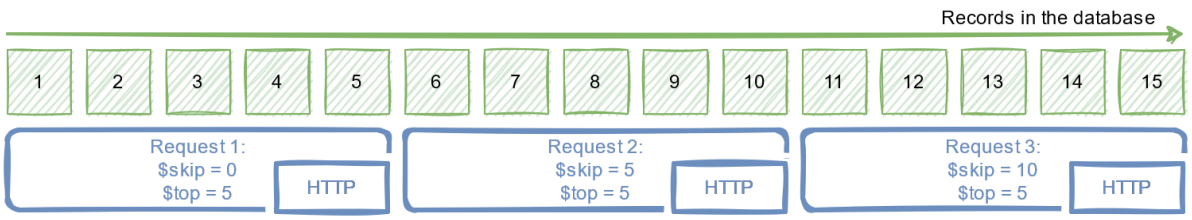
parameter that indicates the first record to fetch. So, to process 15 records, you can send a single request, or we can chunk it into smaller pieces using the combination of $skip and $top parameters.



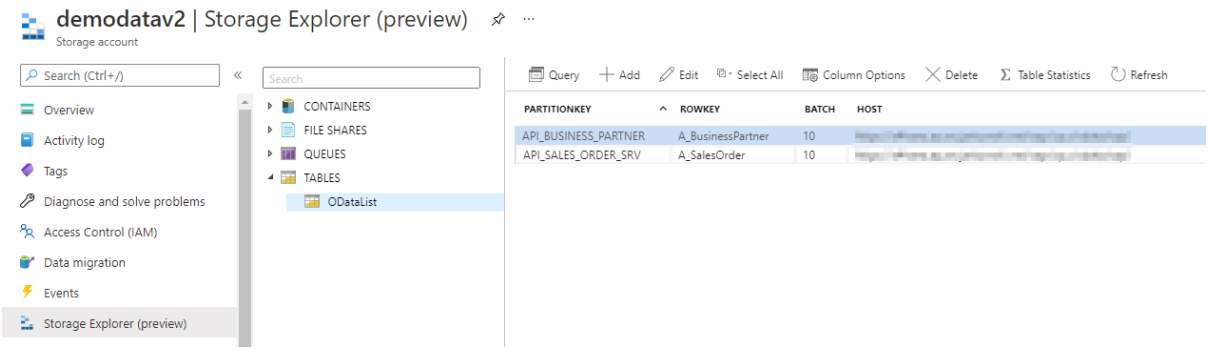

As sending a single request asking for large amounts of data is not the best approach, a similar danger comes from flooding the SAP system with a large number of tiny calls. Finding the right balance is the key!

The above approach is called Client-Side Paging. We will use the logic inside Synapse Pipeline to split the dataset into manageable pieces and then extract each of them. To implement it in the pipeline, we need three numbers:

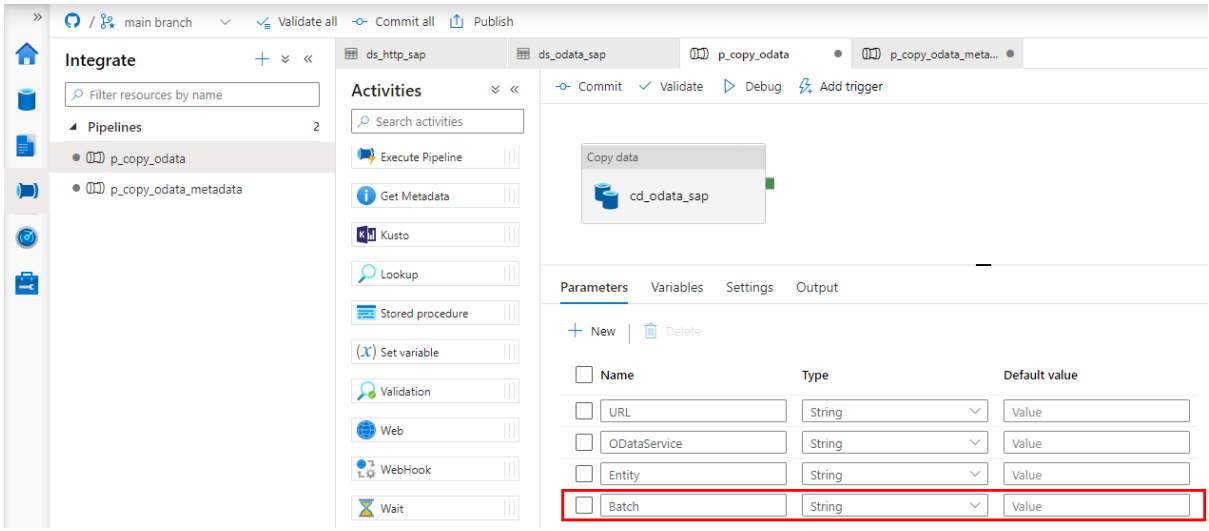
- The number of records in the OData service
- Amount of data to fetch in a single request (batch size)
- Number of requests

Getting the number of records in the OData service is simple. You can use the $count option passed at the end of the URL. By dividing it by the batch size, which we define for each OData service and store in the metadata table, we can calculate the number of requests required to fetch the complete dataset.

Open the Storage Explorer to alter the metadata table and add a new Batch property:

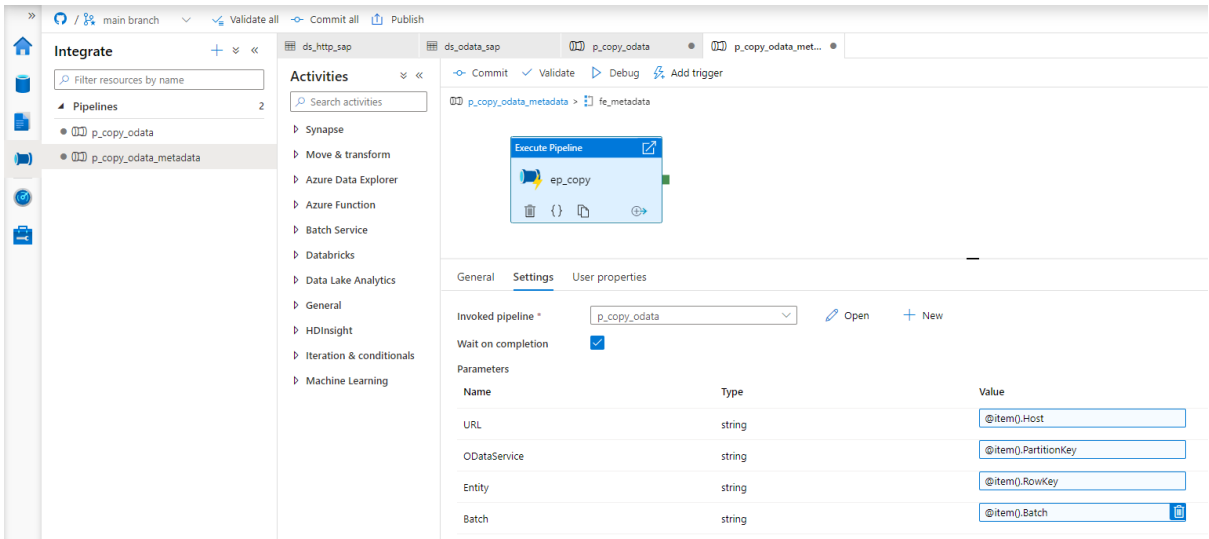



Now go to Synapse Studio and open the child pipeline. Add a new parameter to store the Batch size:




In the metadata pipeline, open the Execute Pipeline activity. The following expression will pass the batch size value from the metadata table. You don't have to make any changes to the Lookup.

```
@item().Batch
```

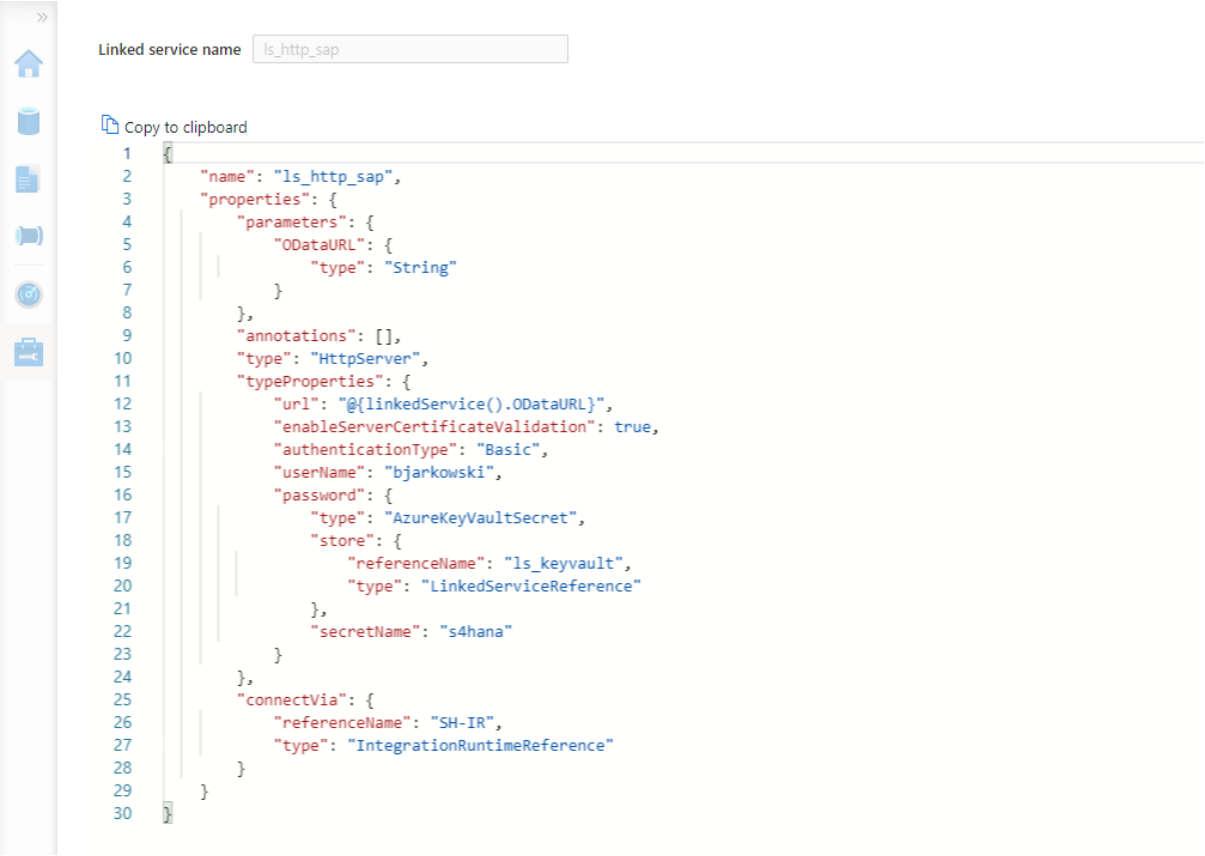



There is a couple of ways to read the record count. Initially, I wanted to use the Lookup activity against the dataset that we already have. But, as the result of a $count is just a number without any data structure, the OData connector fails to interpret the value. Instead, we have to create another Linked Service and a dataset of type HTTP. It should point to the same OData Service as the Copy Data activity.

Create the new Linked Service of type HTTP. It should accept the same parameters as the OData one. Refer to the second episode of the blog series if you'd like to refresh your memory on how to add parameters to linked services.

```
{
    "name": "ls_http_sap",
    "properties": {
```

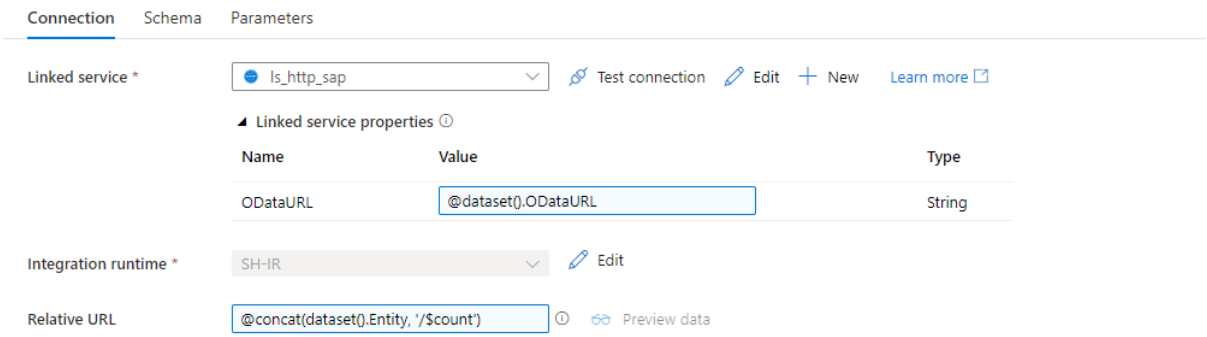
```
        "parameters": {
            "ODataURL": {
                "type": "String"
            }
        },
        "annotations": [],
        "type": "HttpServer",
        "typeProperties": {
            "url": "@{linkedService().ODataURL}",
            "enableServerCertificateValidation": true,
            "authenticationType": "Basic",
            "userName": "bjarkowski",
            "password": {
                "type": "AzureKeyVaultSecret",
                "store": {
                    "referenceName": "ls_keyvault",
                    "type": "LinkedServiceReference"
                },
                "secretName": "s4hana"
            }
        },
        "connectVia": {
            "referenceName": "SH-IR",
            "type": "IntegrationRuntimeReference"
        }
    }
}
```




Now, let's create the dataset. Choose HTTP as the type and DelimitedText as the file format. Add ODataURL and Entity parameters as we did for the OData dataset. On the Settings tab, you'll find the field Relative URL, which is the equivalent of the Path from the OData-based

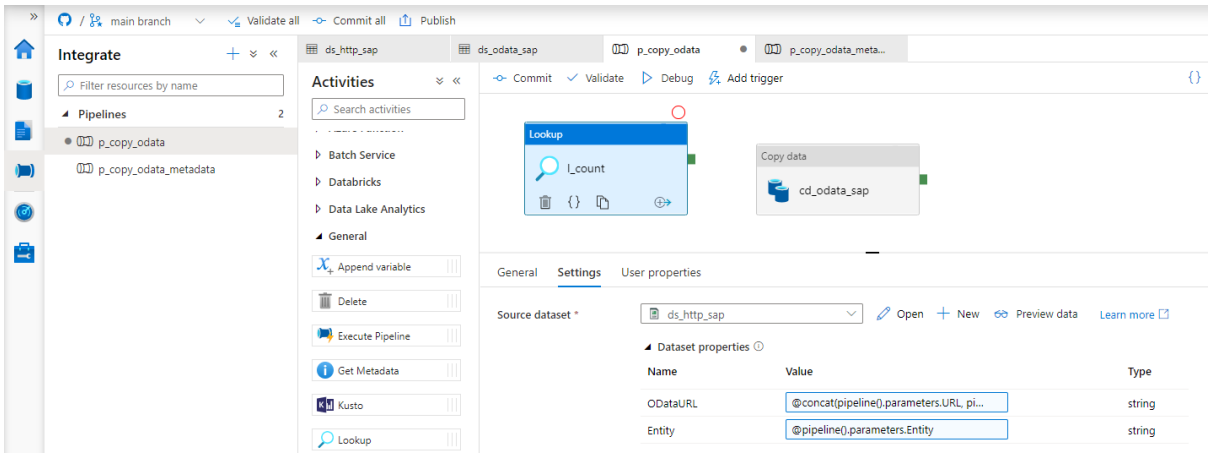
dataset. To get the number of records, we have to concatenate the entity name with the $count. The expression should look as follows:

```
@concat(dataset().Entity, '/$count')
```




Perfect! We can now update the child pipeline that processes each OData service. Add the Lookup activity, but don't create any connection. Both parameters on the Settings tab should have the same expression as the Copy Data activity. There is one difference. It seems there is a small bug and the URL in the Lookup activity has to end with a slash '/' sign. Otherwise, the last part of the address may be removed, and the request may fail.

```
ODataURL: @concat(pipeline().parameters.URL,
pipeline().parameters.ODataService, '/')
Entity: @pipeline().parameters.Entity
```




Difficult moment ahead of us! I'll try to explain all the details the best I can. When the Lookup activity checks the number of records in the OData service, the response contains just a single value. We will use the $skip and $top query parameters to chunk the request into smaller pieces. The tricky part is how to model it in the pipeline. As always, there is no single solution. The easiest approach is to use the Until loop, which could check the number of processed rows at every iteration. But it only allows sequential processing, and I want to show you a more robust way of extracting data.

The ForEach loop offers parallel execution, but it only accepts an array as the input. We have to find a way on how to create one. The @range() expression can build an array of consecutive numbers. It accepts two parameters – the starting position and the length,

which in our case will translate to the number of requests. Knowing the number of records and the batch size, we can easily calculate the array length. Assuming the OData service contains 15 elements and the batch size equals 5, we could pass the following parameters to the @range() function:

```
@range(0,3)
```

As the outcome we receive:

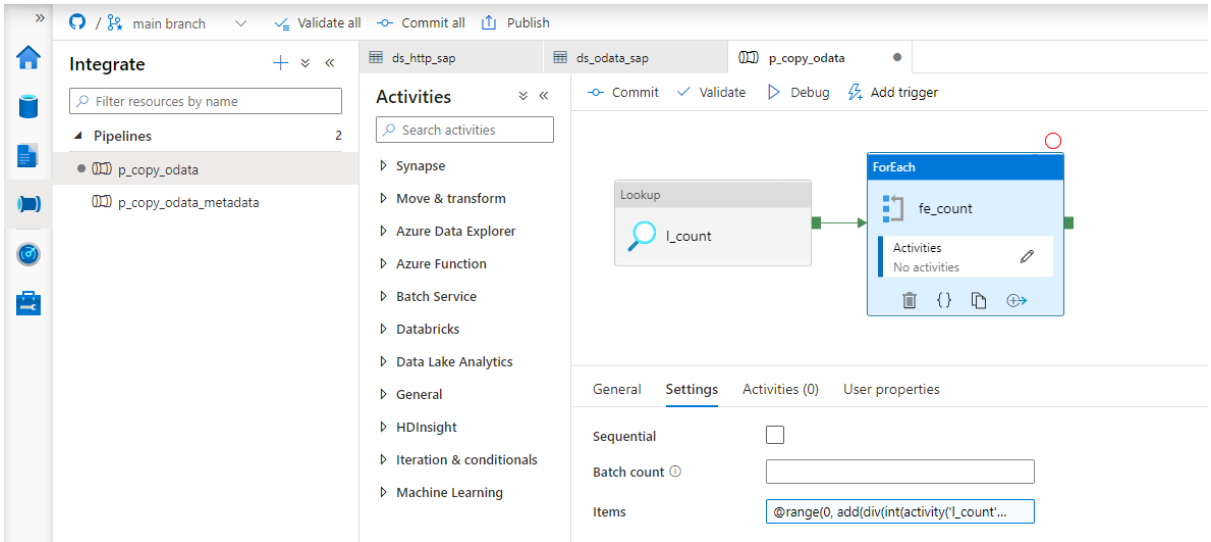
```
[0,1,2]
```

Each value in the array represents the request number. Using it, we can easily calculate the $skip parameter.

But there is one extra challenge. What if the number of records cannot be divided by the batch size without a remainder? As there is no rounding function, the decimal part of the result will be omitted, which means we're losing the last chunk of data. To avoid that, I implemented a simple workaround – I always add 1 to the number of requests. Of course, you could think about a fancy solution using the modulo function, but I'm a big fan of simplicity. And asking for more data won't hurt.

Add the ForEach loop as the successor of the Lookup activity. In the Items field, provide the following expression to create an array of requests. I'm using the int() function to cast the string value to the integer that I can then use in the div().

```
@range(0, add(div(int(activity('l_count').output.firstRow.Prop_0),
int(pipeline().parameters.Batch)),1))
```

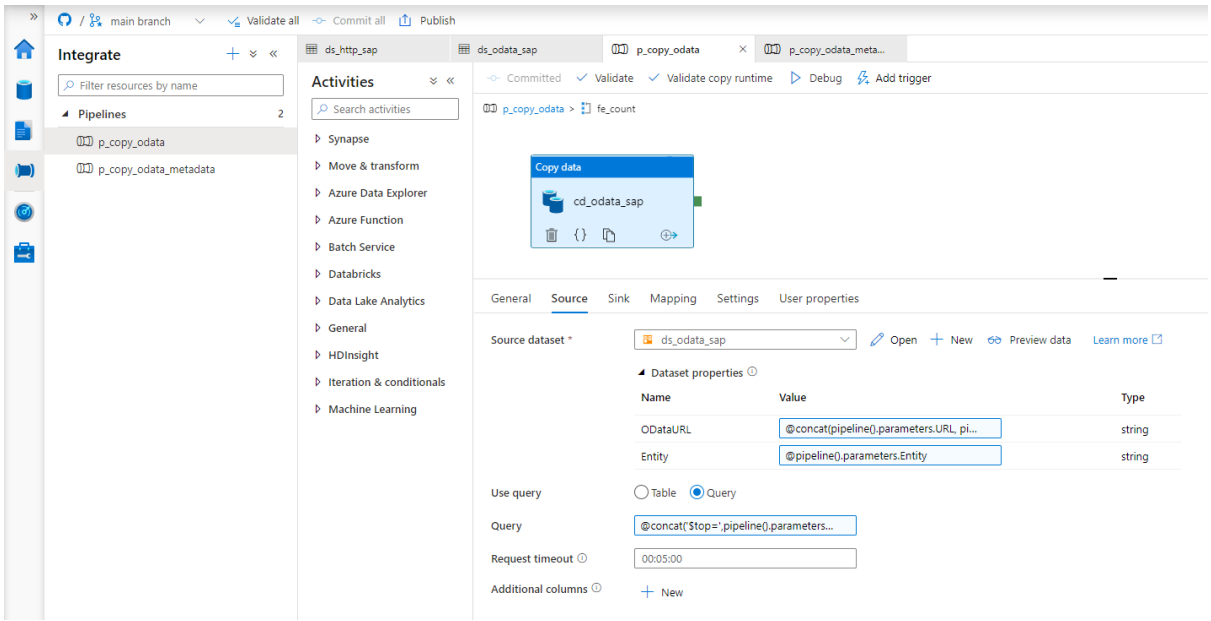



To send multiple requests to the data source, move the Copy Data activity to the ForEach loop. Every iteration will trigger a copy job – but we have to correctly maintain query parameters to receive just one chunk of data. To achieve it, we will use the $top and $skip parameters, as I mentioned earlier in the post.

The $top parameter is static and always equals the batch size. To calculate the $skip parameter, we will use the request number from the array passed to the loop multiplied by the batch size.

Open the Copy Data activity and go to the Settings tab. Change the field Use Query to Query and provide the following expression:

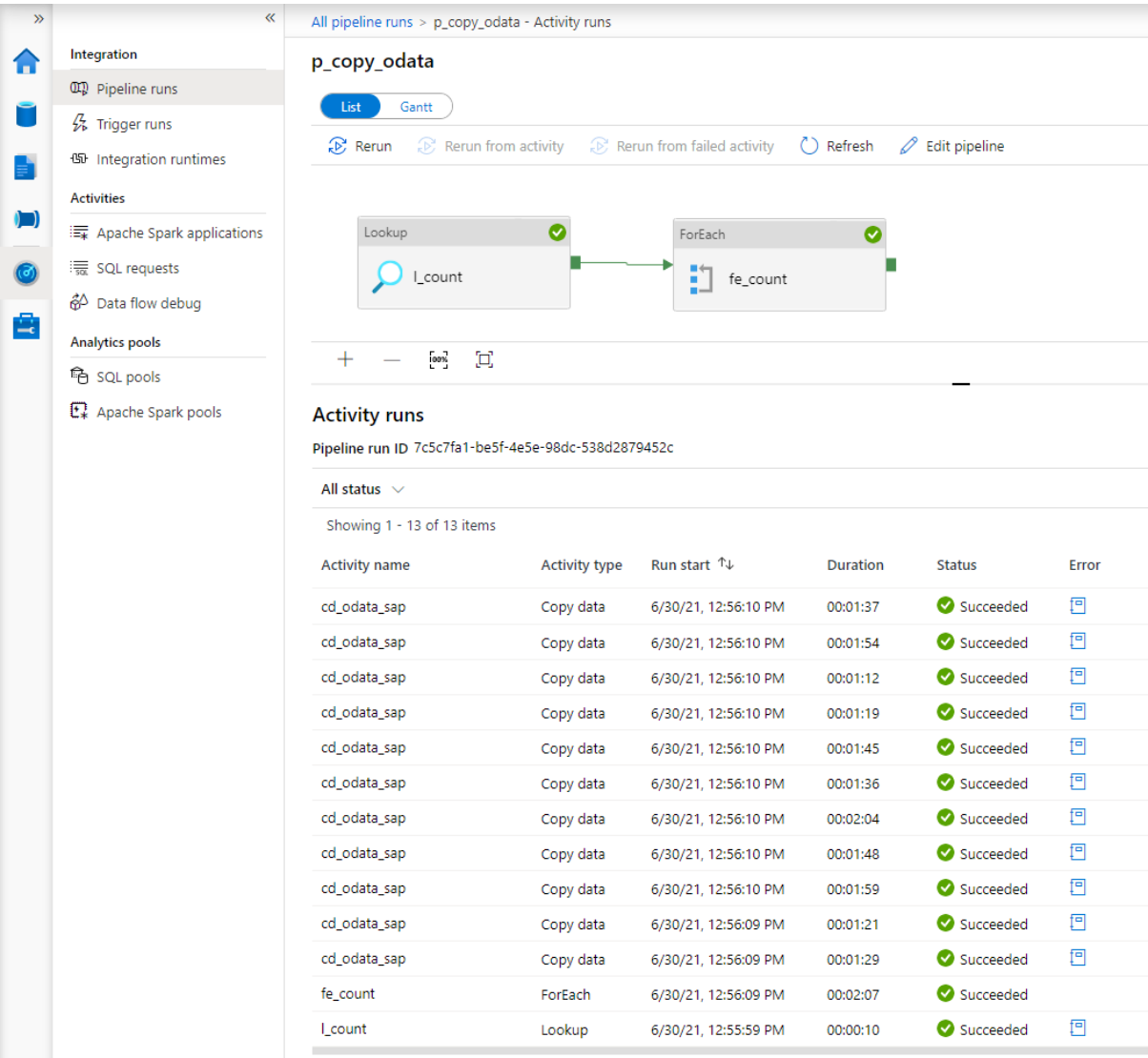
```
@concat('$top=',pipeline().parameters.Batch,
'&$skip=',string(mul(int(item().value), int(pipeline().parameters.Batch))))
```




That was the last change to make. Let's start the pipeline!

## EXECUTION AND MONITORING

Once the pipeline processing finishes we can see succesfully completed jobs in the monitoring view. Let's drill down to see the details.




Comparing to the previous extraction, you can see the difference. Instead of just one, there are now multiple entries for the Copy Data activity. You may be slightly disappointed with the duration of each copy job. It takes much longer to extract every chunk of data – in the previous episode, it took only 36 seconds to extract all sales orders. This time every activity took at least a minute.

There is a couple of reasons why it happens. Let’s take a closer a closer look at the components of the extraction job to understand why the duration increased heavily.

Details Refresh

Learn more on copy performance details from here.

**Activity run id:** 61ba231a-62a4-421c-8cfa-0d4a2b419a63

OData-Preview → Succeeded → Azure Data Lake Storage Gen2
Region: West Europe

| | | | |
|---|---|---|---|
| Data read: | 18.604 KB | Data written: | 7.136 KB |
| Rows read: | 10 | Files written: | 1 |
| Peak connections: | 1 | Rows written: | 10 |
| | | Peak connections: | 1 |

| | |
|---|---|
| Copy duration | 00:01:43 |
| Throughput: | 185.344 bytes/s |

OData-Preview → Azure Data Lake Storage Gen2

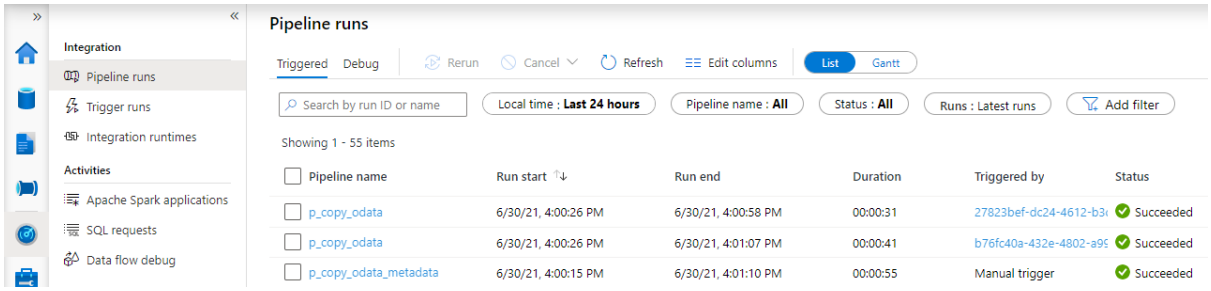
| | |
|---|---|
| Start time | 6/30/21, 3:38:34 PM |
| Used parallel copies | 1 |
| Duration | 00:01:43 |

| Details | | Working duration | Total duration |
|---|---|---|---|
| Queue | | | 00:01:29 |
| Transfer | Time to first byte | 00:00:00 | 00:00:09 |
| | Reading from source | 00:00:00 | |
| | Writing to sink | 00:00:01 | |

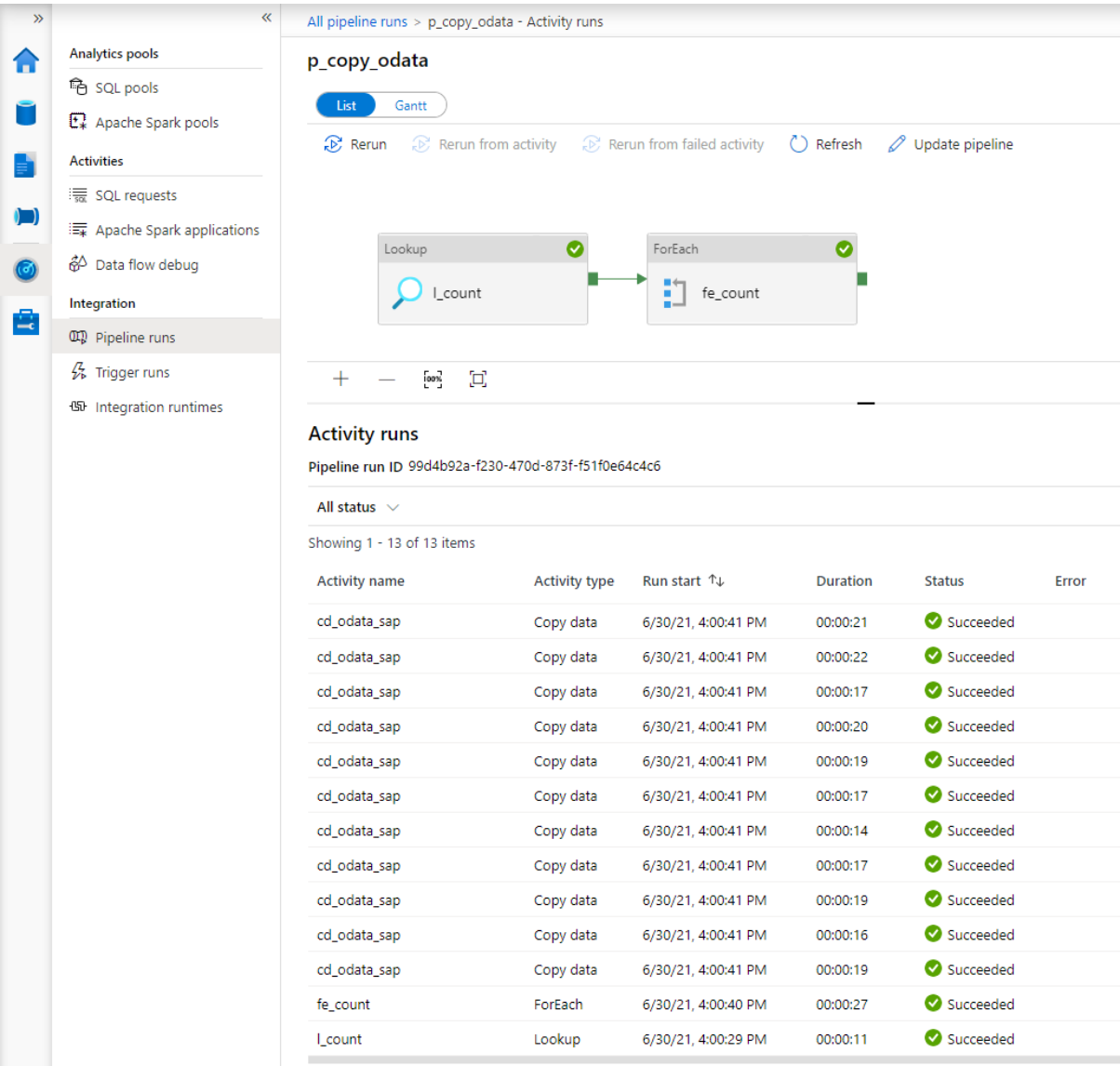
Data consistency verification Not verified

Look at the time analysis. Before the request was processed, it was in the Queue for 1 minute and 29 seconds. Extracting data took only 9 seconds. Why there is such a long wait time?

In the first episode of the blog series, I briefly explained the role of the integration runtime. It provides the computing resources for pipeline execution. To save cost, I host my integration runtime on a very tiny B2ms virtual machine. It provides enough power to process two or three activities at the same time, which means that extracting many chunks is rather sequential than parallel. To fix that, I’ve upgraded my virtual machine to a bigger one. The total duration to extract data decreased significantly as I could process more chunks at the same time.



Here is the duration of each Copy Data activity.




| Activity name | Activity type | Run start | Duration | Status | Error |
|---|---|---|---|---|---|
| cd_odata_sap | Copy data | 6/30/21, 4:00:41 PM | 00:00:21 | Succeeded | |
| cd_odata_sap | Copy data | 6/30/21, 4:00:41 PM | 00:00:22 | Succeeded | |
| cd_odata_sap | Copy data | 6/30/21, 4:00:41 PM | 00:00:17 | Succeeded | |
| cd_odata_sap | Copy data | 6/30/21, 4:00:41 PM | 00:00:20 | Succeeded | |
| cd_odata_sap | Copy data | 6/30/21, 4:00:41 PM | 00:00:19 | Succeeded | |
| cd_odata_sap | Copy data | 6/30/21, 4:00:41 PM | 00:00:17 | Succeeded | |
| cd_odata_sap | Copy data | 6/30/21, 4:00:41 PM | 00:00:14 | Succeeded | |
| cd_odata_sap | Copy data | 6/30/21, 4:00:41 PM | 00:00:17 | Succeeded | |
| cd_odata_sap | Copy data | 6/30/21, 4:00:41 PM | 00:00:19 | Succeeded | |
| cd_odata_sap | Copy data | 6/30/21, 4:00:41 PM | 00:00:16 | Succeeded | |
| cd_odata_sap | Copy data | 6/30/21, 4:00:41 PM | 00:00:19 | Succeeded | |
| fe_count | ForEach | 6/30/21, 4:00:40 PM | 00:00:27 | Succeeded | |
| l_count | Lookup | 6/30/21, 4:00:29 PM | 00:00:11 | Succeeded | |

The request is in the queue only for a couple of seconds instead of over a minute.

Details Refresh

Learn more on copy performance details from here.

Activity run id: d0a131d6-e6c4-4d79-bb2c-b6161682e962

OData-Preview → Succeeded → Azure Data Lake Storage Gen2 Region: West Europe

| | | | |
|---|---|---|---|
| Data read: | 18.604 KB | Data written: | 7.17 KB |
| Rows read: | 10 | Files written: | 1 |
| Peak connections: | 1 | Rows written: | 10 |
| | | Peak connections: | 1 |

| | |
|---|---|
| Copy duration | 00:00:12 |
| Throughput: | 1.55 KB/s |

OData-Preview → Azure Data Lake Storage Gen2

| | |
|---|---|
| Start time | 6/30/21, 4:00:41 PM |
| Used parallel copies | 1 |
| Duration | 00:00:12 |

| Details | | Working duration | Total duration |
|---|---|---|---|
| Queue | | | 00:00:06 |
| Transfer | Time to first byte | 00:00:01 | 00:00:04 |
| | Reading from source | 00:00:00 | |
| | Writing to sink | 00:00:00 | |

Data consistency verification Not verified

Before we finish, I want to show you some results of processing a large dataset. In another system, I have over 1 million sales orders with more than 5 million line items. It wasn't possible to extract it all in a single request as every attempt resulted in a short dump at the SAP side. I've adjusted the batch size to 100 000 which should be alright to process at a time. To further optimize the extraction, I changed the processing of OData services to Sequential, which means the job firstly extracts Sales Order headers before moving to line items. You can do it in the ForEach Loop in the metadata pipeline. To limit the impact of extraction to the SAP system, I also set a concurrency limit for the Copy Data activity (ForEach loop in the child pipeline).

It took 25 minutes to extract in total over 6 million records. Not bad.

Pipeline runs

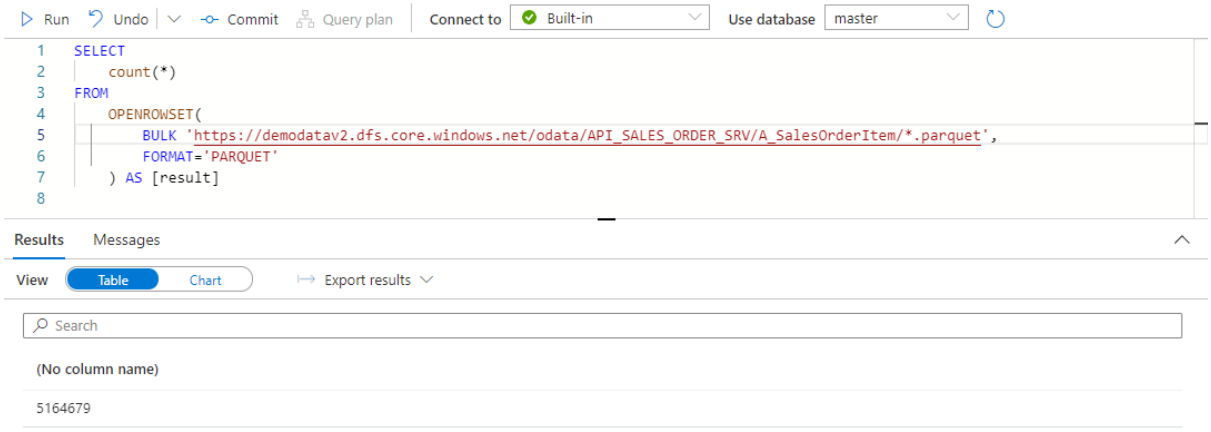
Triggered Debug Rerun Cancel Refresh Edit columns List Gantt

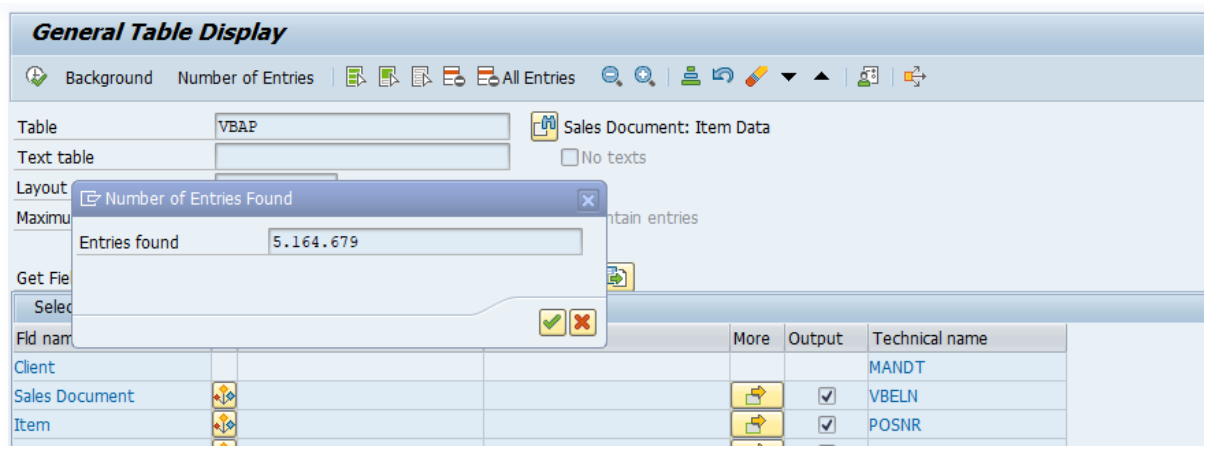
Search by run ID or name | Local time : **Last 24 hours** | Pipeline name : **All** | Status : **All** | Runs : Latest runs | Add filter

Showing 1 - 66 items

| Pipeline name | Run start | Run end | Duration | Triggered by | Status |
|---|---|---|---|---|---|
| p_copy_odata | 6/30/21, 4:58:42 PM | 6/30/21, 5:19:54 PM | 00:21:12 | 29649e49-ed4d-4ffc-9b5 | Succeeded |
| p_copy_odata | 6/30/21, 4:54:15 PM | 6/30/21, 4:58:41 PM | 00:04:25 | 5fa66676-d48e-4350-af5 | Succeeded |
| p_copy_odata_metadata | 6/30/21, 4:53:58 PM | 6/30/21, 5:19:56 PM | 00:25:58 | Manual trigger | Succeeded |

The extraction generated quite a lot of files on the data lake. Let's count all records inside them and compare the number with what I have in my SAP system:




Both numbers match, which means we have the full dataset in the data lake. We could probably further optimize the extraction duration by finding the right number of parallel processes and identifying the best batch size.

Finally, before you even start the extraction, I recommend checking if you need all data. Trimming the dataset by setting filters on columns or extracting only a subset of columns can heavily improve the extraction duration. That's something we'll cover in the next episode!

# Part 5 – Filter and Select

Over the last five weeks, we've built quite a complex Synapse Pipeline that allows extracting SAP data using OData protocol. Starting from a single activity in the pipeline, the solution grew, and it now allows to process multiple services on a single execution. We've implemented client-side caching to optimize the extraction runtime and eliminate short dumps at SAP. But that's definitely not the end of the journey!

Today we will continue to optimize the performance of the data extraction. Just because the Sales Order OData service exposes 40 or 50 properties, it doesn't mean you need all of them. One of the first things I always mention to customers, with who I have a pleasure working, is to carefully analyze the use case and identify the data they actually need. The less you copy from the SAP system, the process is faster, cheaper and causes fewer troubles for SAP application servers. If you require data only for a single company code, or just a few customers – do not extract everything just because you can. Focus on what you need and filter out any unnecessary information.

Fortunately, OData services provide capabilities to limit the amount of extracted data. You can filter out unnecessary data based on the property value, and you can only extract data from selected columns containing meaningful data.

Today I'll show you how to implement two query parameters: $filter and $select to reduce the amount of data to extract. Knowing how to use them in the pipeline is essential for the next episode when I explain how to process only new and changed data from the OData service.

## ODATA FILTERING AND SELECTION

To filter extracted data based on the field content, you can use the $filter query parameter. Using logical operators, you can build selection rules, for example, to extract only data for a single company code or a sold-to party. Such a query could look like this:

```
/API_SALES_ORDER_SRV/A_SalesOrder?$filter=SoldToParty eq 'AZ001'
```

The above query will only return records where the field SoldToParty equals AZ001. You can expand it with logical operators 'and' and 'or' to build complex selection rules. Below I'm using the 'or' operator to display data for two Sold-To Parties:

```
/API_SALES_ORDER_SRV/A_SalesOrder/?$filter=SoldToParty eq 'AZ001' or
SoldToParty eq 'AZ002'
```

You can mix and match fields we're interested in. Let's say we would like to see orders for customers AZ001 and AZ002 but only where the total net amount of the order is lower than 10000. Again, it's quite simple to write a query to filter out data we're not interested in:

```
/API_SALES_ORDER_SRV/A_SalesOrder?$filter=(SoldToParty eq 'AZ001' or
SoldToParty eq 'AZ002') and TotalNetAmount le 10000.00
```

Let's be honest, filtering data out is simple. Now, using the same logic, you can select only specific fields. This time, instead of the $filter query parameter, we will use the $select one. To get data only from SalesOrder, SoldToParty and TotalNetAmount fields, you can use the following query:

```
/API_SALES_ORDER_SRV/A_SalesOrder?$select=SalesOrder,SoldToParty,TotalNetAm
ount
```

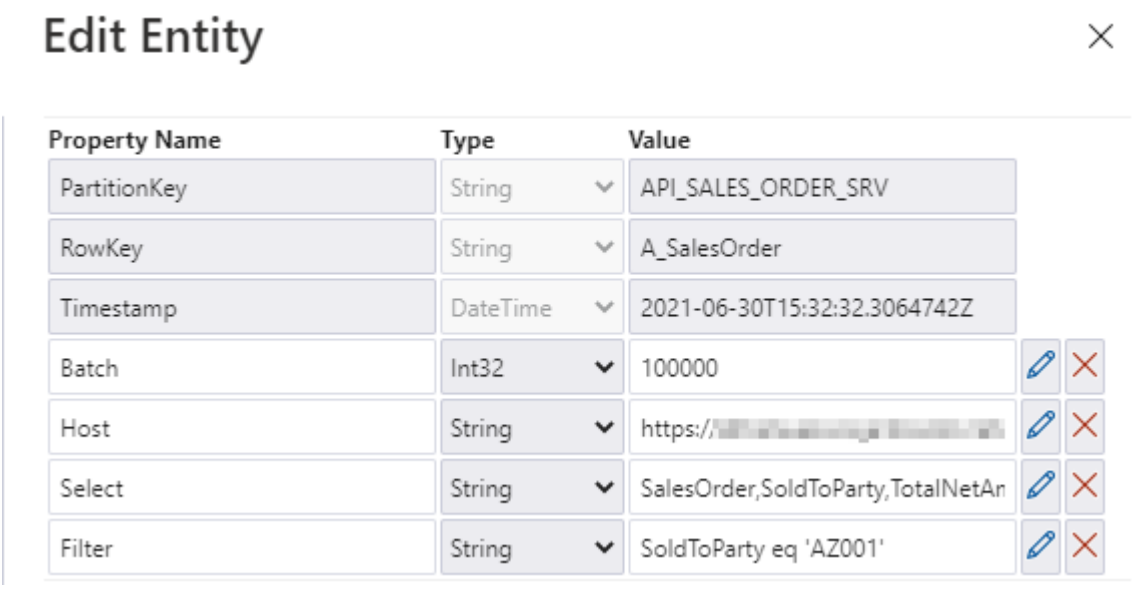
There is nothing stopping you from mixing $select and $filter parameters in a single query. Let's combine both above examples:

```
/API_SALES_ORDER_SRV/A_SalesOrder?$select=SalesOrder,SoldToParty,TotalNetAm
ount&$filter=(SoldToParty eq 'AZ001' or SoldToParty eq 'AZ002') and
TotalNetAmount le 10000.00
```

By applying the above logic, the OData response time reduced from 105 seconds to only 15 seconds, and its size decreased by 97 per cent. That, of course, has a direct impact on the overall performance of the extraction process.

## FILTERING AND SELECTION IN THE PIPELINE

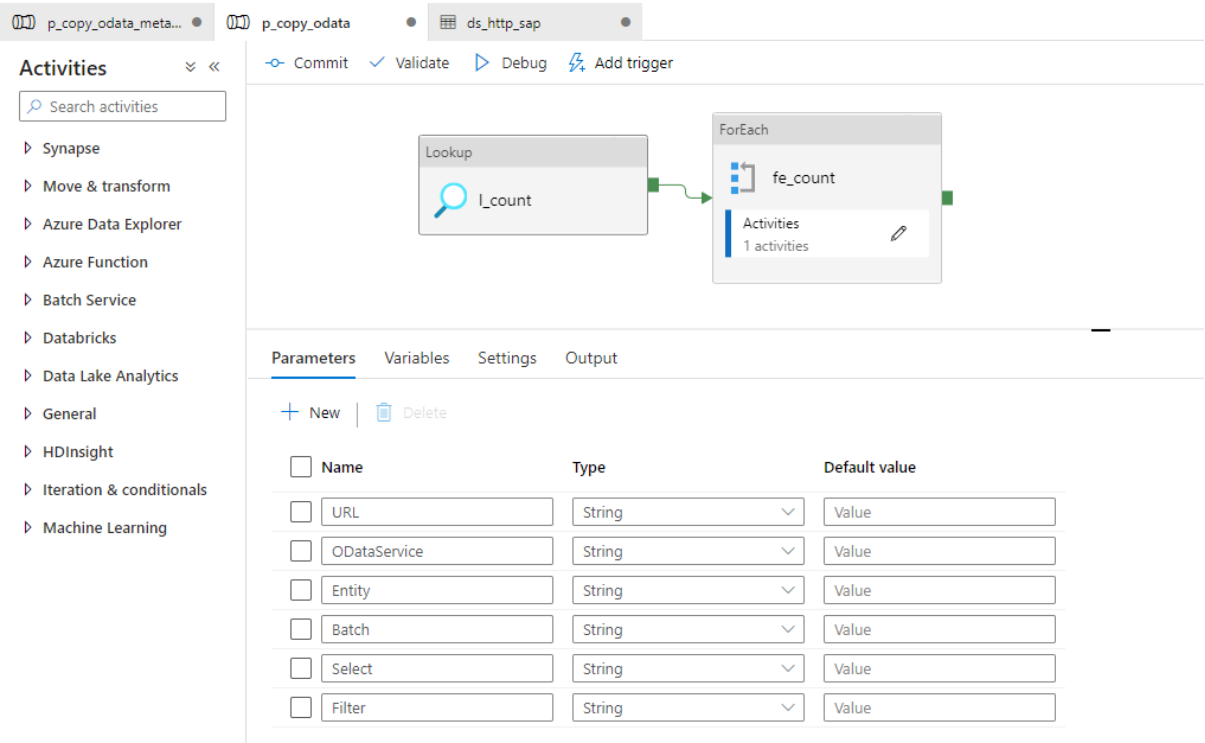
The filtering and selection options should be based on the entity level of the OData service. Each entity has a unique set of fields, and we may want to provide different filtering and selection rules. We will store the values for query parameters in the metadata store. Open it in the Storage Explorer and add two properties: filter and select.



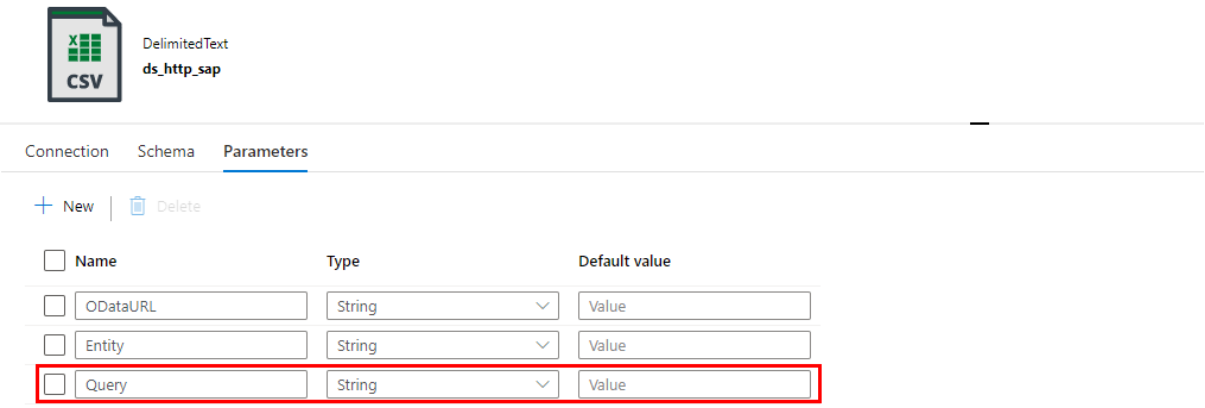

I'm pretty sure that based on the previous episodes of the blog series, you could already implement the logic in the pipeline without my help. But there are two challenges we should be mindful of. Firstly, we should not assume that $filter and $select parameters will always contain a value. If you want to extract the whole entity, you can leave those fields empty, and we should not pass them to the SAP system. In addition, as we are using the client-side caching to chunk the requests into smaller pieces, we need to ensure that we

pass the same filtering rules in the Lookup activity where we check the number of records in the OData service.

Let's start by defining parameters in the child pipeline to pass filter and select values from the metadata table. We've done that already in the third episode, so you know all steps.

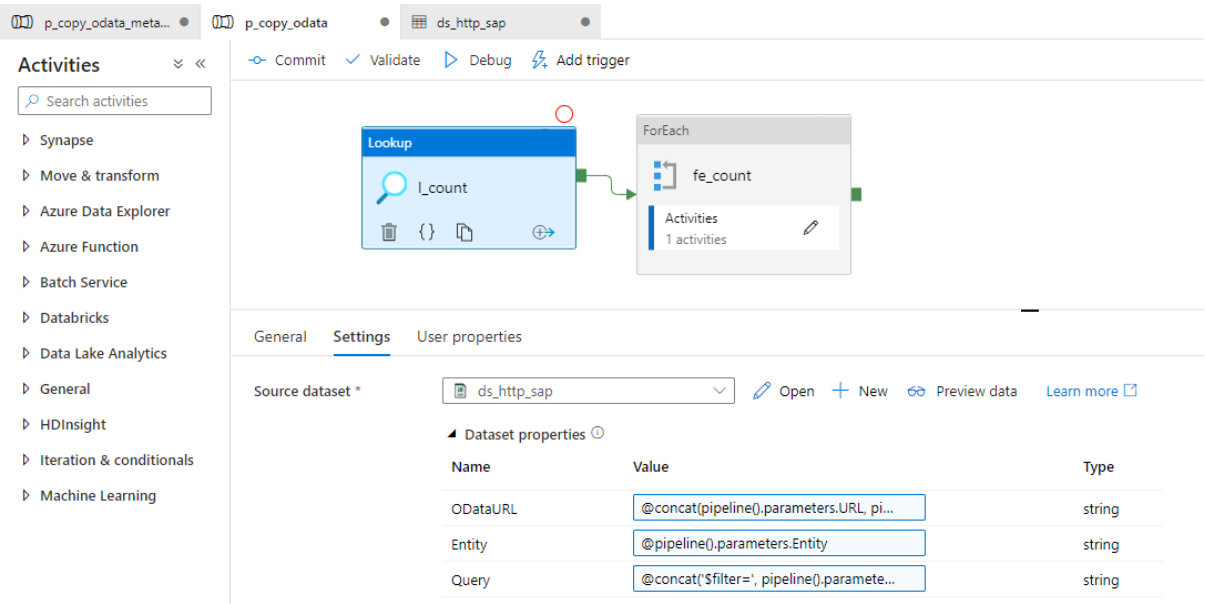



To correctly read the number of records, we have to consider how to combine these additional parameters with the OData URL in the Lookup activity. So far, the dataset accepts two dynamic fields: ODataURL and Entity. To pass the newly defined parameters, you have to add the Query one.



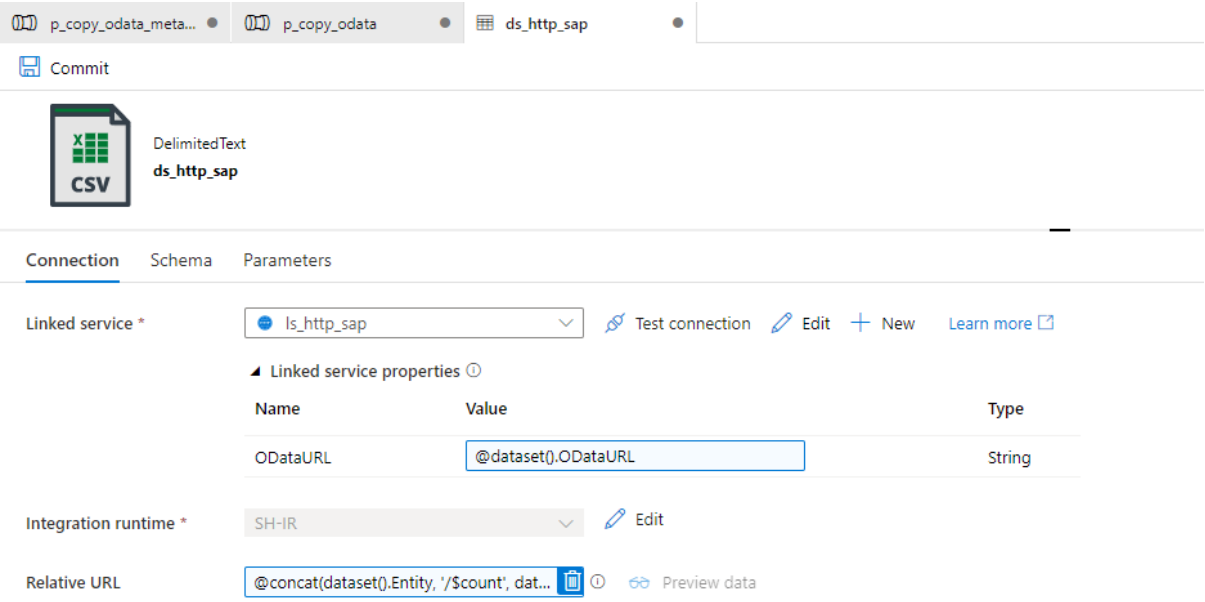

You can go back to the Lookup activity to define the expression to pass the $filter and $query values. It is very simple. I check if the Filter parameter in the metadata store contains any value. If not, then I'm passing an empty string. Otherwise, I concatenate the query parameter name with the value.

```
@if(empty(pipeline().parameters.Filter), '', concat('?$filter=',
pipeline().parameters.Filter))
```



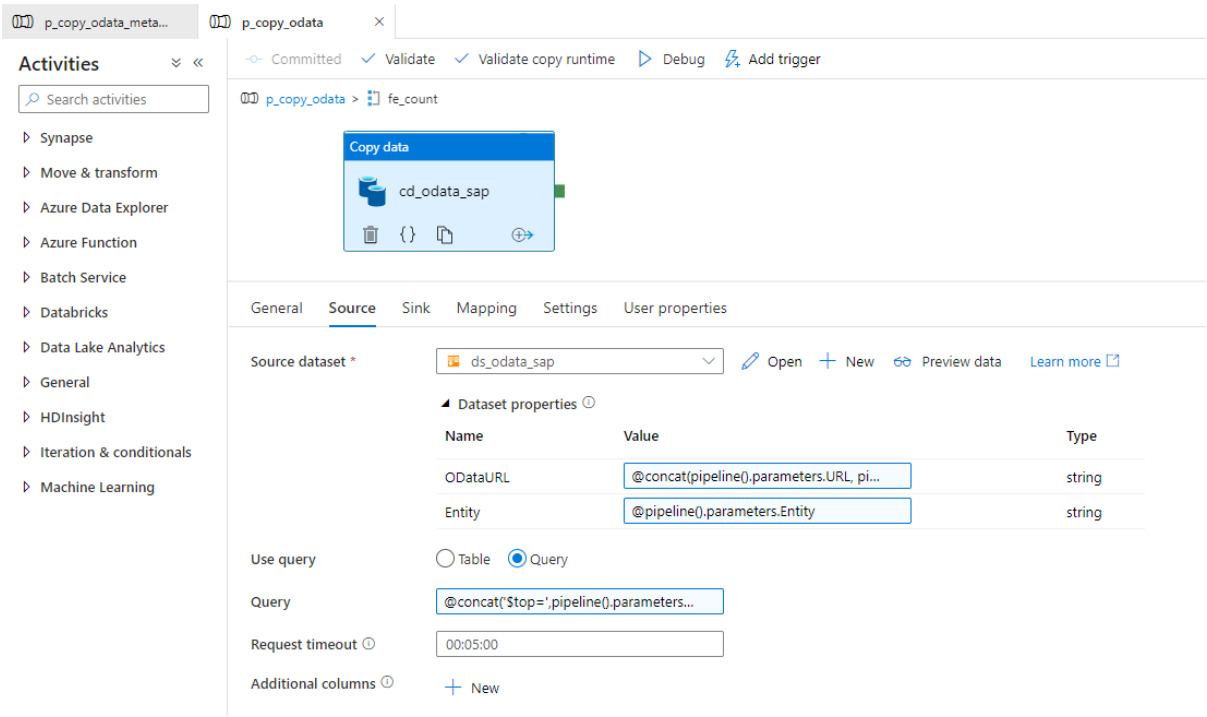

Finally, we can use the Query field in the Relative URL of the dataset. We already use that field to pass the entity name and the $count operator, so we have to slightly extend the expression.

```
@concat(dataset().Entity, '/$count', dataset().Query)
```




Changing the Copy Data activity is a bit more challenging. The Query field is already defined, but the expression we use should include the $top and $skip parameters, that we use for the client-side paging. Unlike at the Lookup activity, this time we also have to include both $select and $filter parameters and check if they contain a value. It makes the expression a bit lengthy.

```
@concat('$top=',pipeline().parameters.Batch,
'&$skip=',string(mul(int(item()), int(pipeline().parameters.Batch))),
if(empty(pipeline().parameters.Filter), '',
concat('&$filter=',pipeline().parameters.Filter)),
if(empty(pipeline().parameters.Select), '',
concat('&$select=',pipeline().parameters.Select)))
```



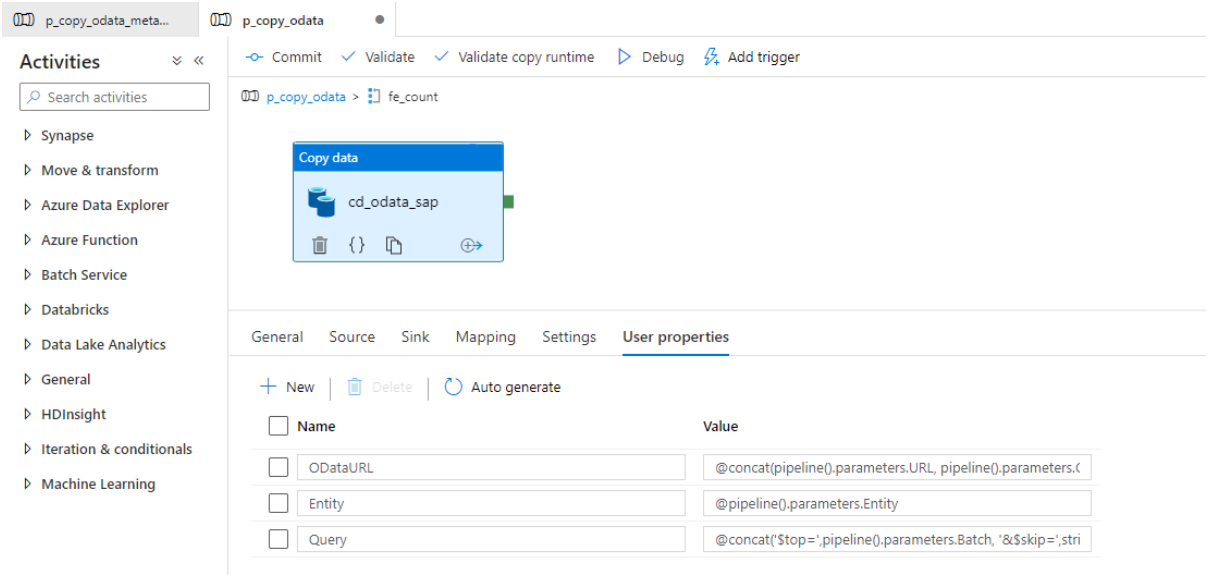

With the above changes, the pipeline uses filter and select values to extract only the data you need. It reduces the amount of processed data and improves the execution runtime.

## IMPROVING MONITORING

As we develop the pipeline, the number of parameters and expressions grows. Ensuring that we haven't made any mistakes becomes quite a difficult task. By default, the Monitoring view only gives us basic information on what the pipeline passes to the target system in the Copy Data activity. At the same time, parameters influence which data we extract. Wouldn't it be useful to get a more detailed view?

There is a way to do it! In Azure Synapse Pipelines, you can define User Properties, which are highly customizable fields that accept custom values and expressions. We will use them to verify that our pipeline works as expected.

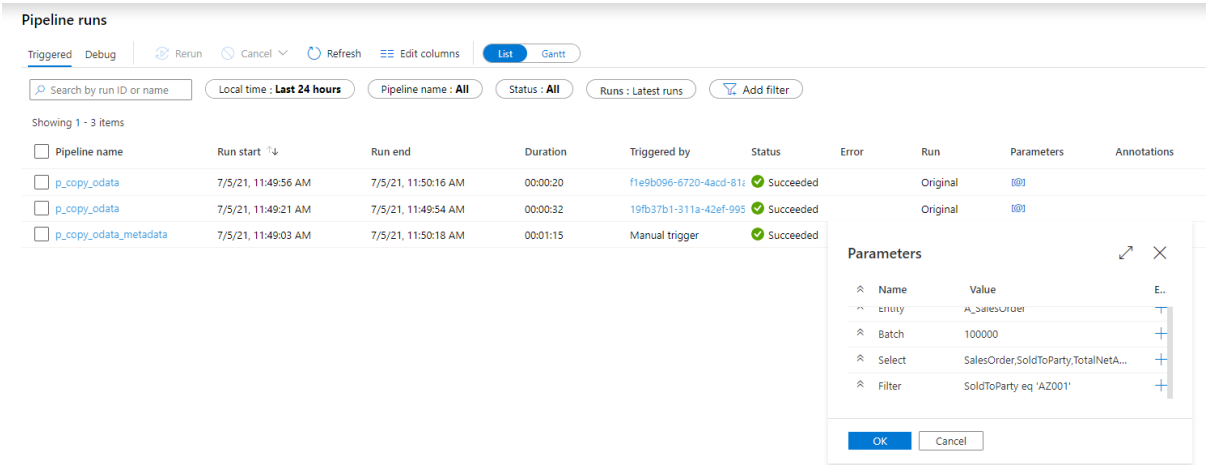
Open the Copy Data activity and select the User Properties tab. Add three properties we want to monitor – the OData URL, entity name, and the query passed to the SAP system. Copy expression from the Copy Data activity. It ensures the property will have the same value as is passed to the SAP system.




Once the user property is defined we start the extraction job.

## MONITORING AND EXECUTION

Let's start the extraction. I process two OData services, but I have defined Filter and Select parameters to only one of them.
Once the processing has finished, open the Monitoring area. To monitor all parameters that the metadata pipeline passes to child one, click on the [@] sign:

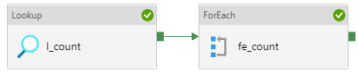



Now, enter the child pipeline to see the details of the Copy Data activity.

p_copy_odata

List Gantt

Rerun | Rerun from activity | Rerun from failed activity | Refresh | Edit pipeline




Activity runs

Pipeline run ID 5d142bfb-98de-41df-88e4-841e19c62237

All status

Showing 1 - 3 of 3 items

| Activity name | Activity type | Run start | Duration | Status | Error | Integration runti... | User properties | Run ID |
|---|---|---|---|---|---|---|---|---|
| cd_odata_sap | Copy data | 7/5/21, 11:49:38 AM | 00:00:14 | Succeeded | | SH-IR | | 590e8021-dbcf-4a70-aad3-083911dcd080 |
| fe_count | ForEach | 7/5/21, 11:49:37 AM | 00:00:17 | Succeeded | | | | e3c1a3a1-2c12-495a-b909-e31c0e070300 |
| l_count | Lookup | 7/5/21, 11:49:22 AM | 00:00:14 | Succeeded | | SH-IR | | 8a14c71f-6a97-4f2c-b6d4-cddbf93e924c |

When you click on the icon in the User Properties column, you can display the defined user properties. As we use the same expressions as in the Copy Activity, we clearly see what was passed to the SAP system. In case of any problems with the data filtering, this is the first place to start the investigation.

Parameters

| Name | Value |
|---|---|
| ODataURL | https://s4hana.azure.jarkowski.net/sap/opu/odata/sap/API_SALES_ORDER_SRV |
| Entity | A_SalesOrder |
| Query | $top=100000&$skip=0&$filter=SoldToParty eq 'AZ001'&$select=SalesOrder,SoldToParty,TotalNetAmount |

The above parameters are very useful when you need to troubleshoot the extraction process. Mainly, it shows you the full request query that is passed to the OData service – including the $top and $skip parameters that we defined in the previous episode.

The extraction was successful, so let's have a look at the extracted data in the lake.

p_copy_odata | p_copy_odata_meta... | odata | SQL script 1

Run | Undo | Commit | Query plan | Connect to Built-in | Use database master

```
SELECT
    TOP 100 *
FROM
    OPENROWSET(
        BULK 'https://[redacted].dfs.core.windows.net/odata/API_SALES_ORDER_SRV/A_SalesOrder/data_590e8021-dbcf-4a70-aad3-083911dcd080_74848556-20ef-402f-9da9-4829afb8632d.parquet',
        FORMAT='PARQUET'
    ) AS [result]
```

Results | Messages

View Table Chart | Export results

Search

| SalesOrder | SoldToParty | TotalNetAmount |
|---|---|---|
| 2 | AZ001 | 4995.000000000000000000 |
| 3 | AZ001 | 2997.000000000000000000 |
| 4 | AZ001 | 19980.000000000000000000 |
| 5 | AZ001 | 11988.000000000000000000 |
| 7 | AZ001 | 14251.000000000000000000 |
| 16 | AZ001 | 69071.000000000000000000 |
| 20 | AZ001 | 62163.000000000000000000 |
| 25 | AZ001 | 65506.000000000000000000 |

There are only three columns, which we have selected using the $select parameter. Similarly, we can only see rows with SoldToParty equals AZ001 and the TotalNetAmount above 1000. It proves the filtering works fine.

I hope you enjoyed this episode. I will publish the next one in the upcoming week, and it will show you one of the ways to implement delta extraction. A topic that many of you wait for!

# Part 6 – Introduction to delta extraction

After five weeks of going through basic data extraction scenarios, we finally can deep dive into a more advanced topic. I already explained how crucial is extraction optimization. The created Synapse Pipeline can use client-side paging to chunk a large request into several smaller ones. In the last episode, we focused on data filtering to copy only the required information. Both solutions improve the performance and reliability of the extraction, but a full data copy is not always an answer. It is not uncommon to have millions or even billions of records in the largest tables. A daily extraction job to copy everything is pretty much impossible in such scenarios. It would take too much time and cause performance challenges on SAP application servers. Multiple extractions every day could cause even more problems.

So what can we do to ensure our data in the lake is almost up-to-date? The solution, at least in theory, is quite simple. We should only extract new and changed records in the system. Such an approach highly limits the amount of data to process, which decreases the duration of the copy job and it allows more frequent updates.

## INTRODUCTION TO DELTA EXTRACTION

A post about processing only new and changed data, also known as a delta or incremental extraction, was the most requested by you. I received multiple comments and even private messages asking to provide some guidance in this area. OData services have some great features to support delta extraction. That's one of the reasons why, despite my concerns about using OData protocol for large amounts of data, I decided to write the whole blog series.

As always, please let me start with a short introduction. Extracting only changed and new information is not a new topic. Whenever we work with a large dataset, processing only a subset of information guarantees better performance and reliability. Having to copy only a couple hundred records instead of millions always makes a difference. The main problem is, however, how to select the right data.

As always, there is no single solution. Which is the right one depends on your requirements and technical possibilities. One of the most common approaches uses the date and time of last change to identify new and changed records since the last extraction. Another way of ensuring we have updated data is using triggers to record and send changes after they occur. The second approach is quite common in real-time data replication scenarios.

An important point of consideration is which system defines which are new and changed information. Using the client software, which in our case is Azure Synapse, gives us full control over what we request and copy. At the same time, we are limited to what is available at the source. If the OData service doesn't expose a field with the timestamp of the last update, it will be nearly impossible to track changes. And unfortunately, that's a common problem. Always work closely with your functional experts as they may know a solution. Depending on the use case, you may be able to use other fields as well – for example, posting or document date. You could also try finding another data source as it may

contain a date to support delta extraction. To be honest, it doesn't even have to be a date – using a posting period can also give good results. Of course, the number of records to process each time will be larger, but still, it is just a tiny piece of all data stored in the database. One of the common misconceptions is that you always have to fetch **only** new and updated records. That's not true – you can always copy more records (for example, extract documents from the current month) and then use processing in Synapse to consolidate multiple extracts. While not perfect, such a solution often helps when the information about the last changes is unavailable.

The SAP system can also help us in identifying delta changes. It offers a set of data extractors and CDS views that out-of-the-box manages delta information, and they expose us only the data we need. That's quite handy, as we can use pre-built processes instead of designing a pipeline to compare timestamps. Under the hood, they also often use the information about the last change. But in newer SAP releases, they also utilize the SLT framework to track changes.

While asking the SAP system to manage delta information is tempting, I would like to show you both scenarios. Therefore I will cover the delta extraction in two blog posts. In today's episode, I'll show you how to extract changes using timestamps stored in the metadata store. You already know how to implement filtering, so we will use that knowledge today. The solution we'll design today is just one of many. As I mentioned earlier, there are many field types you could use. It may happen that the pipeline we create today requires some further tweaks to fulfil your use case. But I'm sure you'll have enough knowledge to make the necessary changes on your own.

## DELTA EXTRACTION USING TIMESTAMP

With OData services, we can use a wide range of query parameters to limit the data we receive. Last week I published a blog about filtering, that allows us to only extract information for a particular company code or sales organization. We can use the same approach to limit the data based on a timestamp field – ideally containing information about last changes, but creation date or posting date can also be useful.

Let's take the API_SALES_ORDER_SRV as an example, as we worked with that service previously. The entity set A_SalesOrder exposes us Sales Order header information. It contains CreationDate field which holds the timestamp of when was the document initially created as well as the LastChangeDateTime which provides information about last changes. While that's not always the case for other OData services, the LastChangeDateTime field is updated even if the document was just created and never edited.

```
    ('7')/to_PricingElement"/>
  <link title="to_Text" type="application/atom+xml;type=feed"
      rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/to_Text" href="A_SalesOrder('7')/to_Text"/>
- <content type="application/xml">
  - <m:properties xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
    xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">
      <d:SalesOrder>7</d:SalesOrder>
      <d:SalesOrderType>OR</d:SalesOrderType>
      <d:SalesOrganization>1110</d:SalesOrganization>
      <d:DistributionChannel>10</d:DistributionChannel>
      <d:OrganizationDivision>00</d:OrganizationDivision>
      <d:SalesGroup>110</d:SalesGroup>
      <d:SalesOffice>110</d:SalesOffice>
      <d:SalesDistrict/>
      <d:SoldToParty>AZ001</d:SoldToParty>
      <d:CreationDate>2020-03-11T00:00:00</d:CreationDate>
      <d:CreatedByUser>BJARKOWSKI</d:CreatedByUser>
      <d:LastChangeDate m:null="true"/>
      <d:LastChangeDateTime>2020-03-11T13:50:53.7244400Z</d:LastChangeDateTime>
      <d:PurchaseOrderByCustomer>AZ001-05</d:PurchaseOrderByCustomer>
      <d:CustomerPurchaseOrderType/>
      <d:CustomerPurchaseOrderDate m:null="true"/>
      <d:SalesOrderDate>2020-03-11T00:00:00</d:SalesOrderDate>
      <d:TotalNetAmount>14251.00</d:TotalNetAmount>
      <d:TransactionCurrency>GBP</d:TransactionCurrency>
      <d:SDDocumentReason/>
      <d:PricingDate>2020-03-11T00:00:00</d:PricingDate>
      <d:RequestedDeliveryDate>2020-04-11T00:00:00</d:RequestedDeliveryDate>
      <d:ShippingCondition>01</d:ShippingCondition>
      <d:CompleteDeliveryIsDefined>false</d:CompleteDeliveryIsDefined>
      <d:ShippingType/>
      <d:HeaderBillingBlockReason/>
      <d:DeliveryBlockReason/>
      <d:IncotermsClassification>EXW</d:IncotermsClassification>
      <d:IncotermsTransferLocation>London</d:IncotermsTransferLocation>
      <d:IncotermsLocation1>London</d:IncotermsLocation1>
      <d:IncotermsLocation2/>
      <d:IncotermsVersion/>
      <d:CustomerPaymentTerms>0001</d:CustomerPaymentTerms>
      <d:PaymentMethod/>
      <d:AssignmentReference/>
      <d:ReferenceSDDocument/>
      <d:ReferenceSDDocumentCategory/>
      <d:CustomerTaxClassification1/>
      <d:TaxDepartureCountry/>
```


In today's post, I'll show you what changes are required to deal with similar OData services. But there are other OData services, like the API_BUSINESS_PARTNER, that require further adjustment. Let's take a look at a sample payload:


```
- <content type="application/xml">
  - <m:properties xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
    xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">
      <d:BusinessPartner>AZ001</d:BusinessPartner>
      <d:Customer>AZ001</d:Customer>
      <d:Supplier/>
      <d:AcademicTitle/>
      <d:AuthorizationGroup/>
      <d:BusinessPartnerCategory>2</d:BusinessPartnerCategory>
      <d:BusinessPartnerFullName>IT Hardware</d:BusinessPartnerFullName>
      <d:BusinessPartnerGrouping>BPAB</d:BusinessPartnerGrouping>
      <d:BusinessPartnerName>IT Hardware</d:BusinessPartnerName>
      <d:BusinessPartnerUUID>000d3a4b-3ab9-1eea-98dc-cbee730798fd</d:BusinessPartnerUUID>
      <d:CorrespondenceLanguage/>
      <d:CreatedByUser>BJARKOWSKI</d:CreatedByUser>
      <d:CreationDate>2020-03-10T00:00:00</d:CreationDate>
      <d:CreationTime>PT15H48M15S</d:CreationTime>
      <d:FirstName/>
      <d:FormOfAddress>0003</d:FormOfAddress>
      <d:Industry/>
      <d:InternationalLocationNumber1>0</d:InternationalLocationNumber1>
      <d:InternationalLocationNumber2>0</d:InternationalLocationNumber2>
      <d:IsFemale>false</d:IsFemale>
      <d:IsMale>false</d:IsMale>
      <d:IsNaturalPerson/>
      <d:IsSexUnknown>false</d:IsSexUnknown>
      <d:Language/>
      <d:LastChangeDate>2020-03-10T00:00:00</d:LastChangeDate>
      <d:LastChangeTime>PT19H54M51S</d:LastChangeTime>
      <d:LastChangedByUser>BJARKOWSKI</d:LastChangedByUser>
      <d:LastName/>
      <d:LegalForm/>
      <d:OrganizationBPName1>IT Hardware</d:OrganizationBPName1>
      <d:OrganizationBPName2/>
      <d:OrganizationBPName3/>
      <d:OrganizationBPName4/>
      <d:OrganizationFoundationDate m:null="true"/>
      <d:OrganizationLiquidationDate m:null="true"/>
      <d:SearchTerm1/>
      <d:AdditionalLastName/>
      <d:BirthDate m:null="true"/>
      <d:BusinessPartnerIsBlocked>false</d:BusinessPartnerIsBlocked>
      <d:BusinessPartnerType/>
      <d:ETag>BJARKOWSKI20200310195451</d:ETag>
      <d:GroupBusinessPartnerName1/>
```

This OData service makes things difficult. There is no single field for the timestamp, and the last update date and time are stored in two separate fields. That makes things slightly more complex – instead of filtering using a single value, we need to use two of them. If you'd like to use the pipeline to work with such OData services, you'll have to make additional changes on your own.
In theory, there is also a field ETag that looks useful. But in fact, this field is used to manage editing concurrency, and some SAP cloud products already prohibit filtering using the ETag value.

There is also the third type of OData service, which doesn't expose any form of date or time information. An infamous example is the A_SalesOrderItem in the API_SALES_ORDER. If you have to process sales order line items, it's better to find another data source, like an extractor or CDS views – next week, I will describe how to use them.

But to be fully honest with you – there are two possible workarounds. After extracting header information, you could create a list of processed sales order numbers, which then becomes a filter when selecting line items. The second potential workaround, which is not available in Synapse so far, is using the $expand query parameter. It asks SAP to include additional information in the response, like the partner information or… line items. I'll keep an eye on new features in Synapse Pipelines, and I let you know when this parameter is fully supported.

Before we make changes to the pipeline, let's think about how the delta extraction should work. We need three additional parameters – the first one to mark we want to use delta extraction with the OData source. In the second and third parameters, we will keep the timestamp information together with the field name where it can be found in the OData source.

During the runtime, the pipeline should initially send a request to the service to retrieve the high watermark, which is the timestamp of the last change. We can do this by using Lookup activity and a couple of query parameters. The Copy Data activity will then only process orders, with the timestamp being between the value stored in the metadata store and read from the OData service. Once the extraction finishes, we have to update the metadata store with the new high watermark.

Initially, I considered implementing the above changes to the same child pipeline we've used so far, but finally, I decided to create a separate one to increase visibility. All OData services where the delta extraction is not required will still use the old pipeline. I will add a Switch activity in the master pipeline, that based on the value read from the metadata store, will trigger the original or delta-enabled pipeline.

## IMPLEMENTATION

Let's start making the required changes by modifying the metadata store. We need three additional fields:

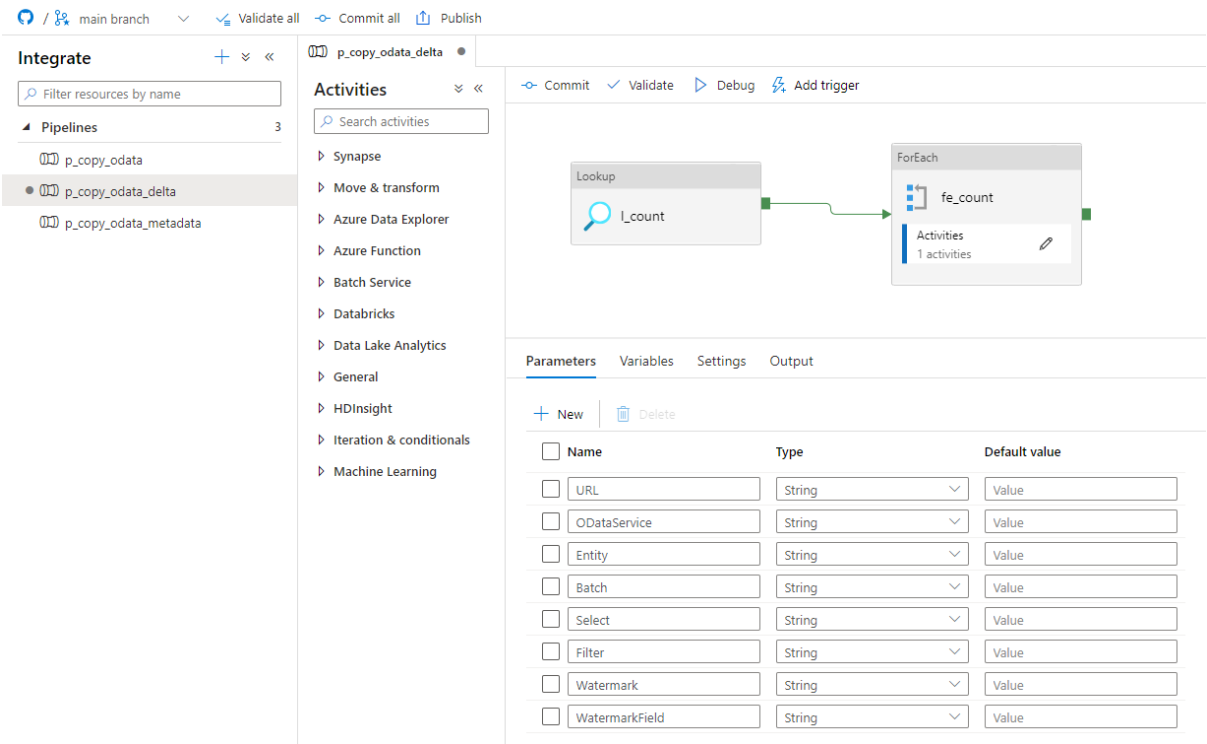
- ExtractionType – which can take Full or Delta values

- Watermark – to store the timestamp value
- WatermarkField – to store the field name of the timestamp, for example, LastChangeDateTime

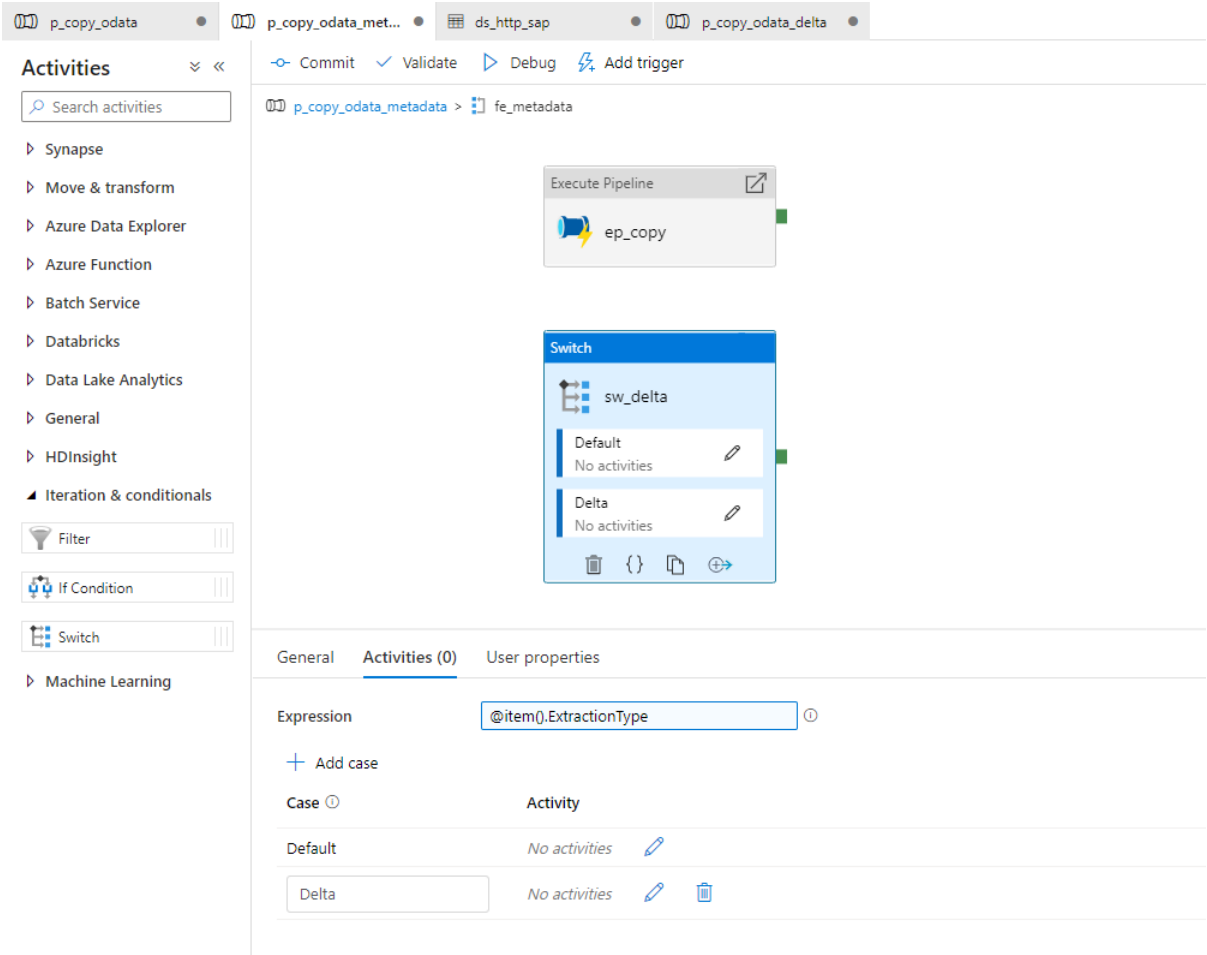
I have also updated my list of OData services to include two that we can consider for delta extraction (sales orders and products) and two without a timestamp field we could use.

| OData service | Entity | ExtractionType | WatermarkField |
|---|---|---|---|
| API_SALES_ORDER_SRV | A_SalesOrder | Delta | LastChangeDateTime |
| API_BUSINESS_PARTNER | A_BusinessPartner | Full | |
| API_COMPANYCODE_SRV | A_CompanyCode | Full | |
| API_PRODUCT_SRV | A_Product | Delta | LastChangeDateTime |

Having new fields defined in the metadata store allows us to focus now on the pipeline. Open Synapse Studio and clone the existing child pipeline. Then define two parameters: Watermark and WatermarkField. As the ExtractionType property will only be used in the metadata pipeline, we don't have to define it here.

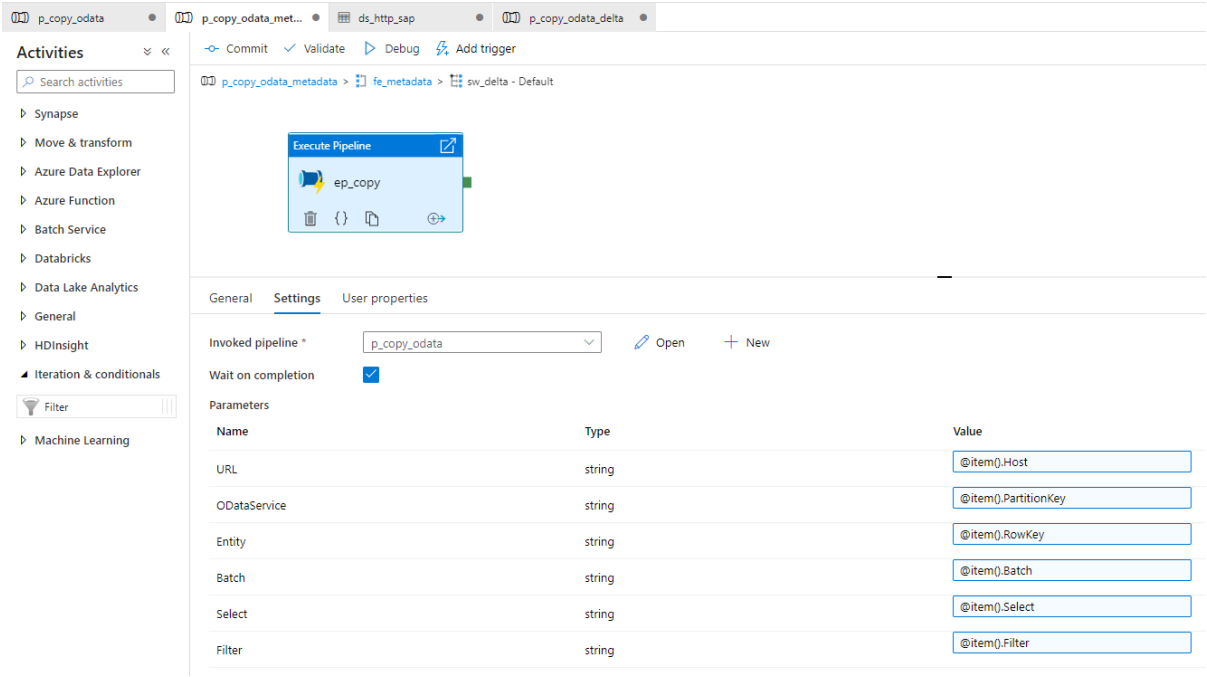



Now we have to modify the metadata pipeline, that it triggers the correct processing depending on the content of the ExtractionType field. Open it, and enter the ForEach loop. Add the Switch activity and reference the ExtractionType in the Expression field to evaluate which pipeline to run. Then add a new Case with the value "Delta".

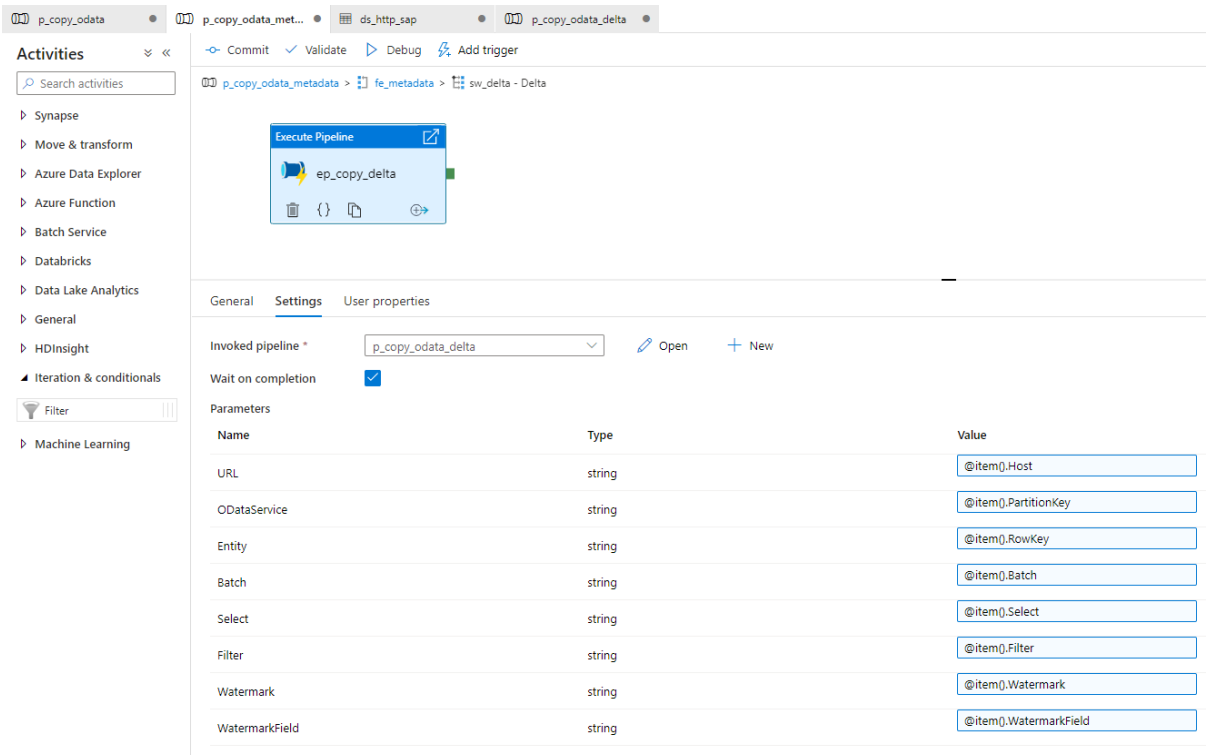



The Switch activity executes the expression and maps the outcome to one of Case groups. If it can't find a match it will process the Default group.

Move the existing Execute Pipeline activity to the default group. You don't have to make any changes to the list of parameters.




Expand the delta group and add a new Execture Pipeline activity that triggers the delta-enabled pipeline. Set the Invoked pipeline and expand the existing list of parameters with Watermark and WatermarkField.




With the above changes, the Switch activity triggers the correct pipeline based on the value stored in the ExtractionType field. We can now proceed with the required improvements to the delta-enabled pipeline.

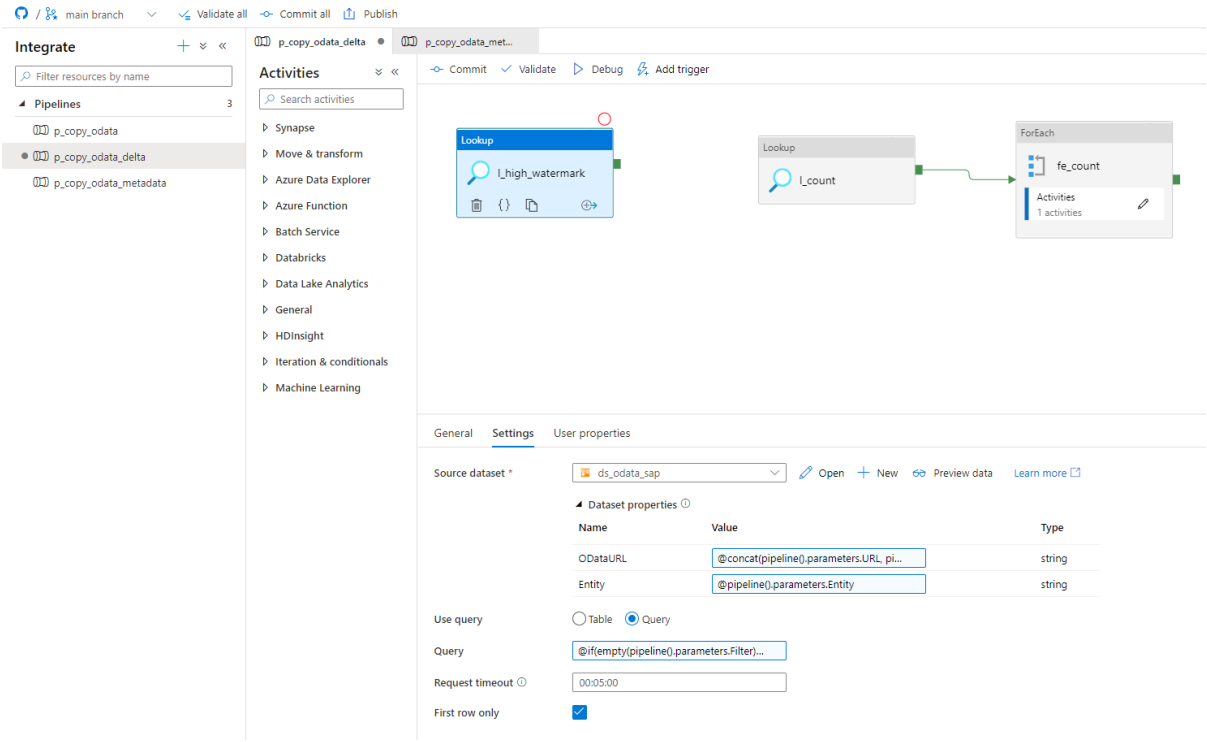
I want to keep the existing functionality that allows using the client-side paging as well as filtering out unnecessary data. While delta extraction reduces the amount of data in subsequent runs, we still have to ensure robust processing of the initial data extraction that copies all records. We have to update all expressions, and keeping the existing functionality means some of them can get quite lengthy.

Let's start with an easy change. Before we check the number of records, we have to read the high watermark figure from the OData service to correctly pass the filtering to all subsequent activities. Add a new Lookup activity and reference the OData dataset. Provide the following expressions:

```
ODataURL: @concat(pipeline().parameters.URL,
pipeline().parameters.ODataService, '/')
Entity: @pipeline().parameters.Entity
```

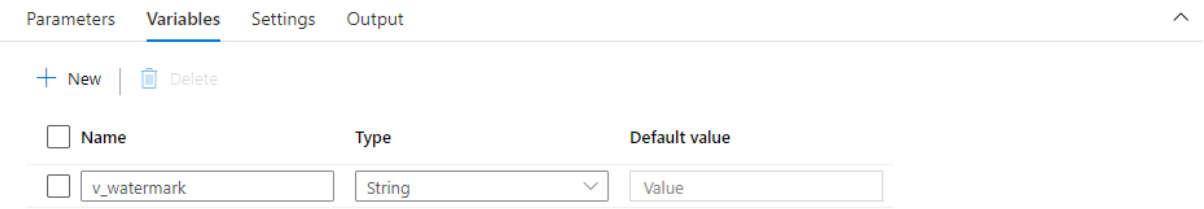
In the query field, we provide the selection criteria. To read the latest timestamp, I'm sorting the whole dataset using the $orderby parameter. We also have to maintain all filtering rules defined in the metadata store. I came up with the following query:

```
@if(empty(pipeline().parameters.Filter), concat('?$select=',
pipeline().parameters.WatermarkField, '&$top=1&$orderby=',
pipeline().parameters.WatermarkField, ' desc'), concat('?$filter=',
pipeline().parameters.Filter, '&$select=',
pipeline().parameters.WatermarkField, '&$top=1&$orderby=',
pipeline().parameters.WatermarkField, ' desc'))
```




To make the timestamp value easily accessible, I decided to assign it to a variable. We haven't used any variables so far. They are very similar to parameters, but unlike them, we can modify variables values during the runtime. Which perfectly suits our case.

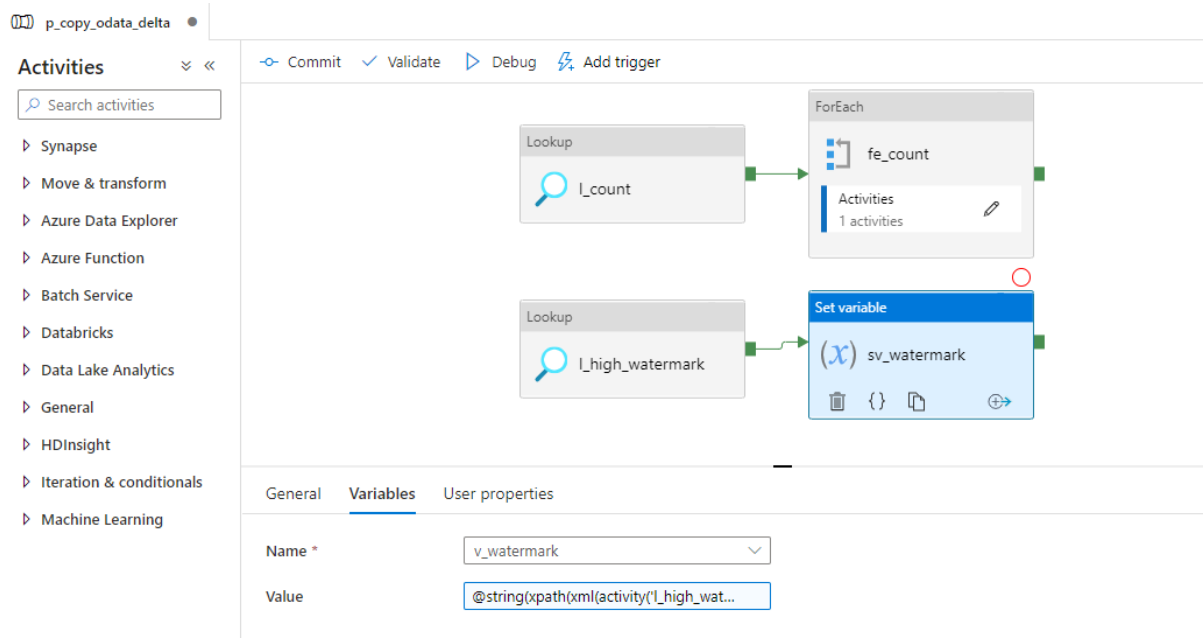
In the pipeline settings, open the Variables tab and create a new one of type String.




Add the Set Variable activity and create a connection from the lookup. To read the value from the output of the previous action I'm using the following query:

```
@activity('l_high_watermark').output.firstRow[pipeline().parameters.Waterma
rkField]
```

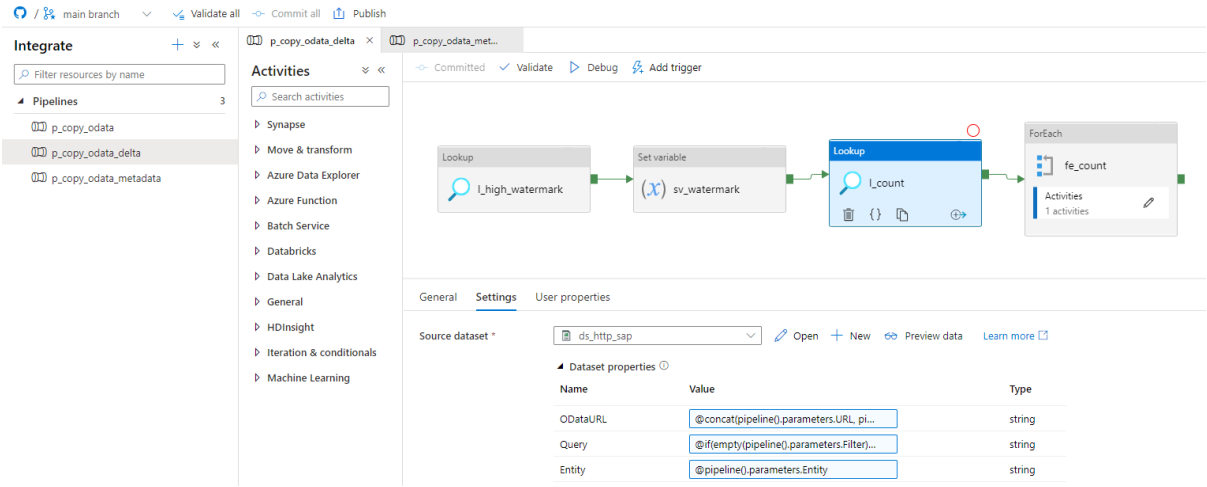
As the timestamp field name is not static, but instead we read it from the metadata store, I'm using square brackets to reference it as part of the expressions (that's quite a cool feature of ADF).




Having the high watermark value assigned to a variable, we can modify the Lookup responsible for reading the record count. We need to add functionality to count only records between the last replication and the high watermark value stored in the variable. We also have to keep in mind, that during the initial replication, the Watermark field in the metadata store is empty – which adds a little bit of complexity. I use the following expression:

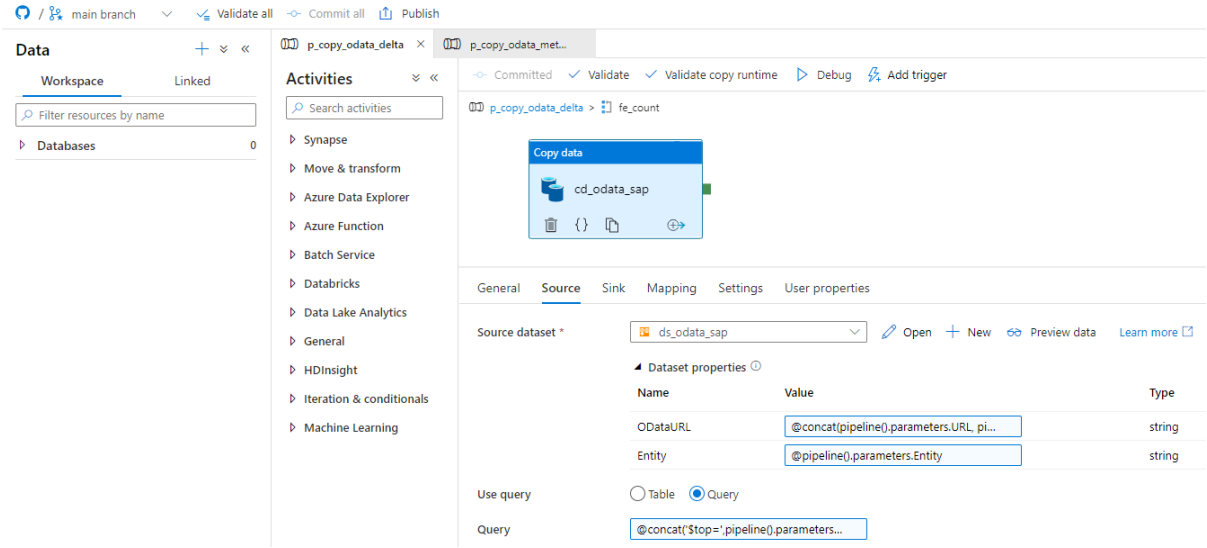
```
@if(empty(pipeline().parameters.Filter), concat('?$filter=',
if(empty(pipeline().parameters.Watermark), '',
concat(pipeline().parameters.WatermarkField, ' gt datetimeoffset''',
formatDateTime(pipeline().parameters.Watermark, 'yyyy-MM-ddThh:ss:sZ'), '''
and ')), pipeline().parameters.WatermarkField, ' le datetimeoffset''',
formatDateTime(variables('v_watermark'), 'yyyy-MM-ddThh:ss:sZ'),''''),
```

```
concat('?$filter=', pipeline().parameters.Filter,
if(empty(pipeline().parameters.Watermark), '', concat(' and ',
pipeline().parameters.WatermarkField, ' gt datetimeoffset''',
formatDateTime(pipeline().parameters.Watermark, 'yyyy-MM-ddThh:ss:sZ'))),
''' and ', pipeline().parameters.WatermarkField, ' le datetimeoffset''',
formatDateTime(variables('v_watermark'), 'yyyy-MM-ddThh:ss:sZ'),''''))
```

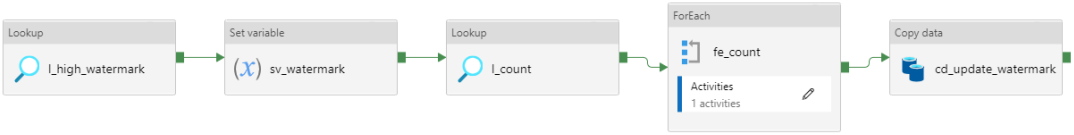



Now it's time to edit the query in the Copy Activity. I had to admit I was a bit scared of this one, as there are quite a few cases. We may or may not have filters or selection defined. It may be the first extraction, where we only have to pass the high watermark, or it may be a subsequent one that requires both timestamps. Finally, after a while, I wrote the following expression that I think addresses all cases.

```
@concat('$top=',pipeline().parameters.Batch,
'&$skip=',string(mul(int(item()), int(pipeline().parameters.Batch))),
if(empty(pipeline().parameters.Filter), concat('&$filter=',
if(empty(pipeline().parameters.Watermark), '',
concat(pipeline().parameters.WatermarkField, ' gt datetimeoffset''',
formatDateTime(pipeline().parameters.Watermark, 'yyyy-MM-
ddThh:ss:sZ'),'''', ' and ')), pipeline().parameters.WatermarkField, ' le
datetimeoffset''', formatDateTime(variables('v_watermark'), 'yyyy-MM-
ddThh:ss:sZ'),''''), concat('&$filter=', pipeline().parameters.Filter,
if(empty(pipeline().parameters.Watermark), '', concat(' and ',
pipeline().parameters.WatermarkField, ' gt datetimeoffset''',
formatDateTime(pipeline().parameters.Watermark, 'yyyy-MM-
ddThh:ss:sZ'),''''')), ' and ', pipeline().parameters.WatermarkField, ' le
datetimeoffset''', formatDateTime(variables('v_watermark'), 'yyyy-MM-
ddThh:ss:sZ'),''''')), if(empty(pipeline().parameters.Select), '',
concat('&$select=',pipeline().parameters.Select)))
```




Great! We're pretty much done!
We just have to update the metadata store with the high watermark value read from the OData service. I will use the Copy Data activity to do it.

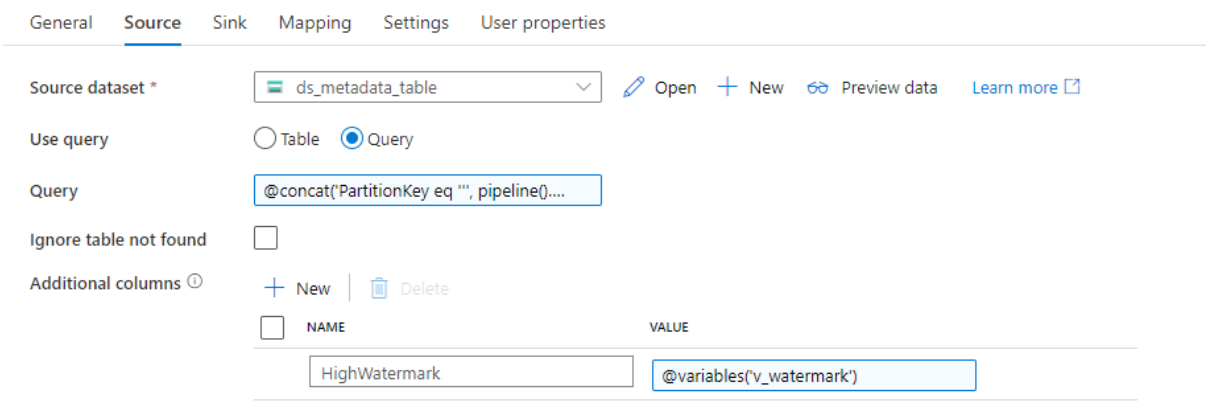



The source and sink stores should refer to the same metadata table dataset, which we use to read content from the store. I will use a query to select the entry to edit based on the OData service name and the Entity.

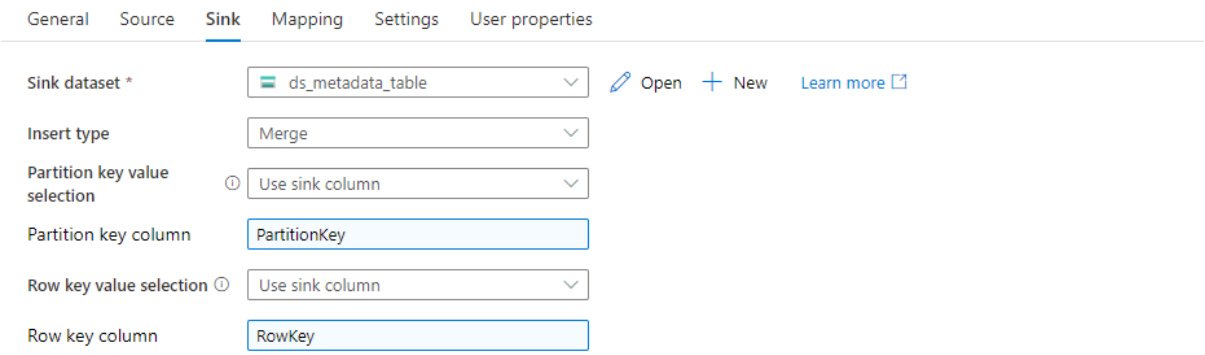
```
@concat('PartitionKey eq ''', pipeline().parameters.ODataService, ''' and
RowKey eq ''', pipeline().parameters.Entity, '''')
```

Then I define an additional column that stores the high watermark value, which we will map to the target column.

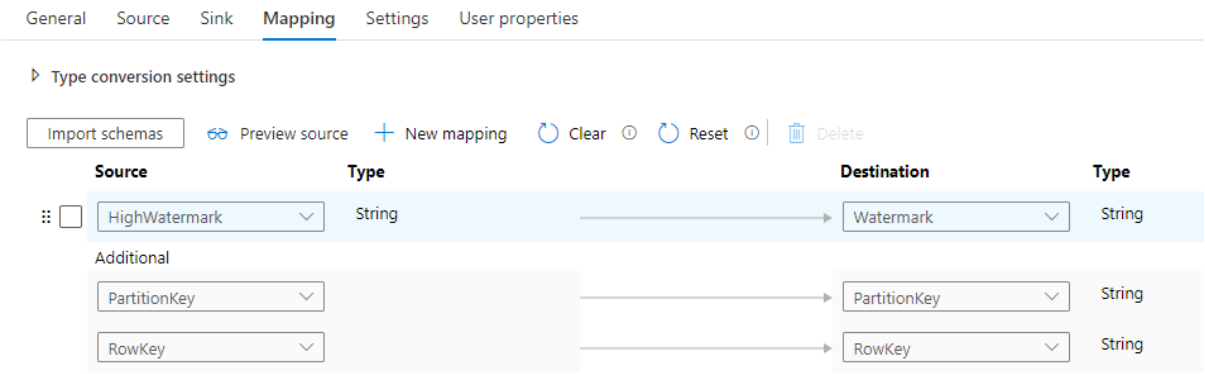
```
@variables('v_watermark')
```



Set the Insert Type to Merge on the Sink tab. That's important – if you leave it to Replace, the whole entry in the table will be recreated, which means we'll lose part of the data. Change the fields Partition Key Value Selection and Row Key Value Selection to PartitionKey and RowKey respectively.




Open the Mapping tab. The goal here is to reference the PartitionKey and RowKey fields to define the record that requires updating and then use the Additional Column HighWatermark to update the destination field Watermark.




That's it! We can start testing!

## EXECUTION AND MONITORING

To verify the pipeline works fine, I will run two tests. In the first one, as the watermark value for all OData services is empty, I'm expecting to see all data landing in the lake. Then I'll edit a couple of sales orders that should be extracted in the second run. I won't make any changes to materials.
3.. 2.. 1.. Go!

| Pipeline name | Run start | Run end | Duration | Triggered by | Status |
|---|---|---|---|---|---|
| p_copy_odata | 8/20/21, 11:59:09 AM | 8/20/21, 11:59:29 AM | 00:00:20 | 071a1593-12c6-45c9-86 | Succeeded |
| p_copy_odata_delta | 8/20/21, 11:58:25 AM | 8/20/21, 11:59:06 AM | 00:00:41 | 2791334c-e52d-4faf-a3b | Succeeded |
| p_copy_odata_delta | 8/20/21, 11:57:37 AM | 8/20/21, 11:58:22 AM | 00:00:45 | 33fde386-5b55-4ffa-a58 | Succeeded |
| p_copy_odata | 8/20/21, 11:57:15 AM | 8/20/21, 11:57:36 AM | 00:00:21 | aa3329dc-f067-4a8b-80 | Succeeded |
| p_copy_odata_metadata | 8/20/21, 11:57:09 AM | 8/20/21, 11:59:33 AM | 00:02:24 | Manual trigger | Succeeded |

Green status is always good. Let's check the number of records extracted.

| | OData service | Entity | Extraction type | Number of extracted records |
|---|---|---|---|---|
| **First run** | API_BUSINESS_PARTNER | A_BusinessPartner | Full | 1 |
| | API_PRODUCT_SRV | A_Product | Delta | 271 |
| | API_SALES_ORDER_SRV | A_SalesOrder | Delta | 105 |
| | API_SALES_ORDER_SRV | A_SalesOrderItem | Full | 379 |

So that looks good. I use tight filtering on the Business Partner which explains why only a single record was processed. A quick look at the metadata table and I can see that the watermark field was updated correctly:

| PARTITIONKEY | ROWKEY | WATERMARK |
|---|---|---|
| API_BUSINESS_PARTNER | A_BusinessPartner | *null* |
| API_PRODUCT_SRV | A_Product | 2021-04-14T13:27:33+00:00 |
| API_SALES_ORDER_SRV | A_SalesOrder | 2021-08-20T09:07:43+00:00 |
| API_SALES_ORDER_SRV | A_SalesOrderItem | |

Now I'm going to edit a couple of sales orders and then I'll trigger the pipeline. Let's see how many records we extract.

| | OData service | Entity | Extraction type | Number of extracted records |
|---|---|---|---|---|
| **2nd run** | API_BUSINESS_PARTNER | A_BusinessPartner | Full | 1 |
| | API_PRODUCT_SRV | A_Product | Delta | 0 |
| | API_SALES_ORDER_SRV | A_SalesOrder | Delta | 2 |
| | API_SALES_ORDER_SRV | A_SalesOrderItem | Full | 379 |

It worked well. Let's look at the metadata store table:

| PARTITIONKEY | ROWKEY | WATERMARK |
|---|---|---|
| API_BUSINESS_PARTNER | A_BusinessPartner | *null* |
| API_PRODUCT_SRV | A_Product | 2021-04-14T13:27:33+00:00 |
| API_SALES_ORDER_SRV | A_SalesOrder | 2021-08-20T12:21:51+00:00 |
| API_SALES_ORDER_SRV | A_SalesOrderItem | |

The high watermark for the Entity A_SalesOrder was updated with the last change timestamp. So everything works correctly.

As always I hope you enjoyed this episode! I showed you how you can implement the delta extraction for OData services using Synapse Pipelines and metadata store to identify records that require processing. It's not a perfect solution. We need to have the timestamp information and heavy scripting is required. But it works. In the next episode, I'll show you

how to convince your SAP system to manage the delta for you – which is much better approach!

# Part 7 - Delta extraction using SAP Extractors and CDS Views

Seven weeks passed in a blink of an eye, and we are at the end of the Summer with OData-based extraction using Synapse Pipeline. Each week I published a new episode that reveals best practices on copying SAP data to the lake, making it available for further processing and analytics. Today's episode is a special one. Not only it is the last one from the series, but I'm going to show you some cool features around data extraction that pushed me into writing the whole series. Since I have started working on the series, it was the main topic I wanted to describe. Initially, I planned to cover it as part of my Your SAP on Azure series that I'm running for the last couple of years. But as there are many intriguing concepts in OData-based extraction, and I wanted to show you as much as I can, I decided to run a separate set of posts. I hope you enjoyed it and learnt something new.

Last week I described how you could design a pipeline to extract only new and changed data using timestamps available in many OData services. By using filters, we can only select a subset of information which makes the processing much faster. But the solution I've shared works fine for just a part of services, where the timestamp is available as a single field. For others, you have to enhance the pipeline and make the complex expressions even more complicated.

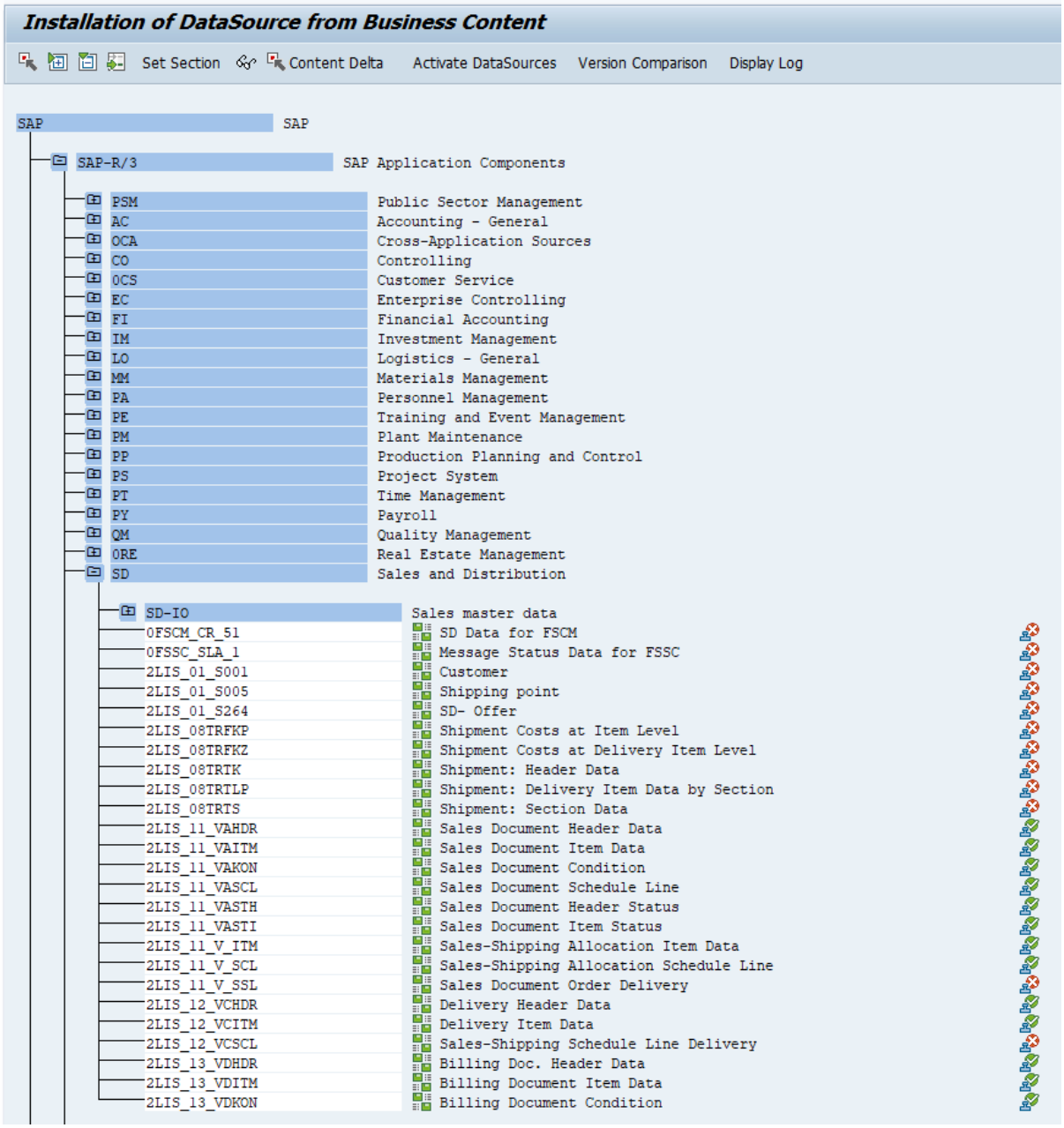
There is a much better approach. Instead of storing the watermark in the data store and then using it as filter criteria, you can convince the SAP system to manage the delta changes for you. This way, without writing any expression to compare timestamps, you can extract recently updated information.

The concept isn't new. SAP Extractors are available since I remember and are commonly used in SAP Business Warehouse. Nowadays, in recent SAP system releases, there are even analytical CDS views that support data extraction scenarios, including delta management! And the most important information is that you can expose both SAP extractors and CDS Views as OData services making them ideal data sources.

## EXPOSE EXTRACTORS AND CDS VIEWS AS ODATA

The process of exposing extractors and CDS views as OData is pretty straightforward. I think a bigger challenge is identifying the right source of data.

You can list available extractors in your system in transaction RSA5. Some of them may require further processing before using.




When you double click on the extractor name, you can list exposed fields together with the information if the data source supports delta extraction.

**DataSource: Business content version Display**

Header Data

| | | | |
|---|---|---|---|
| DataSource | 2LIS_11_VAITM | Package | MCEX |
| Description | Sales Document Item Data | | |

Extraction

| | | | |
|---|---|---|---|
| ExtractStruct. | MC11VA0ITM | | |
| Direct Access | D | | |
| Delta Update | ☑ | DataSource for Reconciliation | ☐ |

| Field Name | Short text | Selection | Hide field | Inversion | Field only.. |
|---|---|---|---|---|---|
| ABGRU | Reason for Rejection of Sales Documents | ☐ | ☐ | ☐ | ☐ |
| ABSTA | Rejection Status (Item) | ☐ | ☐ | ☐ | ☐ |
| AEDAT | Date of Last Change | ☐ | ☐ | ☐ | ☐ |
| ANGDT | Quotation/Inquiry is Valid From | ☐ | ☐ | ☐ | ☐ |
| ANZAUPO | Number of Order Items | ☐ | ☐ | ☑ | ☐ |
| APOPLANNED | Planning in APO | ☐ | ☐ | ☐ | ☐ |
| AUART | Sales Document Type | ☑ | ☐ | ☐ | ☐ |
| AUGRU | Order Reason (Reason for the Business ... | ☐ | ☐ | ☐ | ☐ |
| AWAHR | Order Probability of the Item | ☐ | ☐ | ☐ | ☐ |
| BNDDT | Date Until Which Bid/Quotation is Bindin... | ☐ | ☐ | ☐ | ☐ |
| BRGEW | Gross Weight of the Item | ☐ | ☐ | ☑ | ☐ |
| BUKRS | Company Code | ☑ | ☐ | ☐ | ☐ |
| BWAPPLNM | Application Component | ☐ | ☐ | ☐ | ☐ |
| BWVORG | SAP BW transaction key | ☐ | ☐ | ☐ | ☐ |
| BZIRK | Sales District | ☐ | ☐ | ☐ | ☐ |
| CHARG | Batch Number | ☐ | ☐ | ☐ | ☐ |
| CMKUA | Credit data exchange rate for requested.. | ☐ | ☐ | ☐ | ☐ |
| EAN11 | International Article Number (EAN/UPC) | ☐ | ☐ | ☐ | ☐ |
| ERDAT | Date on which the record was created | ☑ | ☐ | ☐ | ☐ |
| ERNAM | Name of Person who Created the Object | ☐ | ☐ | ☐ | ☐ |

In the previous episode, I mentioned that there is no timestamp information in OData service API_SALES_ORDER_SRV for entity A_SalesOrderItem. Therefore, each time we had to extract a full dataset, which was not ideal. The SAP extractor 2LIS_11_VAITM, which I'm going to use today, should solve that problem.

I found it much more difficult to find CDS views that support data extraction and delta management. There is a View Browser Fiori application that lists available CDS Views in the system, but it lacks some functionality to make use of it – for example, you can't set filters

on annotations. The only workaround I found was to enter @Analytics.dataextraction.enabled:true in the search field. This way you can at least identify CDS Views that can be used for data extraction. But to check if they support delta management you have to manually check view properties.

I_GLAccountLineItemRawData GL Account Line Item without ledger logic

Application Component: FI-GL-IS
Description: Information System
Tags:

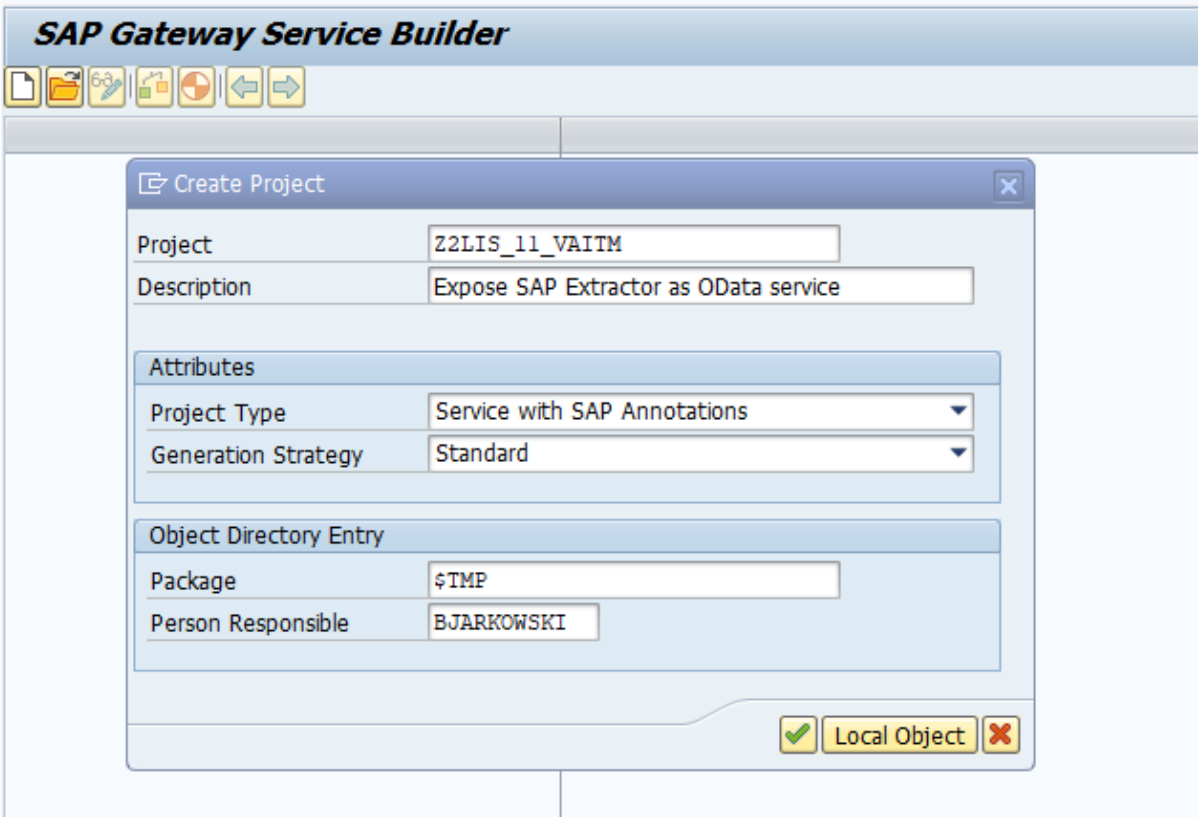
DEFINITION ANNOTATION CROSS REFERENCE

| | |
|---|---|
| ANALYTICS.DATAEXTRACTION.ENABLED | true |
| ANALYTICS.DATAEXTRACTION.DELTA.CHANGEDATACAPTURE.MAPPING$1$.TABLE | 'ACDOCA' |
| ANALYTICS.DATAEXTRACTION.DELTA.CHANGEDATACAPTURE.MAPPING$1$.ROLE | #MAIN |
| ANALYTICS.DATAEXTRACTION.DELTA.CHANGEDATACAPTURE.MAPPING$1$.VIEWELEMENT$1$ | 'SourceLedger' |
| ANALYTICS.DATAEXTRACTION.DELTA.CHANGEDATACAPTURE.MAPPING$1$.VIEWELEMENT$2$ | 'CompanyCode' |

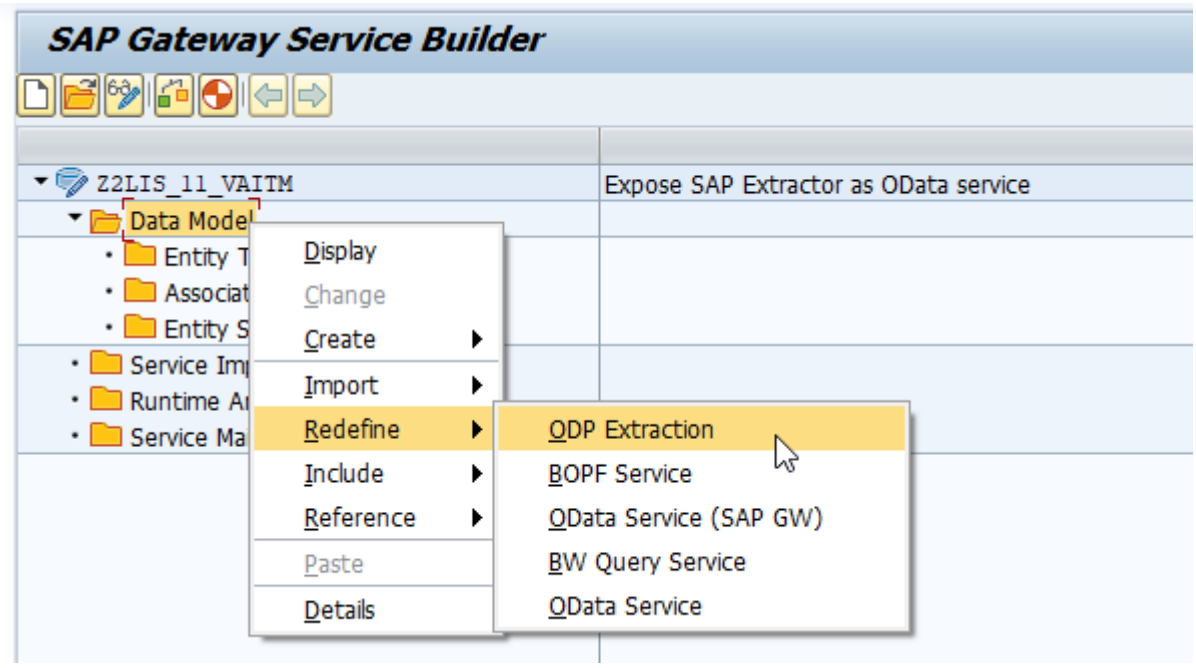
Some CDS Views are still using the timestamp column to identify new and changed information, but as my source system is SAP S/4HANA 1909, I can benefit from the enhanced Change Data Capture capabilities, which use the SLT framework and database triggers to identify delta changes. I think it's pretty cool. If you consider using CDS Views to extract SAP data, please check fantastic blog posts published by Simon Kranig. He nicely explains the mechanics of data extraction using CDS Views.

https://blogs.sap.com/2019/12/13/cds-based-data-extraction-part-i-overview/

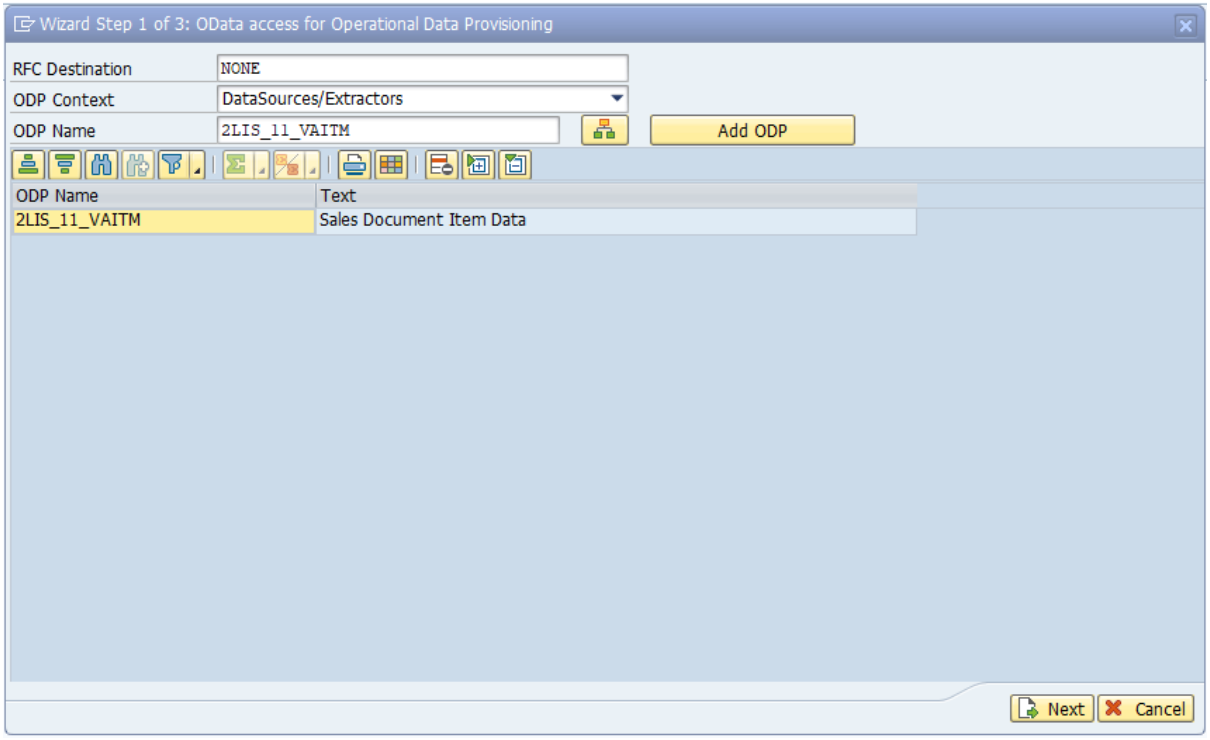
I'll be using the extractor 2LIS_11_VAITM to get item details and the I_GLAccountLineItemRawData to read GL documents. To expose the object as an OData service create a new project in transaction SEGW:



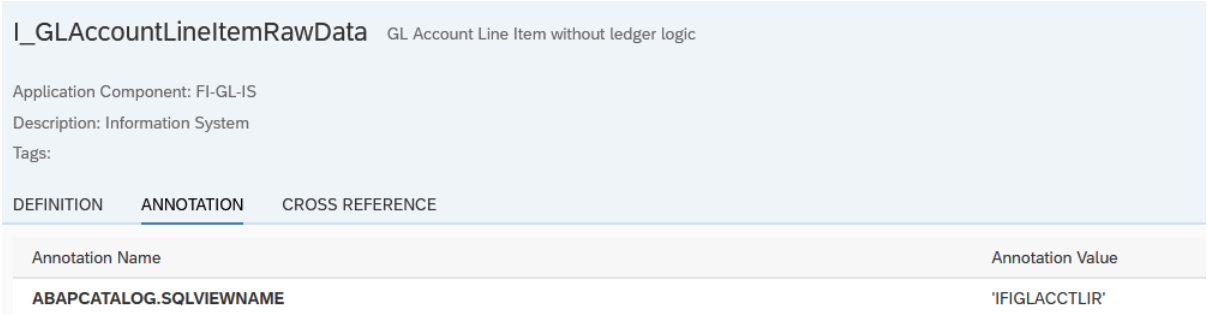

Then select Data Model and open the context menu. Choose Redefine -> ODP Extraction.



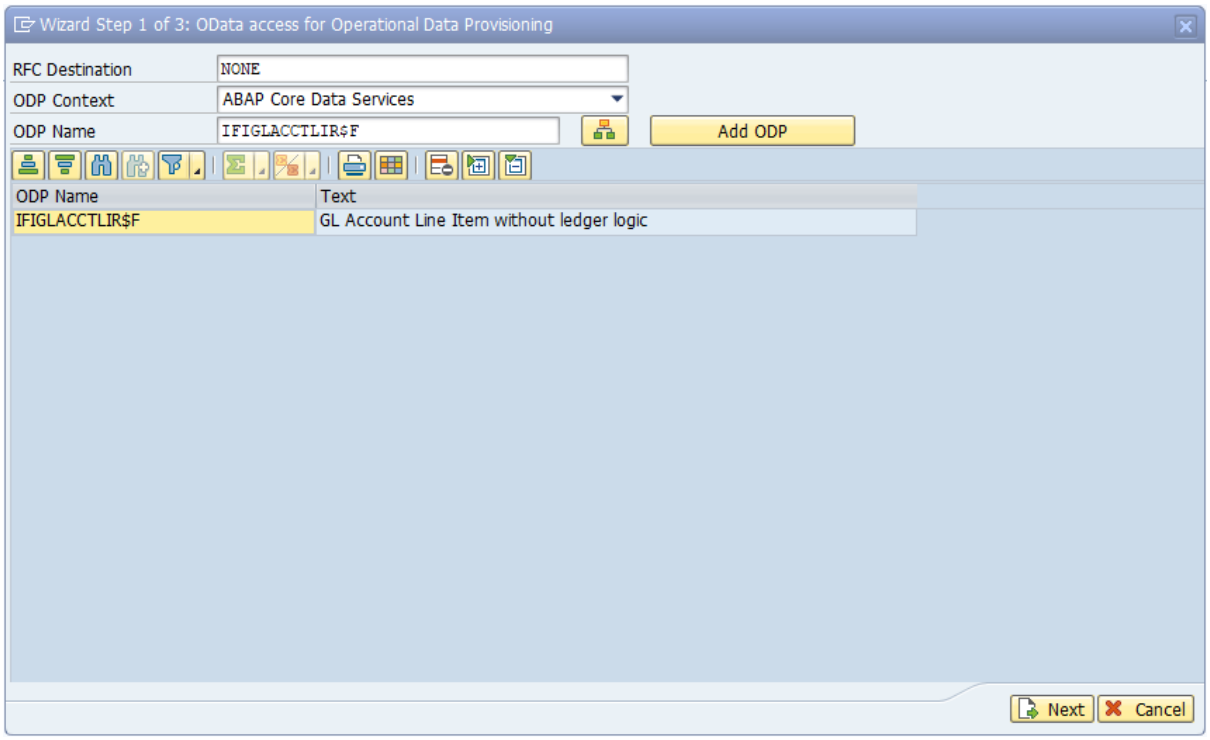

Select the object to expose. If you want to use an extractor, select DataSources / Extractors as the ODP context and provide the name in the ODP Name field:



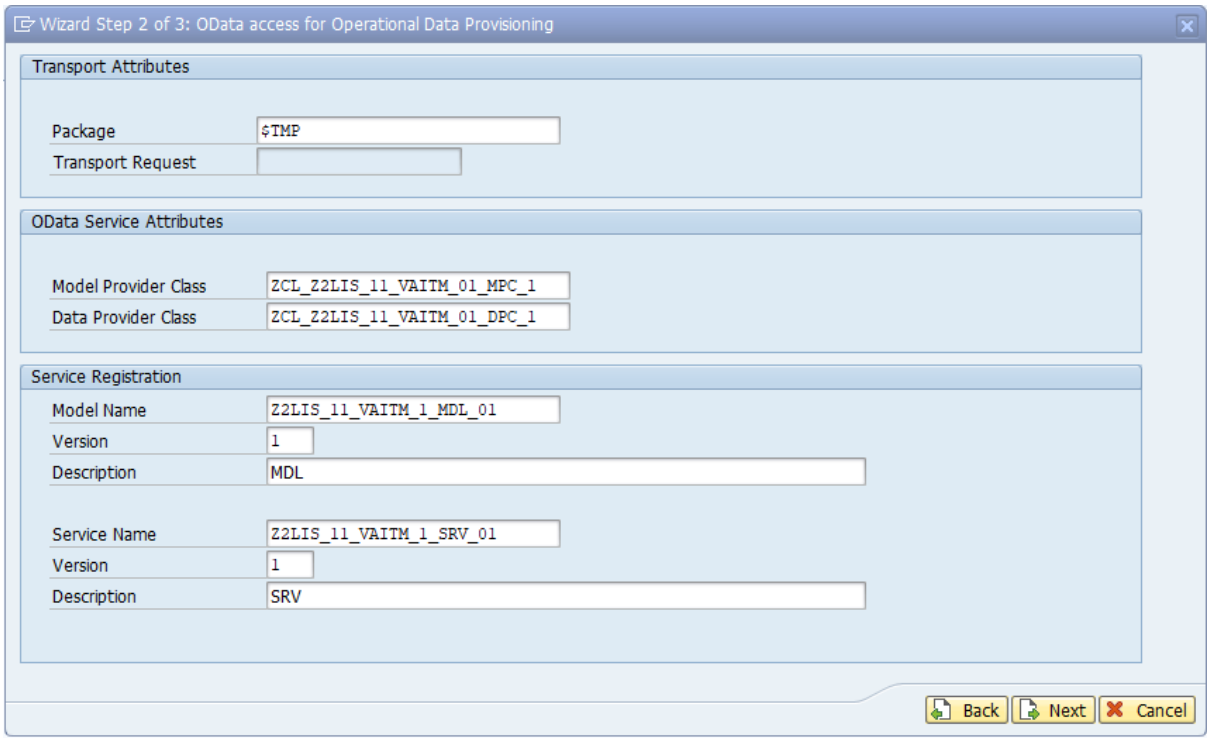

To expose a CDS View, we need to identify the SQL Name. I found it the easiest to use the View Browser and check the SQLViewName annotation:



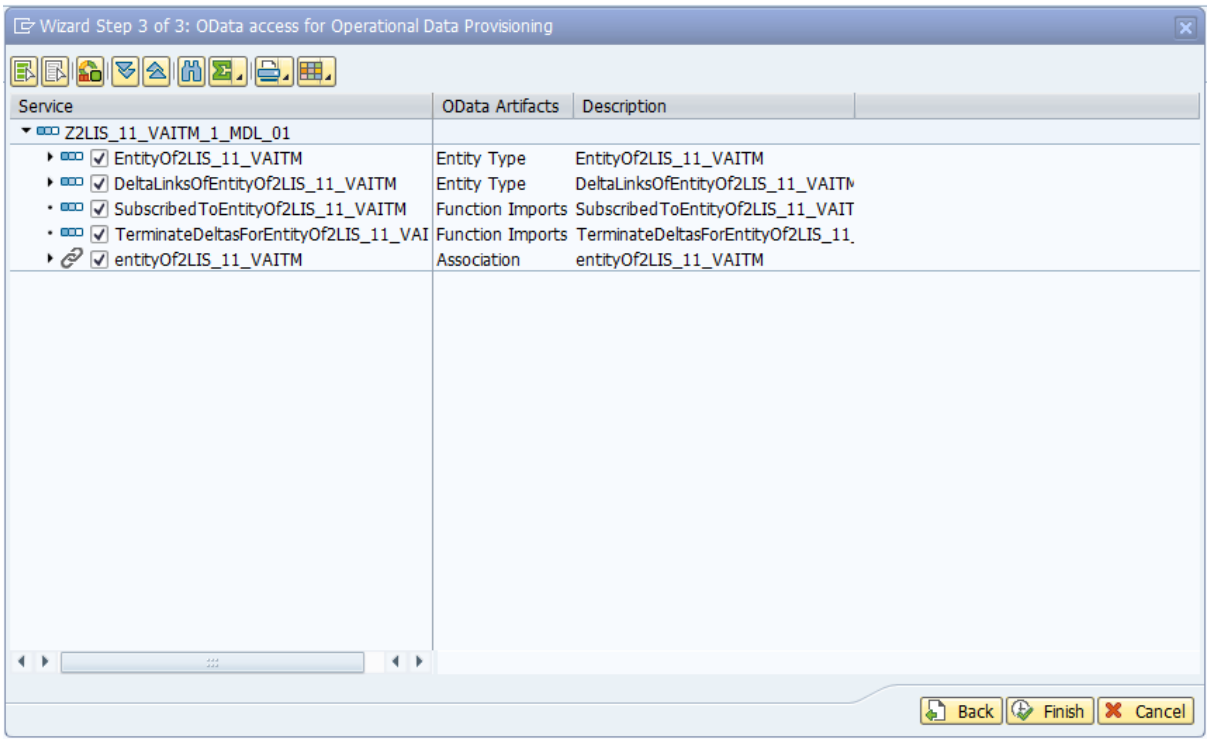

Then in the transaction SEGW create a new project and follow the exact same steps as for exposing extractors. The only difference is the Context, which should be set to ABAP Core Data Services.




Further steps are the same, no matter if you work with an extractor or CDS view. Click Next. The wizard automatically creates the data model and OData service, and you only have to provide the description.




Click Next again to confirm. In the pop-up window select all artifacts and click Finish.




The last step is to Generate Runtime Object which you can do from the menu: Project -> Generate. Confirm model definition and after a minute your OData service will be ready for registration.

Model and Service Definition

Model Provider Class

| | |
|---|---|
| Class Name | ZCL_Z2LIS_11_VAITM_01_MPC_EXT |
| Base Class Name | ZCL_Z2LIS_11_VAITM_01_MPC |

Data Provider Class

☑ Generate Classes

| | |
|---|---|
| Class Name | ZCL_Z2LIS_11_VAITM_01_DPC_EXT |
| Base Class Name | ZCL_Z2LIS_11_VAITM_01_DPC |

Service Registration

| | |
|---|---|
| Technical Model Name | Z2LIS_11_VAITM_MDL_01 |
| Model Version | 1 |
| Technical Service Name | Z2LIS_11_VAITM_SRV_01 |
| Service Version | 1 |

Service Extension

☐ Overwrite Base/Extended Service

| | |
|---|---|
| Technical Service Name | |
| Service Version | 0 |

Open the Activate and Maintain Services report (/n/iwfnd/maint_service) to activate created OData services. Click Add button and provide the SEGW project name as Technical Service Name:

***Add Selected Services***

Get Services

Filter

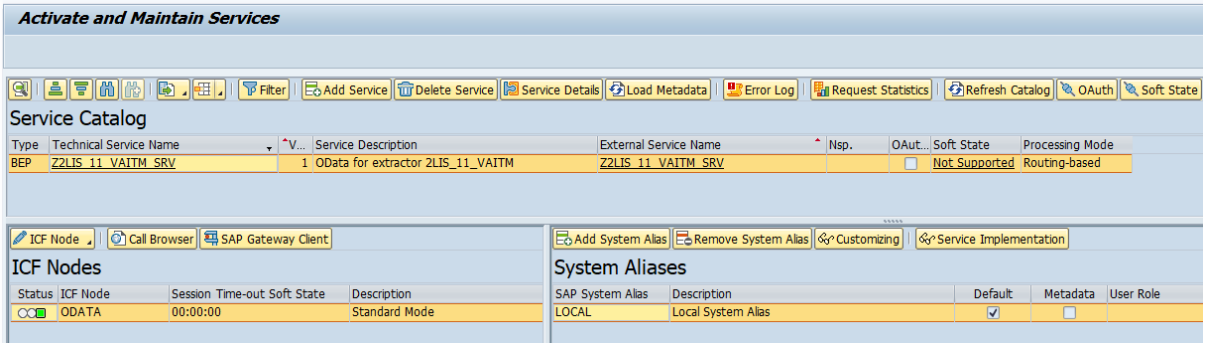
| | | | |
|---|---|---|---|
| System Alias | LOCAL | ☐ Co-Deployed | |
| Technical Service Name | Z2LIS_11_VAITM_SRV | Version | |
| External Service Name | | External Mapping ID | |

Add Selected Services

Select Backend Services

| Type | Technical Service Name | Ver... | Service Description | External Service Name | Namespace |
|---|---|---|---|---|---|
| BEP | Z2LIS_11_VAITM_SRV | 1 | OData for extractor 2LIS_11_VAITM | Z2LIS_11_VAITM_SRV | |

Click Add Selected Services and confirm your input. You should see a popup window saying the OData service was created successfully. Verify the system alias is correctly assigned and the ICF node is active:




OData service is now published and we can start testing it.

## EXTRACTING DATA FROM DELTA-ENABLED ODATA SERVICES

Let's take a closer look at how does data extraction works in delta-enabled OData service.

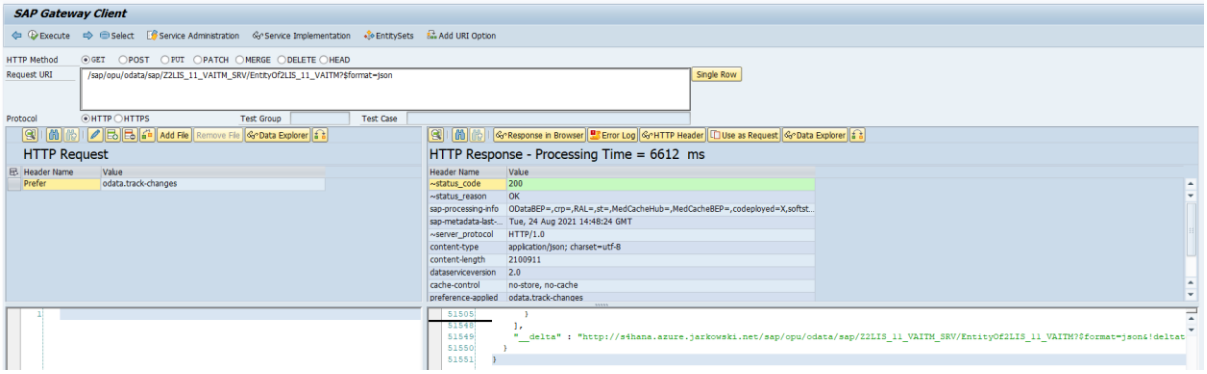
You have probably noticed during the service creation, that extractors and CDS views give you two entities to use:

- Representing the data source model, with the name starting with EntityOf<objectName>, FactsOf<objectName> or AttrOf<objectName> depending on the type of extractor or view
- Exposing information about current and past delta tokens, with the name starting with DeltaLinksOf<objectName>

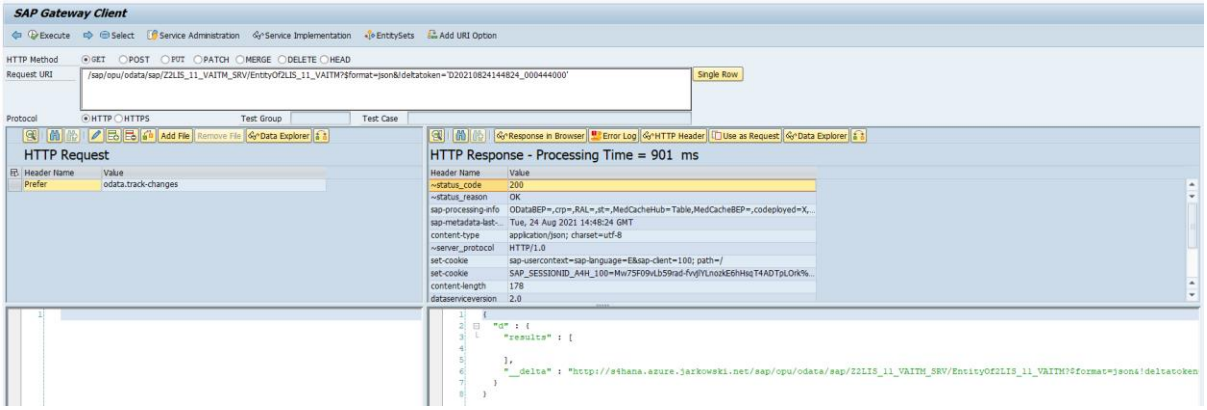
By default, if you send a request to the first service, you will retrieve a full dataset. Just like you'd work with any other OData services we covered in previous episodes. The magic happens if you add a special request header:

```
Prefer: odata.track-changes
```

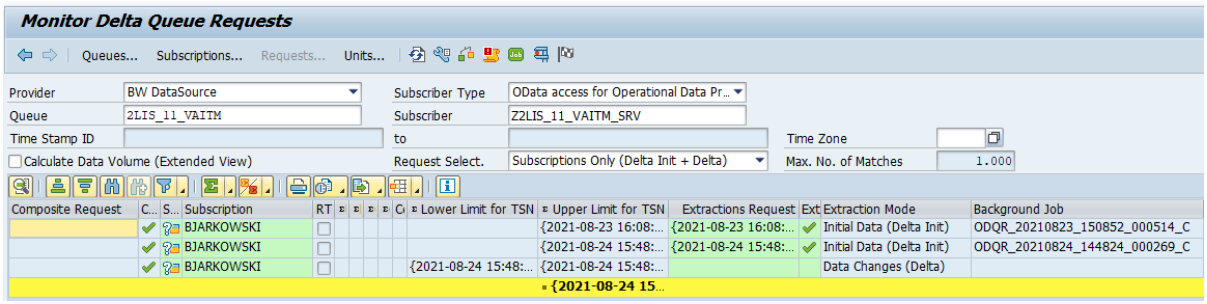
It tells the system that you want it to keep track of delta changes for this OData source. Then, as a result, in the response content, together with the initial full dataset, you can find an additional field __delta with the link you can use to retrieve only new and changed information.



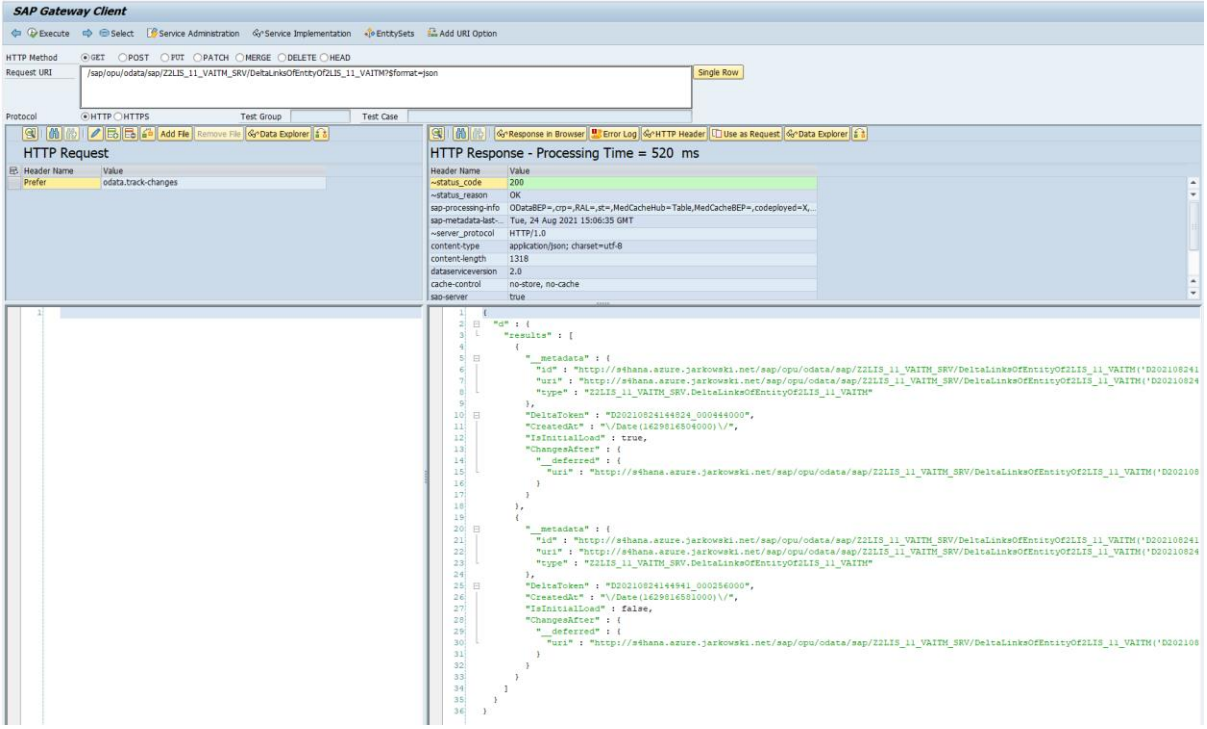

The additional header subscribes you to the delta queue, which tracks data changes. If you follow the __delta link, which is basically the OData URL with extra query parameter !deltatoken, you will retrieve only updated information and not the full data set.




In the SAP system, there is a transaction ODQMON that lets you monitor and manage subscriptions to the delta queue.




You can query the second entity, with the name starting with DeltaLinksOf<EntityName>, to receive a list of the current and past delta tokens.




We will use both entities to implement a pipeline in Synapse. Firstly, we will check if there are already open subscriptions. If not, then we'll proceed with the initial full data extraction. Otherwise, we will use the latest delta token to retrieve changes made since the previous extraction.

## IMPLEMENTATION

Open Synapse Studio and create a new pipeline. It will be triggered by the metadata one based on the ExtractionType field. Previously we have used the keywords Delta and Full to distinguish which pipeline should be started. We will use the same logic, but we'll define a new keyword Deltatoken to distinguish delta-enabled OData services.

I have added both exposed OData services to the metadata store together with the entity name. We won't implement any additional selection or filtering here (and I'm sure you know how to do it if you need it), so you can leave the fields Select and Filter empty. Don't forget to enter the batch size, as it's going to be helpful in the case of large datasets.

Query | Add | Edit | Select All | Column Options | Delete | Table Statistics | Refresh

| PARTITIONKEY | ROWKEY | BATCH | EXTRACTIONTYPE |
|---|---|---|---|
| API_BUSINESS_PARTNER | A_BusinessPartner | 10000 | Full |
| API_PRODUCT_SRV | A_Product | 1000 | Delta |
| API_SALES_ORDER_SRV | A_SalesOrder | 100 | Delta |
| API_SALES_ORDER_SRV | A_SalesOrderItem | 100000 | Full |
| Z2LIS_11_VAITM_SRV | EntityOf2LIS_11_VAITM | 100 | Deltatoken |
| ZI_GLACCOUNTLINEITEMRAWDATA_SRV | FactsOfIFIGLACCTLIR | 10 | Deltatoken |

Excellent. As I mentioned earlier, to subscribe to the delta queue, we have to pass an additional request header. Unfortunately, we can't do it at the dataset level (like we would

do for REST type connection), but there is a workaround we can use. When you define an OData linked service, you have an option of passing additional authentication headers. The main purpose of this functionality is to provide API Key for services that require this sort of authentication. But it doesn't stop us from re-using this functionality to pass our custom headers.

There is just one tiny inconvenience that you should know. As the field should store an authentication key, the value is protected against unauthorized access. It means that every time you edit the linked service, you have to retype the header value, exactly the same as you would do with the password. Therefore if you ever have to edit the Linked Service again, remember to provide the header value again.

Let's make changes to the Linked Service. We need to create a parameter that we will use to pass the header value:

```
"Header": {
        "type": "String"
}
```

Then to define authentication header add the following code under the typeProperties:

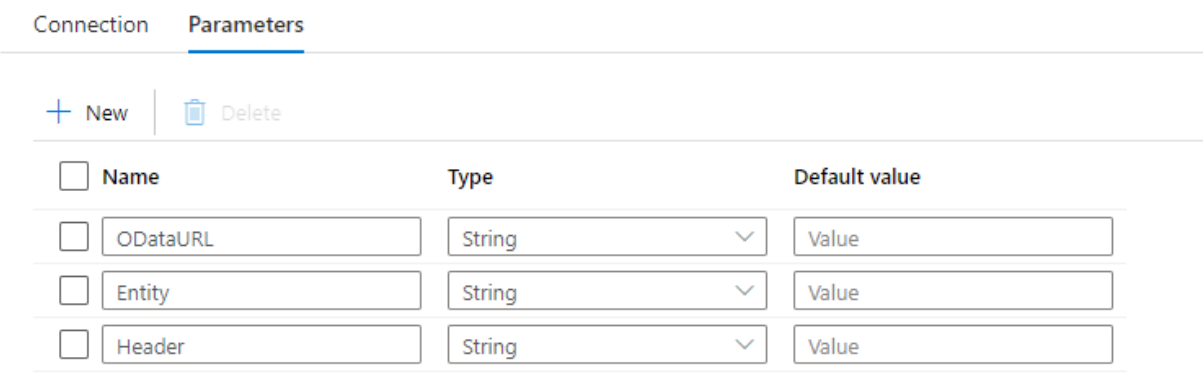
```
"authHeaders": {
    "Prefer": {
        "type": "SecureString",
        "value": "@{linkedService().Header}"
    }
},
```

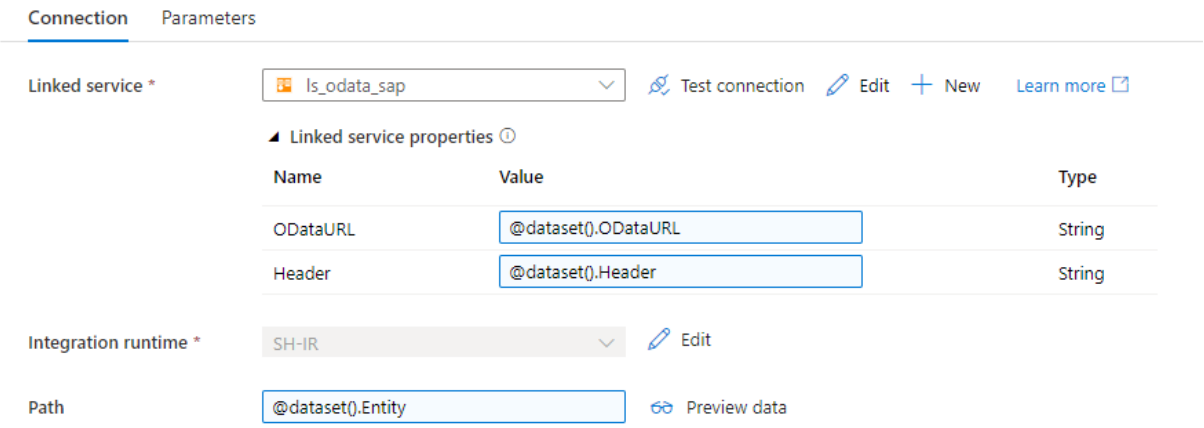
For reference, below, you can find the full definition of my OData linked service.

```
{
    "name": "ls_odata_sap",
    "type": "Microsoft.Synapse/workspaces/linkedservices",
    "properties": {
        "type": "OData",
        "annotations": [],
        "parameters": {
            "ODataURL": {
                "type": "String"
            },
            "Header": {
                "type": "String"
            }
        },
        "typeProperties": {
            "url": "@{linkedService().ODataURL}",
            "authenticationType": "Basic",
            "userName": "bjarkowski",
            "authHeaders": {
                "Prefer": {
                    "type": "SecureString",
                    "value": "@{linkedService().Header}"
                }
            },
            "password": {
```

```
                "type": "AzureKeyVaultSecret",
                "store": {
                    "referenceName": "ls_keyvault",
                    "type": "LinkedServiceReference"
                },
                "secretName": "s4hana"
            }
        },
        "connectVia": {
            "referenceName": "SH-IR",
            "type": "IntegrationRuntimeReference"
        }
    }
}
```

The above change requires us to provide the header every time we use the linked service. Therefore we need to create a new parameter in the OData dataset to pass the value. Then we can reference it using an expression:



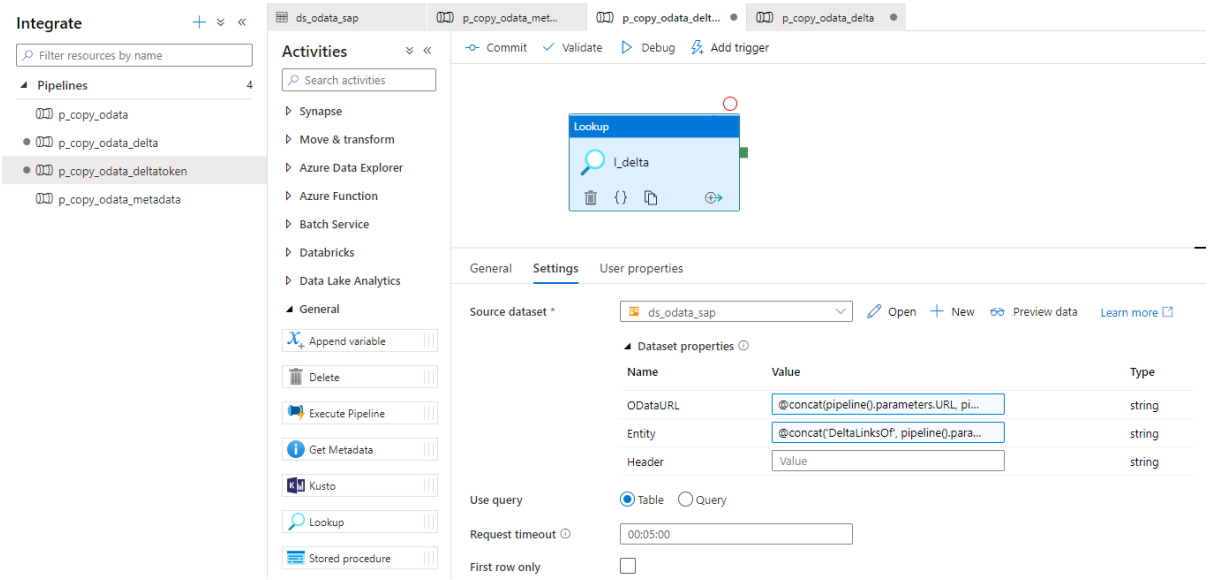




In Synapse, every parameter is mandatory, and we can't make them optional. As we use the same dataset in every pipeline, we have to provide the parameter value in every activity that uses the dataset. I use the following expression to pass an empty string.

@coalesce(null)

Once we enhanced the linked service and make corrections to all activities that use the affected dataset it's time to add Lookup activity to the new pipeline. We will use it to check if there are any open subscriptions in the delta queue. The request should be sent to the DeltaLinksOf entity. Provide following expressions:

```
ODataURL: @concat(pipeline().parameters.URL,
pipeline().parameters.ODataService, '/')
Entity: @concat('DeltaLinksOf', pipeline().parameters.Entity)
Header: @coalesce(null)
```

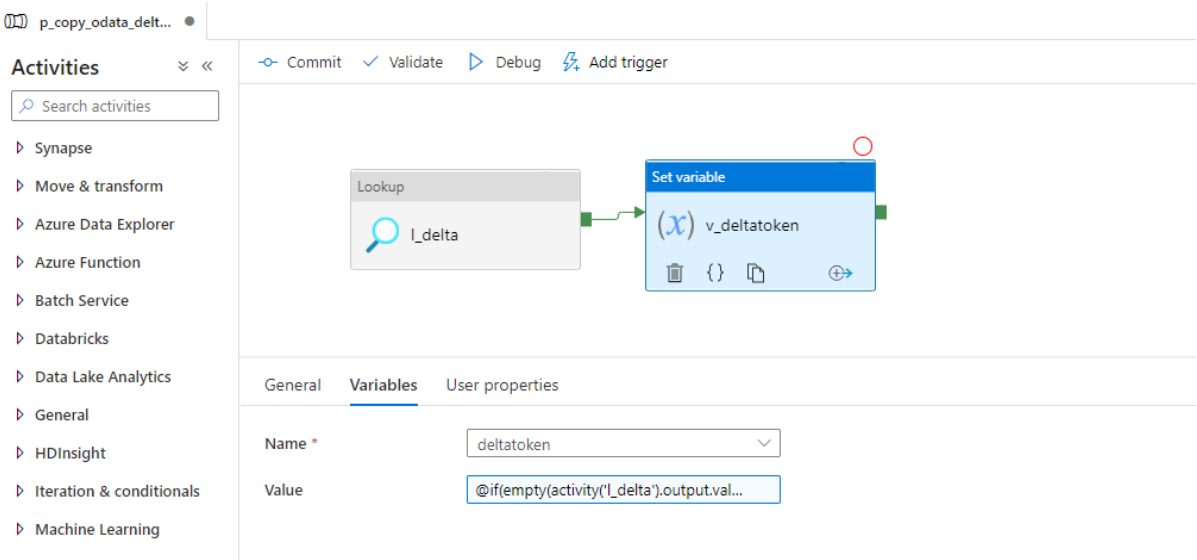



To get the OData service name to read delta tokens I concatenate 'DeltaLinkOf' with the entity name that's defined in the metadata store.

Ideally, to retrieve the latest delta token, we would pass the $orderby query parameter to sort the dataset by the CreatedAt field. But surprisingly, it is not supported in this OData service. Instead, we'll pull all records and use an expression to read the most recent delta token.

Create a new variable in the pipeline and add Set Variable activity. The below expression checks if there are any delta tokens available and then assigns the latest one to the variable.

```
@if(empty(activity('l_delta').output.value), '',
last(activity('l_delta').output.value).DeltaToken)
```

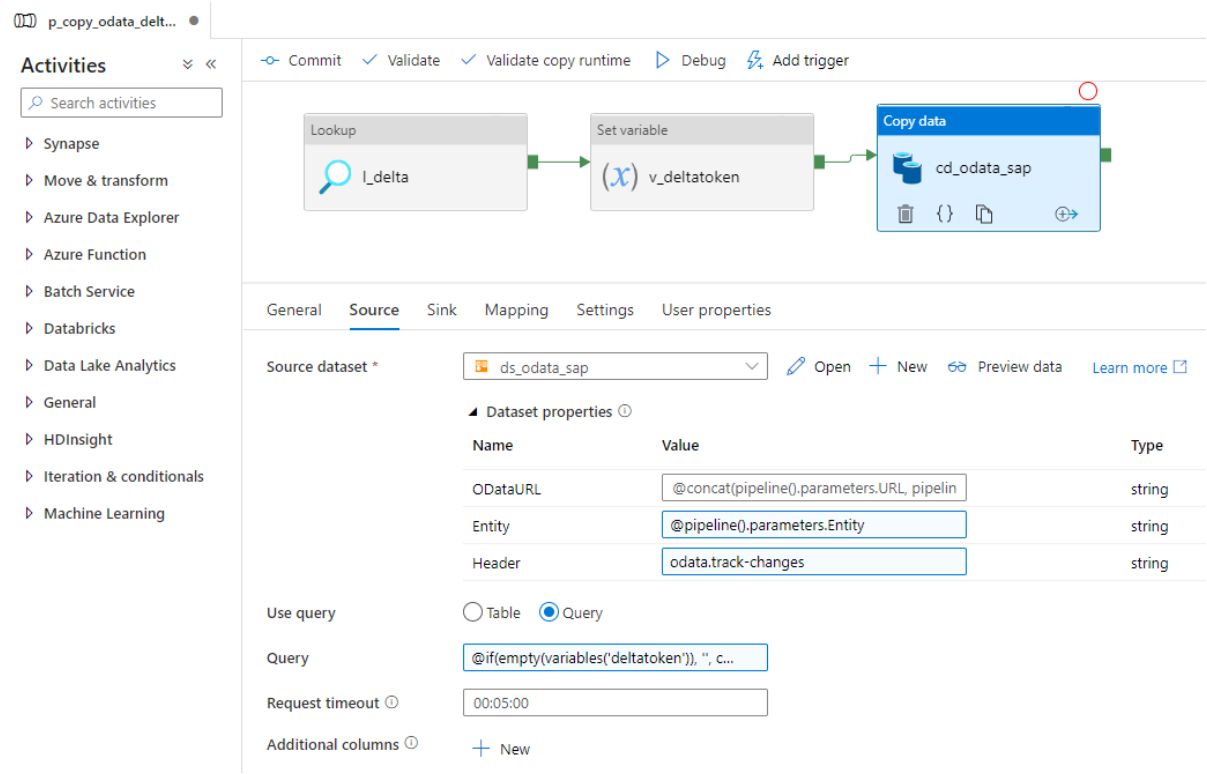



Add the Copy Data activity to the pipeline. The ODataURL and Entity parameters on the Source tab use the same expression as in other pipelines, so you can copy them and I won't repeat it here. As we want to enable the delta capabilities, provide the following value as the header:

```
odata.track-changes
```

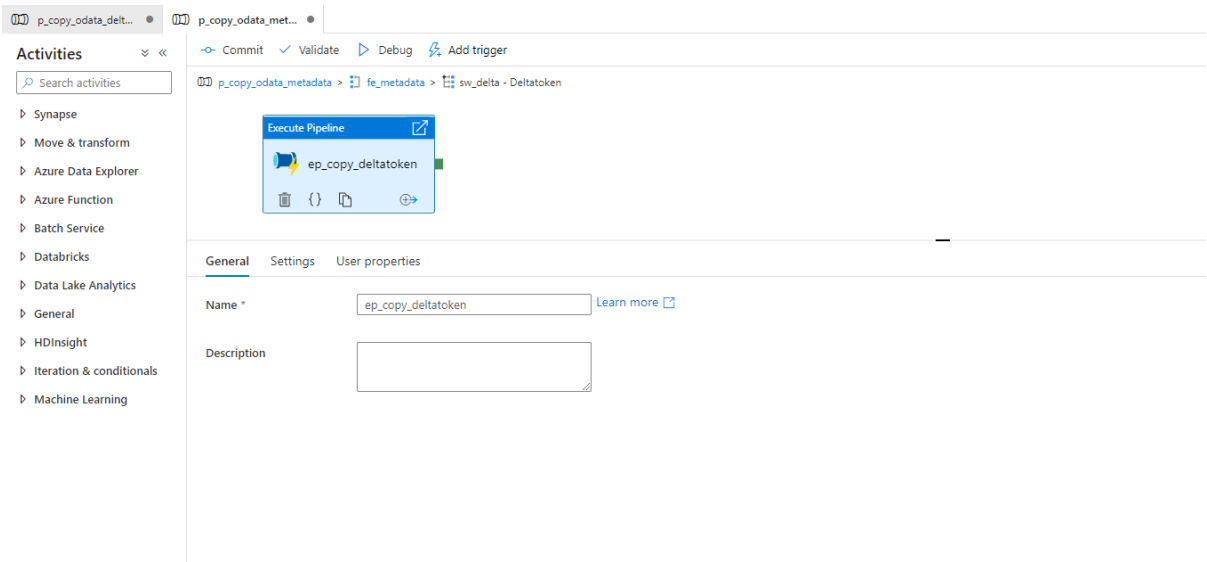
Change the Use Query setting to Query. The following expression checks the content of the deltatoken variable. If it's not empty, its value is concatenated with the !deltatoken query parameter and passed to the SAP system. Simple and working!

```
@if(empty(variables('deltatoken')), '', concat('!deltatoken=''',
variables('deltatoken'), ''''))
```




Don't forget to configure the target datastore in the Sink tab. You can copy all settings from one of the other pipelines – they are all the same.

We're almost done! The last thing is to add another case in the Switch activity on the metadata pipeline to trigger the newly created flow whenever it finds delta token value in the metadata store.

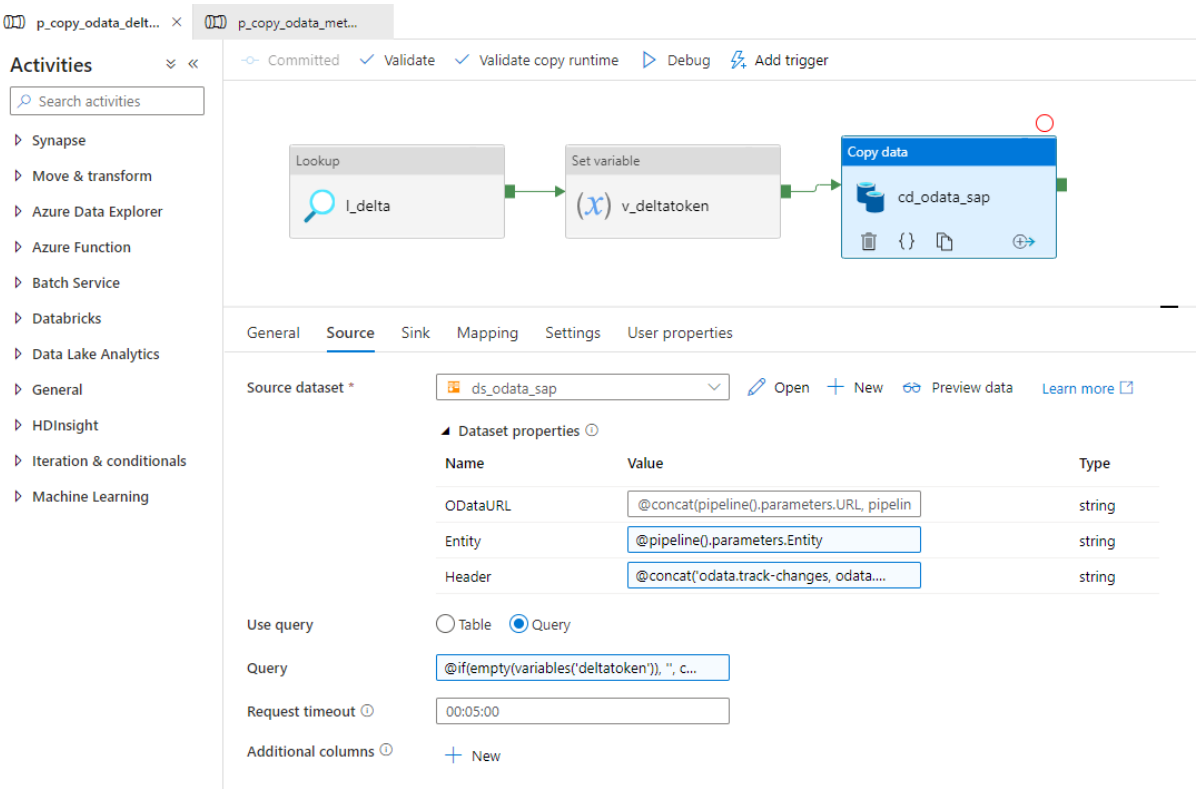



We could finish here and start testing. But there is one more awesome thing I want to show you!

The fourth part of the series focuses on paging. To deal with very large datasets, we implemented a special routine to split requests into smaller chunks. With SAP extractors and CDS views exposed as OData, we don't have to implement a similar architecture. They support server-side pagination and we just have to pass another header value to enable it.

Currently, in the Copy Data activity, we're sending odata.track-chages as the header value. To enable server-side paging we have to extend it with odata.maxpagesize=<batch_size>. Let's make the correction in the Copy Data activity. Replace the Header parameter with the following expression:

```
@concat('odata.track-changes, odata.maxpagesize=',
pipeline().parameters.Batch)
```




Server-side pagination is a great improvement comparing with the solution I desribed in episode four.

## EXECUTION AND MONITORING

I will run two tests to verify the solution works as expected. Firstly, after ensuring there are no open subscriptions in the delta queue, I will extract all records and initialize the delta load. Then I'll change a couple of sales order line items and run the extraction process again.

Let's check it!

Pipeline runs

Triggered Debug | Rerun | Cancel | Refresh | Edit columns | List Gantt

Search by run ID or name | Local time : **Last 24 hours** | Pipeline name : **All** | Status : **All** | Runs : Latest runs | Add filter

Showing 1 - 52 items

| Pipeline name | Run start | Run end | Duration | Triggered by | Status |
|---|---|---|---|---|---|
| p_copy_odata_deltatoken | 8/25/21, 2:44:00 PM | 8/25/21, 2:45:33 PM | 00:01:33 | 5023c2c4-7f4a-492d-8f1 | Succeeded |
| p_copy_odata_deltatoken | 8/25/21, 2:43:15 PM | 8/25/21, 2:43:57 PM | 00:00:41 | 951f3b72-a11c-485f-bcc | Succeeded |
| p_copy_odata | 8/25/21, 2:42:49 PM | 8/25/21, 2:43:12 PM | 00:00:22 | f6f757be-72bd-4c32-8bc | Succeeded |
| p_copy_odata_delta | 8/25/21, 2:42:12 PM | 8/25/21, 2:42:45 PM | 00:00:33 | 8b18365d-b5a6-422d-9b | Succeeded |
| p_copy_odata_delta | 8/25/21, 2:41:27 PM | 8/25/21, 2:42:09 PM | 00:00:41 | ffab8b63-9e1d-4419-868 | Succeeded |
| p_copy_odata | 8/25/21, 2:40:51 PM | 8/25/21, 2:41:25 PM | 00:00:34 | 3eb3fac2-3d47-47b8-aac | Succeeded |
| p_copy_odata_metadata | 8/25/21, 2:40:45 PM | 8/25/21, 2:45:41 PM | 00:04:55 | Manual trigger | Succeeded |

The first extraction went fine. Out of 6 child OData services, two were processed by the pipeline supporting delta token. That fits what I have defined in the database. Let's take a closer look at the extraction details. I fetched 379 sales order line items and 23 316 general ledger line items, which seems to be the correct amount.

**Activity run id:** 58a41547-30a2-48d6-9258-bad72cbd40aa

OData-Preview → Succeeded → Azure Data Lake Storage Gen2
Region: West Europe

| | | | |
|---|---|---|---|
| Data read: | 944.977 KB | Data written: | 65.392 KB |
| Rows read: | 379 | Files written: | 1 |
| Peak connections: | 1 | Rows written: | 379 |
| | | Peak connections: | 1 |

| | |
|---|---|
| Copy duration | 00:00:17 |
| Throughput: | 55.587 KB/s |

OData-Preview → Azure Data Lake Storage Gen2

| | |
|---|---|
| Start time | 8/25/21, 2:43:38 PM |
| Used parallel copies | 1 |
| Duration | 00:00:17 |

| Details | | Working duration | Total duration |
|---|---|---|---|
| Queue | | | 00:00:08 |
| Transfer | Time to first byte | 00:00:04 | 00:00:09 |
| | Reading from source | 00:00:06 | |
| | Writing to sink | 00:00:00 | |

In the ODQMON transaction, I can see two open delta queue subscriptions for both objects, which proves the request header was attached to the request. I changed one sales order line item and added an extra one. Let's see if the pipeline picks them up.

Wait! Three records? How is that possible if I only made two changes?

Some delta-enabled OData services provide the functionality not only to track new items but also records deleted information. That's especially useful in the case of sales orders. Unlike a posted accounting document, which can be only 'removed' by reversal posting, a sales order is open for changes much longer. Therefore, to have consistent data in the lake, we should also include deleted information.

But still, why did I extract three changes if I only made two changes? Because that's how this extractor works. Instead of only sending the updated row, it firstly marks the whole row for deletion and then creates a new one with the correct data.

Results Messages

View Table Chart Export results

Search

| ROCANCEL | VBELN | ABSTA | POSNR | UVALL | UVFAK | UVPRS | UVVLK |
|---|---|---|---|---|---|---|---|
| X | 0000000077 | A | 000030 | C | C | C | C |
| | 0000000077 | A | 000030 | C | C | C | C |
| | 0000000077 | A | 000040 | C | C | C | C |

So the only thing left is the validation of the server-side paging. And I have to admit it was a struggle as I couldn't find a place in Synapse Pipelines to verify the number of chunks. Eventually, I had to use ICM Monitor to check logs at the SAP application servers. I found there an entry suggesting the paging actually took place – can you see the !skiptoken query parameter received by the SAP system?

```
[Thr 139633941321472] HTTP request (raw) [4/332367/33]:
[Thr 139633941321472]   GET /sap/opu/odata/sap/Z2LIS_11_VAITM_SRV/EntityOf2LIS_11_VAITM?$skiptoken=220210825123457_000928000_0000000001_0000000010_0000000032 HTTP/1.1
[Thr 139633941321472]   authorization: XXX
[Thr 139633941321472]   user-agent: azure-data-factory/2.0
[Thr 139633941321472]   prefer: odata.track-changes, odata.maxpagesize=10
[Thr 139633941321472]   host: [redacted]
[Thr 139633941321472]   cookie: sap-usercontext=sap-client=100; SAP_SESSIONID_A4H_100=XXX;
```

Do you remember that when you run delta-enabled extraction, there is an additional field __delta with a link to the next set of data? Server-side paging works in a very similar way. At the end of each response, there is an extra field __skip with the link to the next chunk of data. Both solutions use tokens passed as the query parameters. As we can see, the URL contains the token, which proves Synapse used server-side pagination to read all data.

It seems everything is working fine! Great job!

## EPILOGUE

Next week there won't be another episode of the OData extraction series. During the last seven weeks, I covered all topics I considered essential to create a reliable data extraction process using OData services. Initially, we built a simple pipeline that could only process a single (and not containing much data) OData service per execution. It worked well but was quite annoying. Whenever we wanted to extract data from a couple of services, we had to modify the pipeline. Not an ideal solution.

But I would be lying if I said we didn't improve! Things got much better over time. In the second episode, we introduced pipeline parameters that eliminated the need for manual changes. Then, another episode brought metadata store to manage all services from a single place. The next two episodes focus on performance. I introduced the concept of paging to deal with large datasets, and we also discussed selects and filters to reduce the amount of data to replicate. The last two parts were all about delta extraction. I especially wanted to cover delta processing using extractors and CDS views as I think it's powerful, yet not commonly known.

Of course, the series doesn't cover all aspects of data extraction. But I hope this blog series gives you a strong foundation to find solutions and improvements on your own. I had a great time writing the series, and I learnt a lot! Thank you!