

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

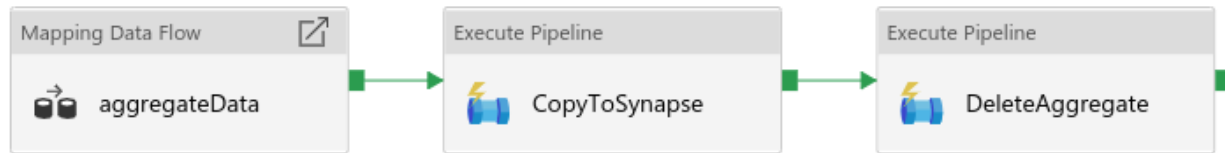
The basic idea of our ETL pipeline is to load the up-to-date Dimension tables on a monthly basis, add a technical key to them, dump them to Azure Data Lake Storage (ADLS), and copy them from ADLS to the DWH.

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.



Parameters Variables Output

+ New | Delete

<input type="checkbox"/>	NAME	TYPE	DEFAULT VALUE
<input type="checkbox"/>	date	String	1997-12-31

The Pipeline Overview

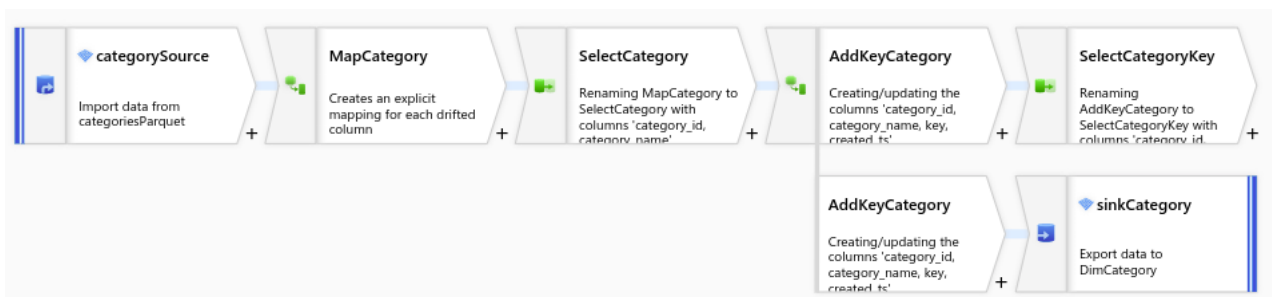
We will set a pipeline parameter "date" of type string that we will initialize with an arbitrary default value like '1997-12-31'. The idea of that parameter is to hand over a trigger parameter, namely the scheduled execution timestamp. This parameter is used in the aggregation, where we only read data 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.



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).

The left screenshot shows the 'Source settings' tab with the following configuration:

- Wildcard paths:** northwind / [prepared/categories/year= ' + toString(\$year.p) + '/month= ' + toString(\$month.p) + '*/parquet]
- Partition root path:** northwind /
- List of files:** ☐
- Column to store file name:**
- After completion *:** ☒ No action ☐ Delete source files ☐ Move
- Start time (UTC):**
- End time (UTC):**

The right screenshot shows the 'Data preview' tab with the following table:

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

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.

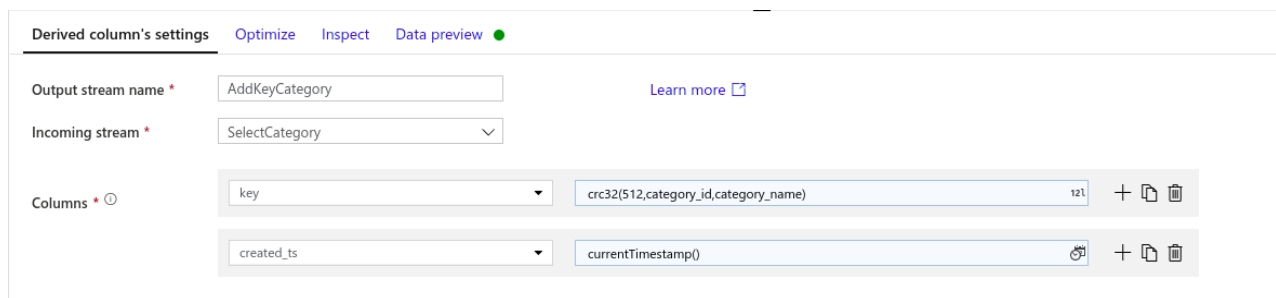
In the case of our dimension processing, we have very short streams. It would not make sense to have a selection at the beginning and one at the end. Therefore we have just one.

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 stream and 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.



Adding Columns

3.1.4. Splitting and Sinking

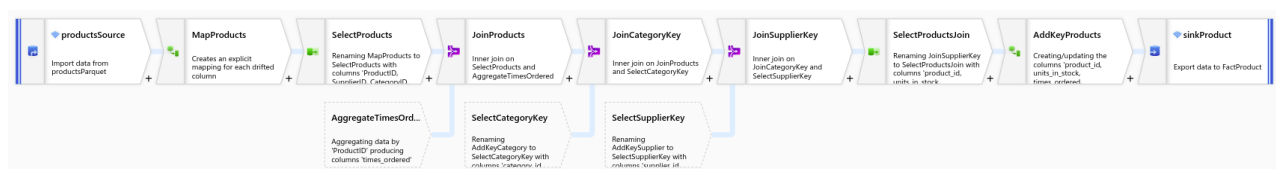
Here we split the stream with the "New Branch"-option. One of the branches is used to just select the previously defined key and the ID. We use that branch to join the key to the fact tables later. The second branch is writing the data to storage as parquet files.

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).



3.2.1. Joining Technical Keys

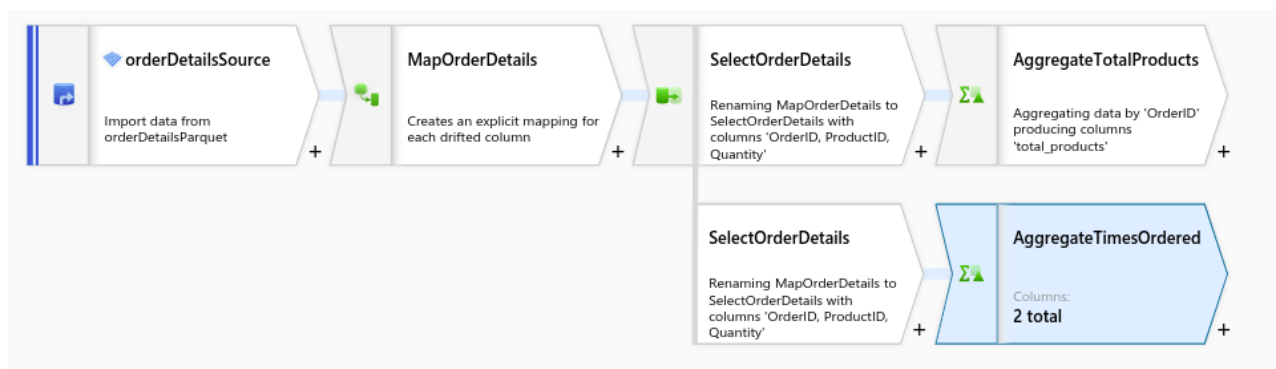
Here we add the technical key from the dimension table to the fact table by joining on the according ID. Another possibility would be to recreate the hash with the same hash function. However, one would need to join the whole data that is needed for the hashing, resulting in less performance. For the 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)?



Aggregate settings Optimize Inspect Data preview

Output stream name * AggregateTimesOrdered [Learn more](#)

Incoming stream * SelectOrderDetails

Group by Aggregates

Columns

Name as

123 ProductID ProductID +

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.

The screenshot displays two configuration panels for a data pipeline. The left panel, titled 'Aggregate settings', shows a stage named 'AggregateDistinctReportTo' with 'Columns: 2 total'. It has an incoming stream 'SelectEmployee' and an output stream 'AggregateDistinctReportTo'. The 'Group by' section is set to 'Aggregates'. The 'Columns' section shows 'ReportsTo' being mapped to 'ReportsTo'. The right panel, titled 'Filter settings', shows a stage named 'FilterReportToNotNull' with 'Columns: 2 total'. It has an incoming stream 'AggregateDistinctReportTo' and an output stream 'FilterReportToNotNull'. The 'Filter on' section is set to '!isNull(ReportsTo)'. Both panels include 'Optimize', 'Inspect', and 'Data preview' tabs.

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.

The image displays two side-by-side screenshots of the Databricks Source options configuration interface. Both screenshots show the 'Source options' tab with the following fields: Wildcard paths, Partition root path, List of files, Column to store file name, After completion, and Filter by last modified. The left screenshot is titled 'Read all partitions' and shows the Wildcard paths field with the value 'northwind / 'prepared/orders/**/*/*'parquet''. The right screenshot is titled 'Read the actual year, month only' and shows the Wildcard paths field with the value 'northwind / 'prepared/orders/year=' + toString(\$year_p) + '/month=' + toString(\$month_p) + '/*'parquet'.

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

+ New

OK

Cancel

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)

Minute(s)

Hour(s)

End *

☒ No End
☐ On Date

Advanced

Add dependencies

+ New

Delete

☐ 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

+ New

OK

Cancel

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](#)