Data Engineering with Dagster — Week 3

IO Management

CODE LOCATION

```
week_3/content/io_rety.py
```

Storing Data Between Steps

Data pipelines can take a long time to run. It is not uncommon to have pipelines that take hours or days to complete. Nothing is more frustrating than a job that has been executing for a long time that hits an exception on one of the final steps. How do you recover from this? Do you rerun the entire process? Have steps happened that shouldn't be rerun?

We have talked extensively about inputs and outputs and the Dagster philosophy of chaining them together. A very useful Dagster concept to become familiar with is the `io_manager`. An `io_manager` stores the information of outputs between steps in our DAG. Dagster provides a default configuration for I/O management, so even though we haven't explicitly set any `io_managers`, we have already been using them.

When not set explicitly, the default Dagster setting for I/O management uses the local file system. Our pipeline `hello_local_io_manager` also uses the file system, but we set that explicitly here by adding the `io_manager` to the resource definitions:

```
job_local_io_manager = hello_dagster.to_job(
    name="hello_local_io_manager",
    resource_defs={"io_manager": fs_io_manager},
)
```

This pipeline itself consists of just two steps, a long-running step and an unreliable step. This is a fairly common structure, where the pipeline first has to process a large amount of data and then send it to another API. For our purposes, the long-running step of our pipeline just returns a string, and our shaky API is just an op with a 50% chance to fail. With this simple example, we can get a sense of what a pipeline failure might look like:

In Dagit, we can see that our unreliable step tripped up our pipeline. If we go to execute the pipeline, we have the option "From failure." This will only perform the failed step. And since the output from our long-running step has been stored, it can still function as intended. Now we can try finishing our pipeline.

It took a few tries, but we got our job to complete! If you look at the right-hand panel, you can see that each subsequent run executed with its own run ID. Our original run (57f24898) is listed as the root. Every run after that is linked to the root job and its parent job, which is the job that occurred directly before it. So our final successful run 3be92115 is linked to both the root 57f24898 and its parent 9e16a0f4.

Automatic Retries

While it is nice to be able to re-execute our jobs. There are times when we'd rather not worry about the less reliable parts of our jobs. We can combine `io_managers` with a `RetryPolicy` to help ensure the success of our pipelines. The job `hello_local_io_manager_retry` defines a retry policy at the job level.

```
job_local_io_manager_retry = hello_dagster.to_job(
    name="hello_local_io_manager_retry",
    resource_defs={"io_manager": fs_io_manager},
    op_retry_policy=RetryPolicy(max_retries=10),
)
```

Now, any op that fails will retry up to 10 times. There are additional parameters we can set in a retry policy, such as delays and backoffs, but this should work fine. Now, if we execute our pipeline we can see what a failure looks like:

Even though there was a failure with our unreliable op, our pipeline still finished and is marked as having run successfully. Also, unlike our first example, everything executed within the same run ID.

OP LEVEL RETRY AND I/O

It is worth noting that both I/O managers and retry policies can be set at the individual op level. With retries this can be helpful if you have situations where certain operations can be retried safely in the same graph (maybe an API is being rated limited) compared to other steps where a failure represents a larger failure in the system.

For I/O managers there are times when a one or more op stores their data differently from the other steps in your pipeline. For small outputs with easily calculated outputs that only hold value for the pipeline run, you may want to use the in memory `mem_io_manager` manager. For ops that produce valuable data sets that you want to store in your data lake, S3 may be a better fit.

Storage Variations

One thing may have stood out if you were reading the code carefully. We defined our I/O management within our resource definition. This makes sense when you consider that the I/O management is just an external system outside Dagster used for storage. I/O management behaves like resources would for ops that have an external dependency (like we saw in week two).

That means we can use different storage layers with the same plug-ability as resources. For example, if our pipeline was truly doing large data processing with very large outputs, we might not want to store it on the local file system. Instead, we should use something like S3.

DIFFERENT ENVIRONMENTS

Like resources, different configurations of our job can use different I/O managers. Perhaps we want to use the local file system for local testing, but S3 for production. This flexibility in configuration can be helpful because writing our output to different layers can take additional time. If you are developing something locally and want a faster feedback loop, you may not be as interested with the additional overhead of writing to an external system.

You can also think of I/O managers as a cleaner way to write the output of your ops to storage. For example, if your pipeline is ultimately responsible for writing to a database, you could set I/O management to that database table, rather than define an op with that logic.

A great example of this is our ETL pipeline that writes to Postgres. Instead of including logic to write to Postgres, we could just have the op return the rows that should be written. Then, we can attach a Postgres I/O manager to the op to handle everything. Like most everything else in Dagster, we can define our own I/O managers with the `io_manager` decorator.

Summary

Let's make some changes to the pipeline and do a quick review of some of these settings.

```
@op(retry_policy=RetryPolicy(max_retries=5, delay=0.2),
out=Out(io_manager_key="fs_io"))
def time_consuming_step() -> str:
    return "dagster"


@op(out=Out(io_manager_key="s3_io"))
def unreliable_step(name: str):
    if randint(0, 1) == 1:
        raise Exception("Flaky op")
    print(f"Hello, {name}!")


@graph
def hello_dagster():
    unreliable_step(time_consuming_step())
```

```
quiz = hello_dagster.to_job(
    name="hello_local_io_manager",
    resource_defs={
        "fs_io": fs_io_manager,
        "s3_io": s3_pickle_io_manager,
    },
    op_retry_policy=RetryPolicy(max_retries=3),
)
```

QUICK QUIZ (SINGLE SELECTION)

How many times will the `unreliable_step` op retry, and what is its storage layer?

- ○ 3 retries, file system
- ○ 3 retries, s3
- ○ 5 retries, file system
- ○ 5 retries, s3

Submit → Clear

Being able to configure our pipelines with this level of detail ensures that we can handle occasional issues. Coupled with I/O management, we can safely store information for later retrieval. As you design your pipelines, it is important to know where they might fail and how you want to deal with those situations.

Object	Relationship	Description
I/O Manager	An I/O manager is defined at the resource level and is applied at the job level or the individual op level	An I/O manager defines the storage layer or how we store the output of jobs.

Retry Policy	A retry policy is applied at the job level or the individual op level	A retry policy determines if and how ops in a job should be executed again on failure.