKey
SelectProductsJoin
Renaming JoinSupplierKey to SelectProductsJoin with columns 'product_id, units_in_stock,
AddKeyProducts
Creating/updating the columns 'product_id, units_in_stock, times_ordered,
sinkProduct
Export data to FactProduct
AggregateTimesOrd...
Aggregating data by 'ProductID' producing columns 'times_ordered'
SelectCategoryKey
Renaming AddKeyCategory to SelectCategoryKey with columns 'category_id,
SelectSupplierKey
Renaming AddKeySupplier to SelectSupplierKey with columns 'supplier_id,

### 3.2.1. Joining Technical Keys

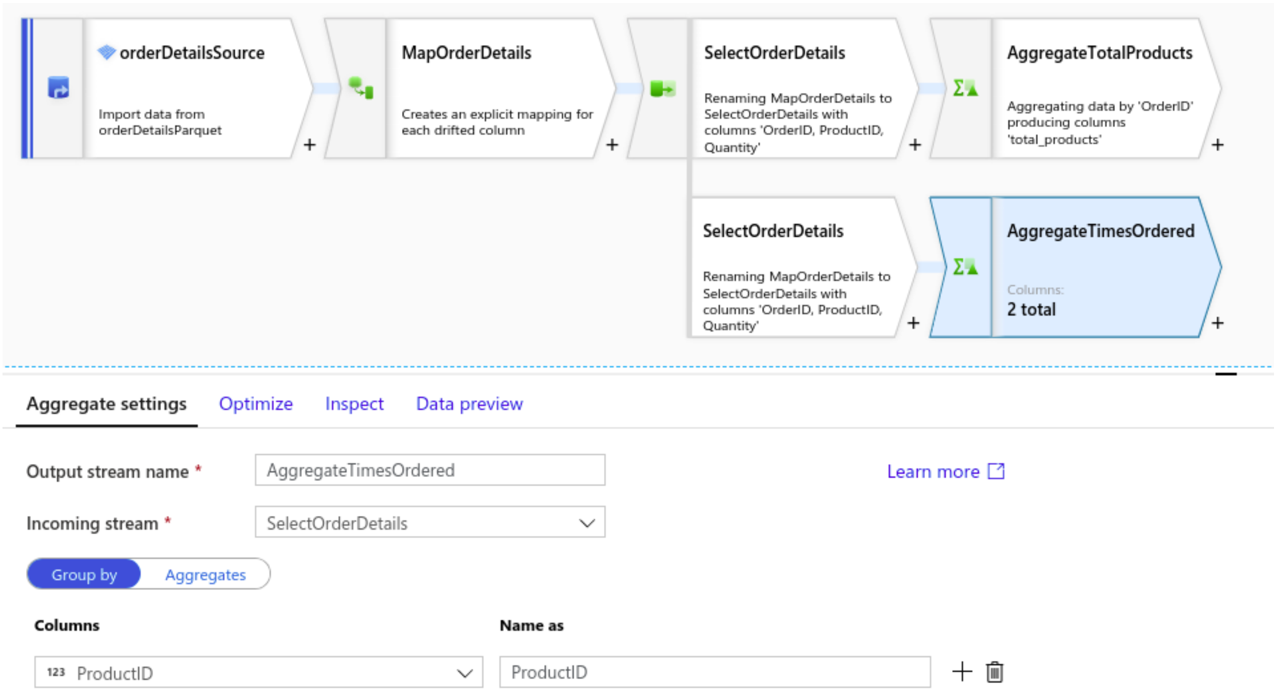
Here we add the technical key from the dimension table to the fact table by joining on the according ID. Another possibility would be to recreate the hash with the same hash function. However, one would need to join the whole data that is needed for the hashing, resulting in less performance. For the Join type, we use the inner join. It does not really matter in our case, whether it is a left join or an inner join. However, the inner join might be the more fault tolerant solution. One could imagine the case that a dimension table is missing some entries by mistake, which might lead to subsequent errors. These cases would be filtered out by the inner join.

### 3.2.2. Joining the Aggregated Data

These steps bring in the whole business logic of the fact tables. However, the join itself is trivial. One just has to choose between a left join or an inner join, which again depends on your business needs and how you process data afterward. We choose the inner join. The aggregation itself is done in separate streams depending on which data you want to aggregate.

Typically they follow the same principles: Read, Map, Select but the last step is the aggregation. If you need to aggregate data for various fact tables you have to branch your data as many times as you have columns to group by. In our case, we want to aggregate order Details, but we need them for two fact tables, namely:

- How many items were ordered in a certain order (group by order)?
- How many items were ordered for a certain product (group by product)?


orderDetailsSource
Import data from orderDetailsParquet
MapOrderDetails
Creates an explicit mapping for each drifted column
SelectOrderDetails
Renaming MapOrderDetails to SelectOrderDetails with columns 'OrderID, ProductID, Quantity'
AggregateTotalProducts
Aggregating data by 'OrderID' producing columns 'total_products'
SelectOrderDetails
Renaming MapOrderDetails to SelectOrderDetails with columns 'OrderID, ProductID, Quantity'
AggregateTimesOrdered
Columns:
2 total
Aggregate settings
Optimize
Inspect
Data preview
Learn more
Group by
Aggregates

Output stream name * AggregateTimesOrdered

Incoming stream * SelectOrderDetails

| Columns | Name as |
|---|---|
| 123 ProductID | ProductID |

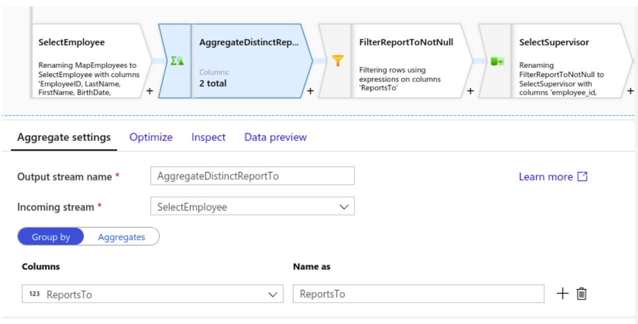
Joining the Aggregated Data

## 3.3. The Employee Stream (Mixture of Dim and Fact)

In our example, the stream of the employee may appear awfully chaotic. The reason is that we are creating four tables from it. Obviously one could also have four sources, but that would need four reads and might cost us performance.

Here we create the Dimension Employee as well as the Fact Employee, Fact Employee Monthly and Fact Supervisor in one stream. However, the principles are all the same: Read, Map, Select, Split and Add Keys and timestamp. For the facts, we also need to join the aggregated data. That's it. There are just two specialities:

1. Fact Supervisor: This table is a subsample of the employees, which is why we need to filter them. Furthermore, that Fact Table creates its own aggregates. We do not join any other aggregates, but aggregate on the "ReportsTo" column. However, it is exactly that aggregation that is also doing most of the filtering. The filter just drops the null values. You have to use the "!"-operator (!isNull()) as there is no isNotNull() function.

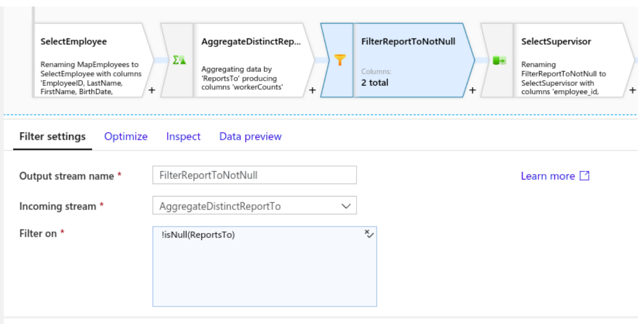

SelectEmployee
Renaming MapEmployees to SelectEmployee with columns 'EmployeeID, LastName, FirstName, BirthDate,
AggregateDistinctRep...
Columns:
2 total
FilterReportToNotNull
Filtering rows using expressions on columns 'ReportsTo'
SelectSupervisor
Renaming FilterReportToNotNull to SelectSupervisor with columns 'employee_id,
Aggregate settings
Optimize
Inspect
Data preview
Learn more
Group by
Aggregates

Output stream name * AggregateDistinctReportTo

Incoming stream * SelectEmployee

| Columns | Name as |
|---|---|
| 123 ReportsTo | ReportsTo |

Create own aggregates


SelectEmployee
Renaming MapEmployees to SelectEmployee with columns 'EmployeeID, LastName, FirstName, BirthDate,
AggregateDistinctRep...
Aggregating data by 'ReportsTo' producing columns 'workerCounts'
FilterReportToNotNull
Columns:
2 total
SelectSupervisor
Renaming FilterReportToNotNull to SelectSupervisor with columns 'employee_id,
Filter settings
Optimize
Inspect
Data preview
Learn more

Output stream name * FilterReportToNotNull

Incoming stream * AggregateDistinctReportTo

Filter on * !isNull(ReportsTo)

Filter the aggregated values

Fact Supervisor

2. Fact Employee and Employee Monthly: Here we cannot just join all aggregated features. Instead, we need to join partially aggregated stuff and aggregate it. For example, we need to know the distinct count of ProductIDs per employee. Such distinct counts cannot be aggregated in their origin tables. Furthermore, we need to join the orders aggregate with an inner join, which is creating the filtering mechanism on whether it is only last month's data or the full data set. This is, however, not possible when you want to run the monthly aggregation on a different schedule than the complete aggregation. In this case, you would need to have completely separated pipelines.

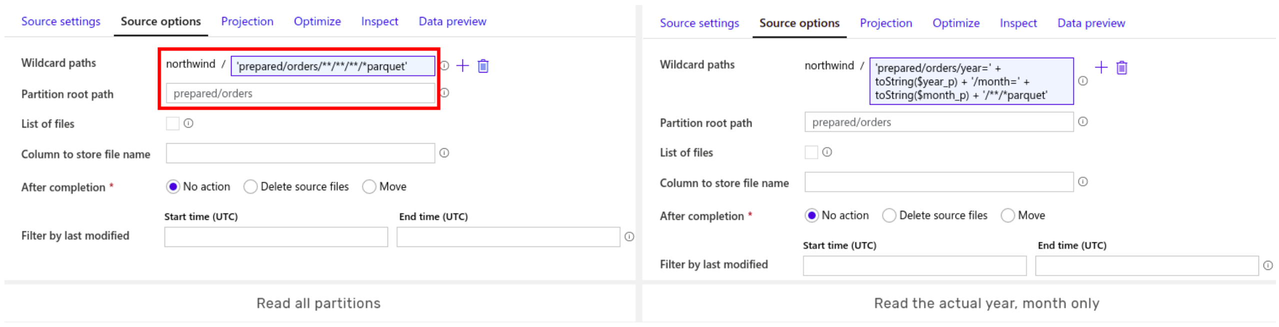
## 3.4. The Partitioned Orders Aggregates

The last thing that we need to handle is how to read partitioned data. Here we read the orders, which are partitioned by year, month, and day. We need them for full aggregates and monthly aggregates.

Therefore, we either read every partition or just the specific year and month. There are two ways of doing it. Either just read the data once, split the stream and filter one of the branches on the needed year and month. Or read the data twice, once with every piece of data and once with only one partition. It would not matter much in performance as the data is anyhow partitioned on that.

Here we choose to read data in two different streams. The way to treat partitioned parquet data is to set up a wildcard path and a partition root path. In our case, we just need to add the parameters year and month to the wildcard path.

This is all that is needed for the aggregation. However, there is the last section on the triggering of the pipeline.



| | |
|---|---|
| Wildcard paths | northwind / 'prepared/orders/**/**/**/*parquet' |
| Partition root path | prepared/orders |
| List of files | |
| Column to store file name | |
| After completion * | [✓ No action] Delete source files Move |
| Filter by last modified | Start time (UTC) / End time (UTC) |

Read all partitions

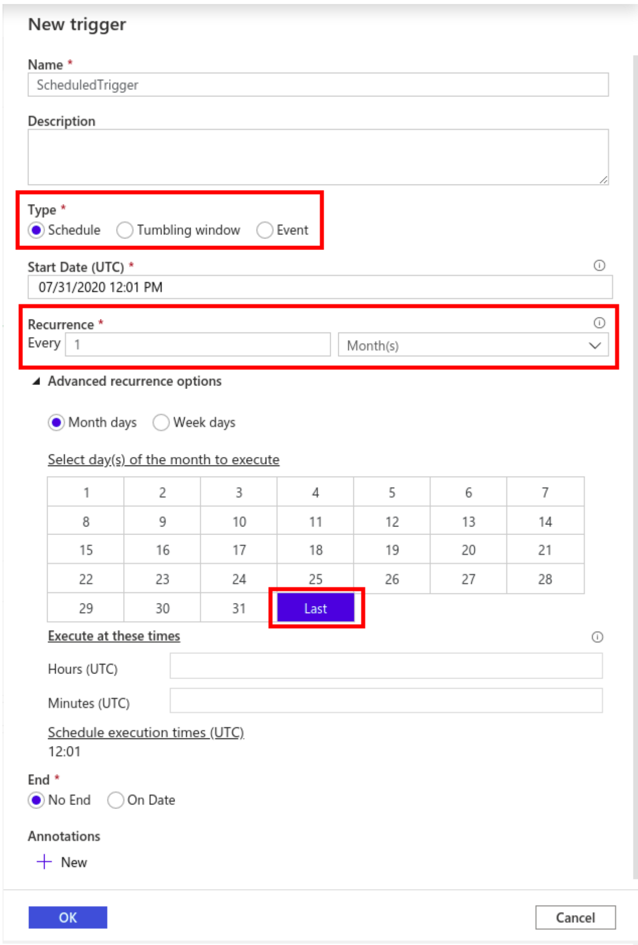
| | |
|---|---|
| Wildcard paths | northwind / 'prepared/orders/year=' + toString($year_p) + '/month=' + toString($month_p) + '/**/*parquet' |
| Partition root path | prepared/orders |
| List of files | |
| Column to store file name | |
| After completion * | [✓ No action] Delete source files Move |
| Filter by last modified | Start time (UTC) / End time (UTC) |

Read the actual year, month only

The Partitioned Orders Aggregates

## 4. The Triggering

Using Triggers is a way to manage and schedule the run of your workflows. However, the triggering is not yet fully matured in the data factory. There are three different types of triggers to choose from: event, scheduled, and tumbling window. The event trigger can fire when a blob is created or deleted in a certain container, which is not what we want in our case.



New trigger

Name *: ScheduledTrigger

Description:

Type *: [✓ Schedule] Tumbling window Event

Start Date (UTC) *: 07/31/2020 12:01 PM

Recurrence *: Every 1 Month(s)

Advanced recurrence options

[✓ Month days] Week days

Select day(s) of the month to execute

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| 22 | 23 | 24 | 25 | 26 | 27 | 28 |
| 29 | 30 | 31 | [✓ Last] |  |  |  |

Execute at these times

Hours (UTC):

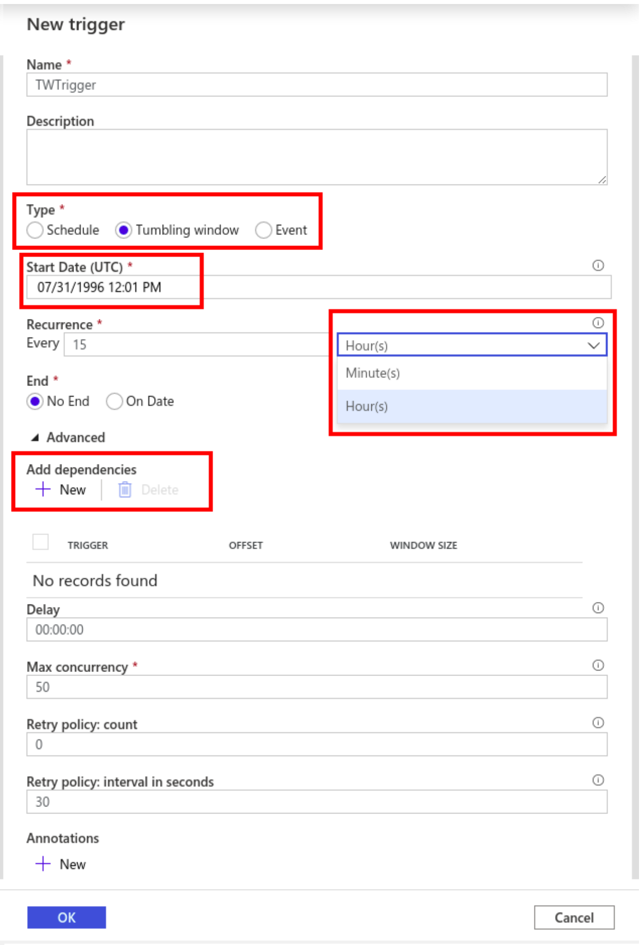
Minutes (UTC):

Schedule execution times (UTC): 12:01

End *: [✓ No End] On Date

Annotations

Scheduled Trigger and its trigger variable



New trigger

Name *: TWTrigger

Description:

Type *: Schedule [✓ Tumbling window] Event

Start Date (UTC) *: 07/31/1996 12:01 PM

Recurrence *: Every 15 Hour(s)

End *: [✓ No End] On Date

Advanced

Add dependencies

| TRIGGER | OFFSET | WINDOW SIZE |
|---|---|---|

No records found

Delay: 00:00:00

Max concurrency *: 50

Retry policy: count: 0

Retry policy: interval in seconds: 30

Annotations

Tumbling Window Trigger and its trigger variable

The Triggering

## 4.1. Scheduled Trigger

The scheduled trigger might be a good choice as you can clearly define a frequency (in minutes, hours, days, weeks, months) in which you want to execute your pipeline. One can hand over the trigger start time to the pipeline via the system variable "@trigger().scheduledTime".

And it is also possible to fine tune the trigger by setting a certain day of month or day of the week. Therefore, it seems to be a perfect choice. However, the scheduled trigger does not work for any backfilling scenarios. If you enter a date that is in the past, it is not executing the pipeline. As our toy data set includes data from the 90's, we can not get that data directly by using the scheduled trigger. In contrast, the tumbling window trigger is capable of handling backfilling scenarios.

## 4.2. Tumbling Window Trigger

Besides handling backfilling scenarios, the Tumbling Window trigger has various other options such as dependencies on other triggers (also self-dependency) or retry policies.

One can hand over the trigger start time to the pipeline via the system variable "@trigger().outputs.windowStartTime". Unfortunately, it is very limited in the choice of frequency, only allowing to choose between minutes and hours. Further, it does not support to trigger every first or last day of a month or running it on several specific days of the week. Therefore, a scenario like that would have to be solved with a workaround solution such as triggering it every 24 hours but adding in an If-Condition activity which evaluates whether it is the last day of a month.

**Wrapping It Up**

Bringing it all together, we presented a detailed solution of a typical ETL procedure with the Azure tool set. We started by creating resources and pre-processing data. We then explained the details of a certain data warehouse schema on Azure Synapse.

As a final ingredient, we showed a detailed ETL procedure with data factory's data flow. If you liked that blog, stay tuned. Next time we will report on the data analytics with PowerBI. Cheers!

**Sources**

- Azure Docs: Information on pipeline triggers
- Azure Docs: Concept of Data flow
- Azure Docs: Data flow expression functions
- Azure Docs: Data flow parametrization