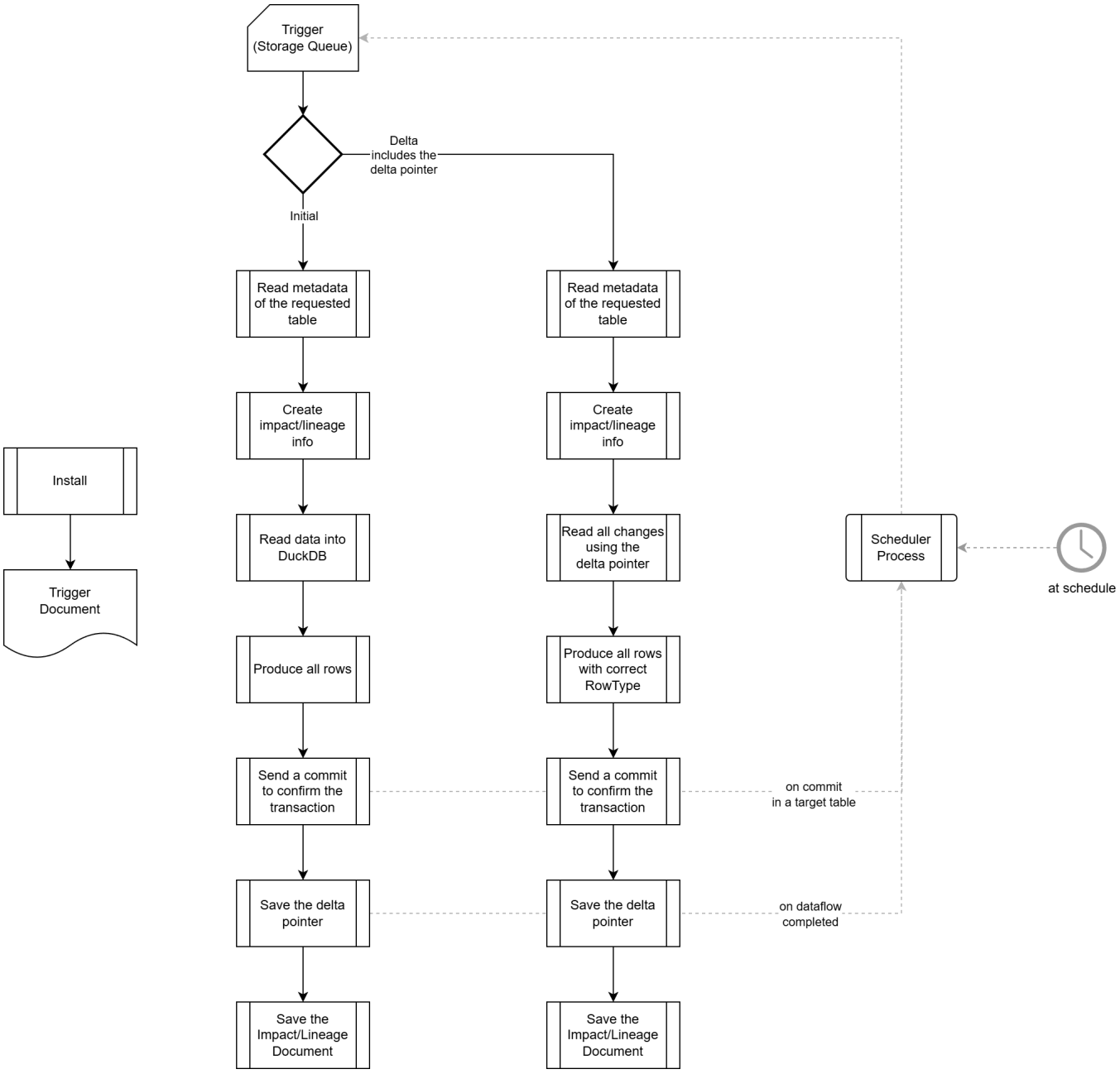
Decisions before writing a pipeline

We follow a shift-left paradigm, which means we extract from the source not only the data but also metadata and knowledge the developer acquires during the discussion with the source system owner.

Anatomy of a batch job



Logical hierarchy

One of the first decisions will be how many repos, how many function apps, how many functions,...

Let's take SAP as an example of a source. We want to read 10'000 tables from SAP, some of the tables are very large and must be broken apart into multiple parallel loaded partitions.

The function app defines in the trigger document, what should be executed when. This can be separated by function name by dataflow name by partition number. For this use case it does not make sense to write many functions, one function that gets the table name (=dataflow name) is enough. This function, when called, connects to SAP, reads the table metadata of the requested table and processes it. In parallel another instance of the same function loads another table.

If a table is too big to be loaded as one unit, we can further be split into multiple partitions. Partitions are optional. Even dataflows are optional.

Whatever is defined in the trigger document, the scheduler service will send one queue message for all functions/dataflows/partitions each as a queue-trigger and depending on the function app configuration, n instances are executed in parallel.

But from an abstract point of view, the hierarchy is:

- One repo one function app
- One function app many functions
- One function many dataflows (optional)
- One dataflow many partitions (optional)
- One execution one delta pointer
- One execution many target tables
- One execution many commits

Okay, so that shows what is possible, but what is advised? That obviously depends on the use case.

Number of repositories scope resp. function apps

The guidance is to have one repo/function app per domain. One repo for Labware source, another for SAP source, another for commercial pricing targets.

Important considerations

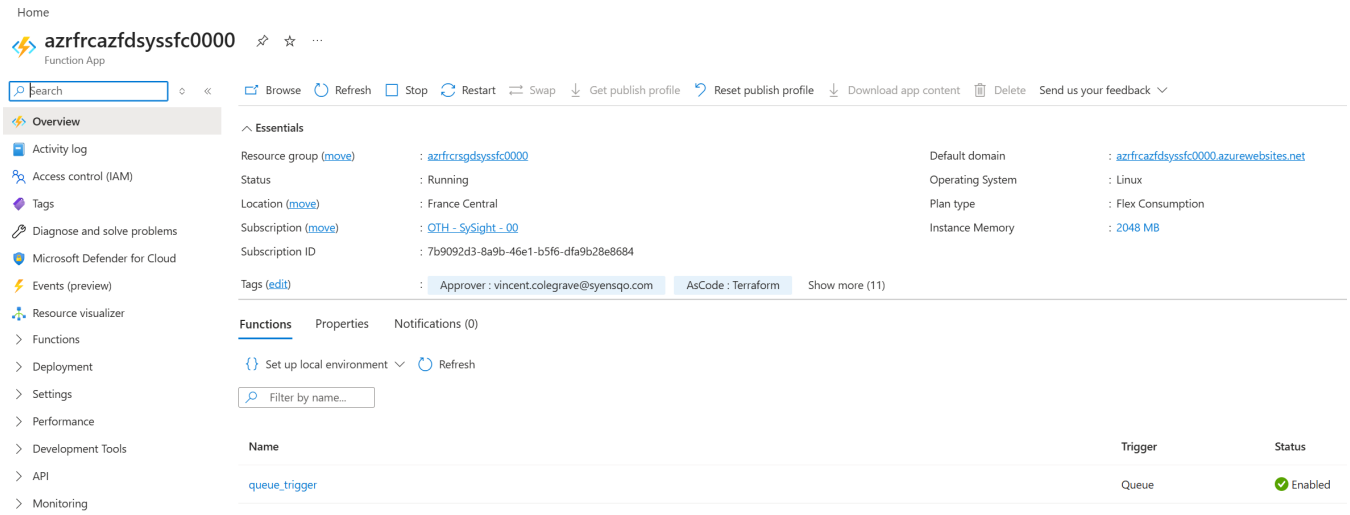
- A repository is deployed as a unit. If there is a bug fix committed into the repo, all its code is deployed. All functions it includes.
- More repositories create higher maintenance costs. Just imagine we would have one repo per table and there are 10000 tables. The initial deployment would mean to go into 10000 repos and deploy their code.
- The more people work within a repository, the higher the chances they work on the same code segments at the same time.

Number of functions

A repo contains a function_app.py file and there the functions are defined via annotations.

```
@app.function_name("queue_trigger")
@app.queue_trigger (
    arg_name="msg",
    queue_name=QUEUE_NAME,
    connection="StorageConnectString"
)
async def run_queue(msg: functions.QueueMessage):
    pass
```

This shows then up as a function called "queue_trigger" within the function app in Azure. Multiple function would mean multiple of these annotations with different names.



Reasons to have multiple functions is mainly for organizational reasons. Each function will call another class and therefore multiple developers can work on different files in the same repo.

One function/many dataflows

If the task is small enough for a single function execution, it is totally fine to have one function and no dataflows. In case of the SAP example, that would mean one function loads first the table 1, then table 2, until all 10'000 tables are loaded and then the function is finished. That would obviously be sub optimal. As each table load is an independent unit of work, it makes more sense to process each table separately. Via the trigger document the function defines the names of the dataflows and when it should be called, the function itself can read the name from the queue message and handle the load of this particular table.

In short, dataflows are used for parallel processing and separation of concerns.

Partitions

Are used to break apart the execution into smaller units. It is to be noted that each partition is also handled separately. An initial load is executed for each partition separately, not a dataflow as a whole. If the function is reading the data split into five partitions, Kafka will have five partitions and the Delta table will also have the same five partitions.

Examples

1. A function copying two independent tables: Because the tables are independent, each has its own delta pointer, each is committed independently, hence two triggers make the most sense.
2. A function copying two dependent tables: This is a scenario where tableA tableA and tableB tableB and both are committed in the same transaction, both share the same delta pointer. Hence one trigger is used to load both tables.
3. A function reading a single source table but loading two target tables: One trigger, two targets is just fine.
4. A function that can be called n times with the table name as parameter, the SAP example from above. In this case the function itself reads a table and writes a table. One function, multiple dataflows, maybe multiple partitions for some.

RowType

Because the data is streamed to Kafka, the information how it should be added to the Lakehouse tables must be provided.

In the simplest case the RowType is INSERT and the Lakehouse writer does append all data. That has consequences when a pipeline is restarted, however. The writer inserts all records immediately, even the not committed ones, so in such a scenario some data would be present twice.

A probably better approach is to use UPSERT as rowtype, as this instructs the writer to insert/update depending on if the primary key exists. While this solves the duplicate data, it has also potential side effects. First, the table must have a primary key for this to work properly. Second, deletes must also handled properly. And third, if the source updates the primary key, e.g. the record which used to have customer_id=1 is now customer_id=2, must be handled as if customer_id=1 was deleted and customer_id=2 is a new record.

Another scenario is an initial load. If the target table happens to have data, it should get wiped and replaced with new data. This is the scenario used above, where a first TRUNCATE row is sent followed by many data rows of type REPLACE. Note that the value schema might have some fields as not-nullable. In that case the truncate row must write dummy values into those, otherwise the row cannot be serialized.

T: truncate table - The database did truncate the entire table. These come not as millions of deletes but as a single change, a single command.
T: truncate with details - These are typically "alter table drop partition" like statements.
B: before image - Depending on the source, for an update the before and after image is available. This can be valuable information for downstream processing. Most important, what if the primary key changed? Without the before and after image values, this case cannot be resolved.
D: delete with before image - This is the variant where the delete contains the entire old record, all field values.
X: eXterminate - A delete with the primary key fields set only.
R: Replace - If records should be truncated and replaced by new ones. For example, we know the order has changed but we don't know what, order or order item. In that case the entire order is truncated and replaced with new rows.
P: Archived - For the source database it is a truncate or delete but the target should not remove the rows. This was just an archiving run of old data in the source.
A: Upsert - The table has a primary key but we do not know if that record exists in the target table already or not, if it is an insert or update.
I: Insert - Insert a new record
U: Update - The values of the update, the after image.

Transactions and commit

Each sent record has a transaction_id field and that is populated by the producer with its current value. The value is created/set at the creation of the producer or after the commit() method had been called. This allows to see which records belong to the same transaction and the commit() method writes the same id into the commit topic.

Transformations

Depending on the pipeline, it either includes more or less transformations. The rule of thumb shall be

- Ingestion pipelines offer source data to everybody who has the rights to use the data
  - Extract all columns except the ones that can be guaranteed to never be used. Keep in mind, we do not extract data for a project but to offer the data to all consumers.
  - Should not aggregate data. Only exception is when regulated data is aggregated to a level it is no longer regulated.
  - Rename the columns into something business orientated.
  - Interesting row transformations should be added, e.g. `ORDER_ENTRY_AMOUNT = case ABGRU <> 'X' then case AUGRU = 'R' then -NETWR else NETWR end else 0 end;` The thought is, these rules are known by the source system owner and during the data ingestion this person is interacted with, hence add such transformations here already.
  - Read only the changes for a delta run.
  - Compare the read data with the target table to create only the change data.
  - Avoid joining data in the source.
  - Apply a first set of data cleansing rules, those that act on single rows.
- Transformation pipelines loading into the Platform Zone cleanse the source data but the table is source system specific still.
  - Still no aggregations, avoid joins
- Transformation pipelines loading into the Enterprise Zone integrate the data into an enterprise wide Galaxy Schema.
  - The table name, primary key, naming conventions are all owned by the Data Model owner. You cannot decide on those yourself.
  - The columns are derived from the source
  - The goal is that all sources are incorporated into Star Schemas and Star Schemas share dimensions.

It is allowed to skip zones, e.g. read the data from the source and create a dimension table directly out of that. Or more common, all cleansing rules can be applied when reading the source and then the data deserves to be put into the Platform Zone directly.

Each pipeline also have the obligation to create the impact/lineage document, in above code the example `target_table.add_1_to_1_mapping (source_table, "Id", "Id")` was used, which describes how the tables and columns are mapped. This document provides a simple list of what sources contribute to a target (column and table level), the mapping formula (syntax not 100% defined) and a textual explanation of why that mapping was created.

Each row also has an `__audit` structure to put the applied cleansing rules, the rule result and the transformations for each single processed row. While the impact/lineage document describes the dataflow, so table1.Id table2.Id, the data here allows to list that the customer address was validated and is PASS, the customer name is not null, the gender column is set to `"-"` because this is a legal entity, not a natural customer. The information allows to check later the quality of the data.

The transformations are either done in native Python or SQL, whatever fits the requirement best. For example, programmatic logic can be done in Python, joins are easy in SQL.

- For SQL we use DuckDB, an in-process, in-memory, columnar orientated database.
- DuckDB itself can read from various sources, most important for us
  - From pyarrow structure mapped to Python arrays
    ```
    arrow_table = pa.Table.from_pydict({"a": [42]})
    duckdb.sql("SELECT * FROM arrow_table")
    ```
  - From CSV, Parquet or Json files
    ```
    duckdb.sql("SELECT * FROM 'example.json'")
    ```
  - From the delta lake in Fabric
    ```
    duckdb.sql(f"SELECT * FROM delta_scan('abfss://SySight-{env.name.lower()}-emea@onelake.dfs.fabric.
    microsoft.com/RAW_LAKEHOUSE.Lakehouse/Tables/{table_name}');")
    ```
- DuckDB provide data to Python as
  - In pyarrow table format
    ```
    tbl = duckdb.execute("SELECT * FROM items").arrow()
    ```
  - As array of dictionaries
    ```
    data = duckdb.execute("SELECT * FROM items").fetchall()
    ```
- pyarrow is a view on the data without copying the data from one format to the other.

This allows quite efficient dataflows, e.g. reading all source records into a Python dict, compare those with the current version in MS Fabric using the `delta_scan()` from clause.