Data Engineering with Dagster — Week 3

Partitions

CODE LOCATION

```
week_3/content/etl.py
```

Organizing Data

It is very common to have some form of partitioning for your job. This is because data itself usually has some form of partitioning when it is stored. Data is generally preserved in some organized way, whether by date or some other shared characteristic.

In Dagster, adding partitioning configurations to your jobs helps to add some of the same organization, ensuring that we can track executions of our pipelines and not miss anything. Good partitioning can also make it easier to safely re-execute jobs that were skipped or failed without custom workflows.

Partition Configs

We are going to make some modifications to our ETL Postgres jobs from the content section of week two. If you remember, that job took in a table name, created the table if it did not exist, and then wrote a random number of rows to it. We are going to add another field to the config schema for week three:

```
@op(config_schema={"table_name": String, "process_date": String},
required_resource_keys={"database"})
def create_table(context) -> String:
    ...
```

The `create_table` op now also requires a "process_date" that we will use for partitioning. (We will actually not do anything with the date in the pipeline, but partitioning by date is a very common partitioning strategy.)

Until now, our job configurations have been static. In week two, we passed in a dictionary of the configs needed to initialize a job from it. After adding the "process_date" config, the code would now look something like this:

```
local = {"ops": {"create_table": {"config": {"table_name": "fake_table",
"process_date": "2020-07-01"}}}}

etl_local = etl.to_job(
    name="etl_local",
    config=local,
    resource_defs={"database": ResourceDefinition.mock_resource()},
    tags={"Demo": True},
)
```

But partitioning is more dynamic! If we wanted the "process_date" to represent every day from the beginning of July, rather tham being a static value, we could use a Dagster partitioning strategy:

```
@daily_partitioned_config(start_date=datetime(2022, 7, 1))
def local_config(start: datetime, _end: datetime):
    return {
        "ops": {"create_table": {"config": {"table_name": "fake_table",
"process_date": start.strftime("%Y-%m-%d")}}}
    }
```

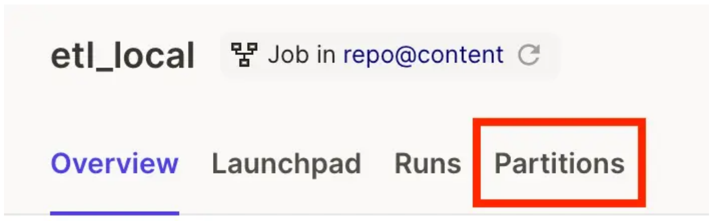
Instead of our static `local` dictionary, we are decorating a function with the Dagster provided `daily_partitioned_config`. (Dagster provides prebuilt partitioning configs for common patterns around dates.) Let's take a look at what this is doing. This function returns a dictionary, and the dictionary looks very similar to our `local` dictionary from before. However, we are replacing the set "process_date" with the dynamic element `start`. This element represents all dates from July 1, 2022 on. We can now swap in `local_config` for `local` in the job definition:

```
etl_local = etl.to_job(
    name="etl_local",
    config=local_config,
    resource_defs={"database": ResourceDefinition.mock_resource()},
    tags={"Demo": True},
)
```

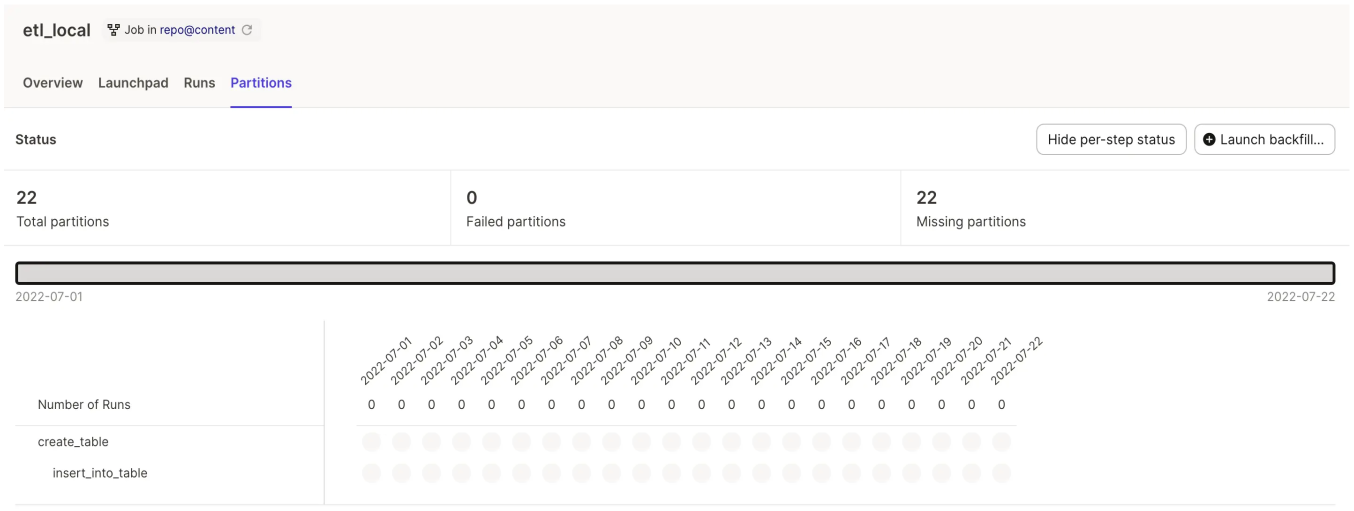
Until we look in Dagit, launching a job with a partitioning config set will not seem very different from what we've done before.

Partitions in Dagit

Let's see our `etl_local` pipeline in Dagit. When we look at the pipeline, you will see a new tab is present for "Partitions."



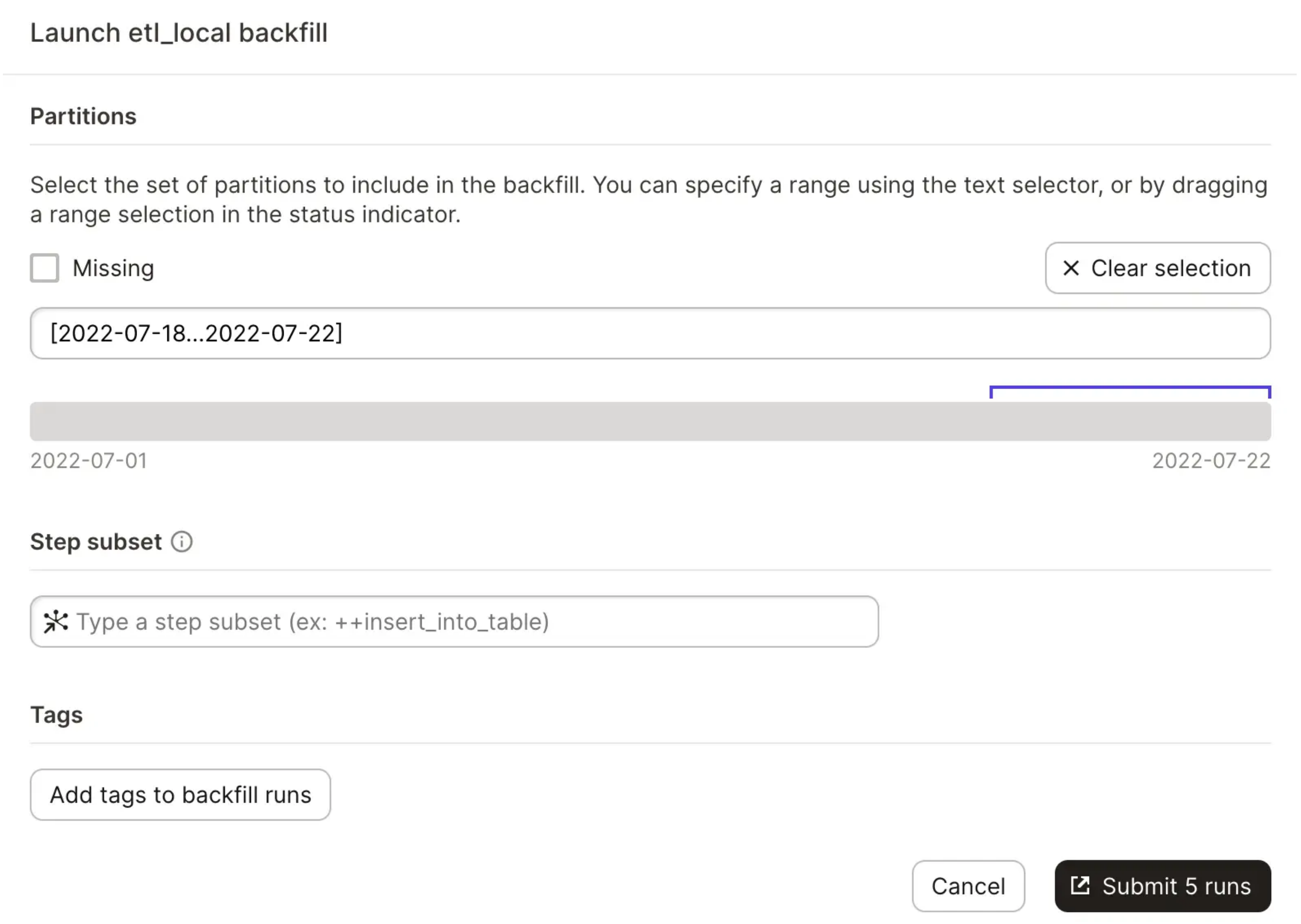
If we look in this tab, we will see our partitioning strategy laid out visually:



	2022-07-01	2022-07-02	2022-07-03	2022-07-04	2022-07-05	2022-07-06	2022-07-07	2022-07-08	2022-07-09	2022-07-10	2022-07-11	2022-07-12	2022-07-13	2022-07-14	2022-07-15	2022-07-16	2022-07-17	2022-07-18	2022-07-19	2022-07-20	2022-07-21	2022-07-22
Number of Runs	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

We see that there are 22 partitions for our job. Each partition represents a single day, from our set start date of 2022-07-01 to the current date, as of writing, 2022-07-22. Each dot represents a day of the partition for each op in our pipeline. They are all gray, meaning we have not run our job for these partitions.

Usually this is where data engineers would write some script to execute the missing days. But Dagster allows us to easily backfill missing partitions! If you click "Launch Backfill" you will be able to set a custom backfill to perform. You can click "Missing" to backfill any partitions that have yet to be performed or partitions within a set range. We will set the range from 2022-07-18 to 2022-07-22.



Launch etl_local backfill

Partitions

Select the set of partitions to include in the backfill. You can specify a range using the text selector, or by dragging a range selection in the status indicator.

Missing

Clear selection

[2022-07-18...2022-07-22]

2022-07-01 2022-07-22

Step subset

Type a step subset (ex: ++insert_into_table)

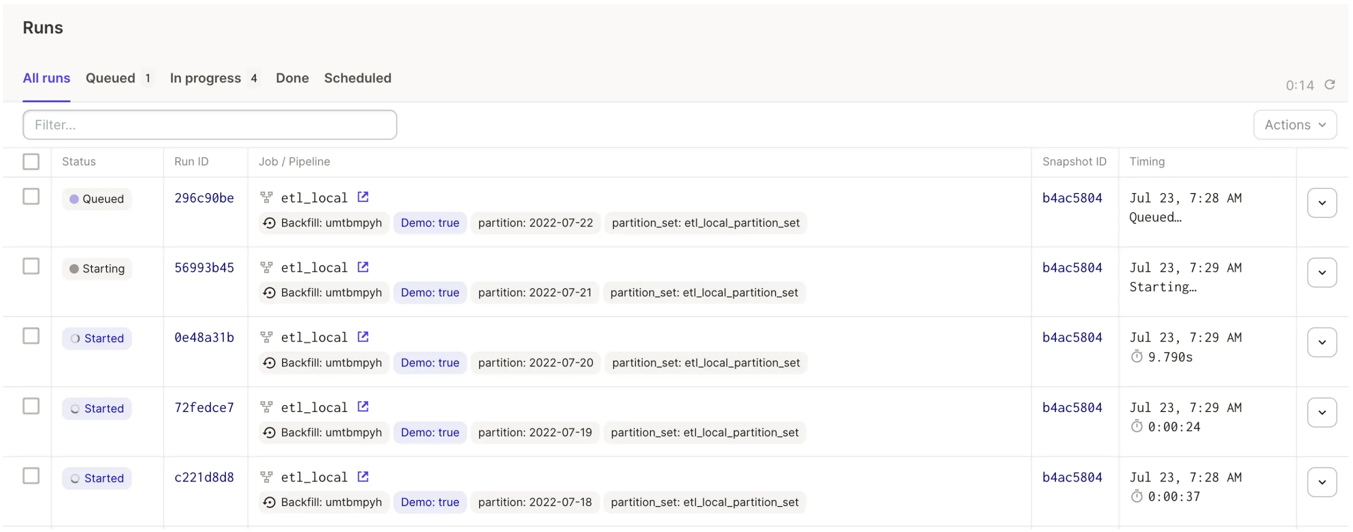
Tags

Add tags to backfill runs

Cancel

Submit 5 runs

After we submit our backfill, Dagster will launch runs for the requested partitions:



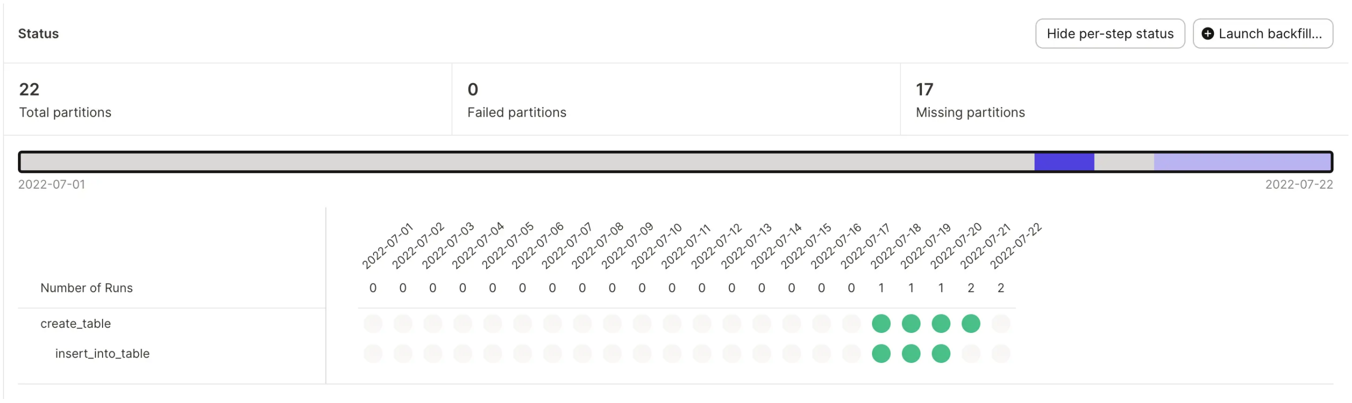
Runs

All runs Queued 1 In progress 4 Done Scheduled 0:14

Filter... Actions

Status	Run ID	Job / Pipeline	Snapshot ID	Timing
Queued	296c90be	etl_local Backfill: umtbmpyh Demo: true partition: 2022-07-22 partition_set: etl_local_partition_set	b4ac5804	Jul 23, 7:28 AM Queued...
Starting	56993b45	etl_local Backfill: umtbmpyh Demo: true partition: 2022-07-21 partition_set: etl_local_partition_set	b4ac5804	Jul 23, 7:29 AM Starting...
Started	0e48a31b	etl_local Backfill: umtbmpyh Demo: true partition: 2022-07-20 partition_set: etl_local_partition_set	b4ac5804	Jul 23, 7:29 AM 9.790s
Started	72fedce7	etl_local Backfill: umtbmpyh Demo: true partition: 2022-07-19 partition_set: etl_local_partition_set	b4ac5804	Jul 23, 7:29 AM 0:00:24
Started	c221d8d8	etl_local Backfill: umtbmpyh Demo: true partition: 2022-07-18 partition_set: etl_local_partition_set	b4ac5804	Jul 23, 7:28 AM 0:00:37

As the jobs finish, you can start to see our partitions being marked as a success in the "Partitions" tab:



Status

22	0	17
Total partitions	Failed partitions	Missing partitions

2022-07-01 2022-07-22

	2022-07-01	2022-07-02	2022-07-03	2022-07-04	2022-07-05	2022-07-06	2022-07-07	2022-07-08	2022-07-09	2022-07-10	2022-07-11	2022-07-12	2022-07-13	2022-07-14	2022-07-15	2022-07-16	2022-07-17	2022-07-18	2022-07-19	2022-07-20	2022-07-21	2022-07-22
Number of Runs	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	2	2
create_table	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	●	●	●	○
insert_into_table	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	●	●	○	○

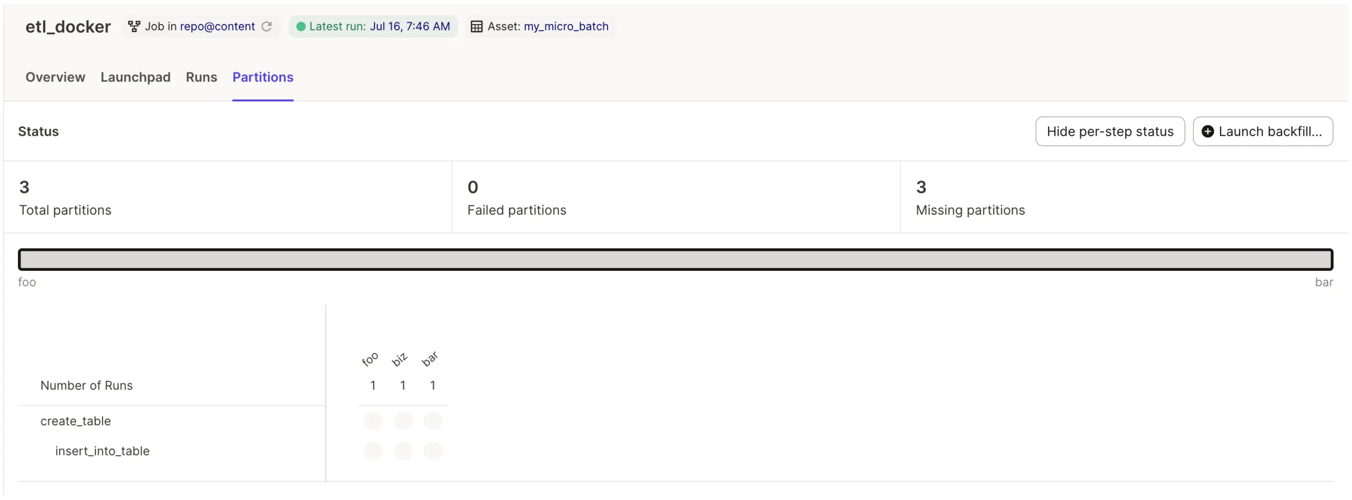
Static Partitions

Dates are not the only way to set our partitions. We will use the same ETL graph, but for our Docker configuration of the job. Here, we will partition based on table name. Say our Docker ETL pipeline is responsible for populating three different tables "foo", "biz" and "bar." We can decorate a function similar to what we did with the `local_config`, but instead we will use `static_partitioned_config` now:

```
@static_partitioned_config(partition_keys=["foo", "biz", "bar"])
def docker_config(partition_key: str):
    return {
        "resources": {
            "database": {
                "config": {
                    "host": "postgresql",
                    "user": "postgres_user",
                    "password": "postgres_password",
                    "database": "postgres_db",
                }
            }
        },
        "ops": {"create_table": {"config": {"table_name": partition_key,
"process_date": "2020-07-01"}}},
    }
```

Like before, we are providing the config necessary to run the pipeline. (Remember that the Docker configuration requires the `resources` to be set, since we are connecting to a database instance. This differs from local execution, which just uses mocks.) But now, our static partition is providing our `table_name`.

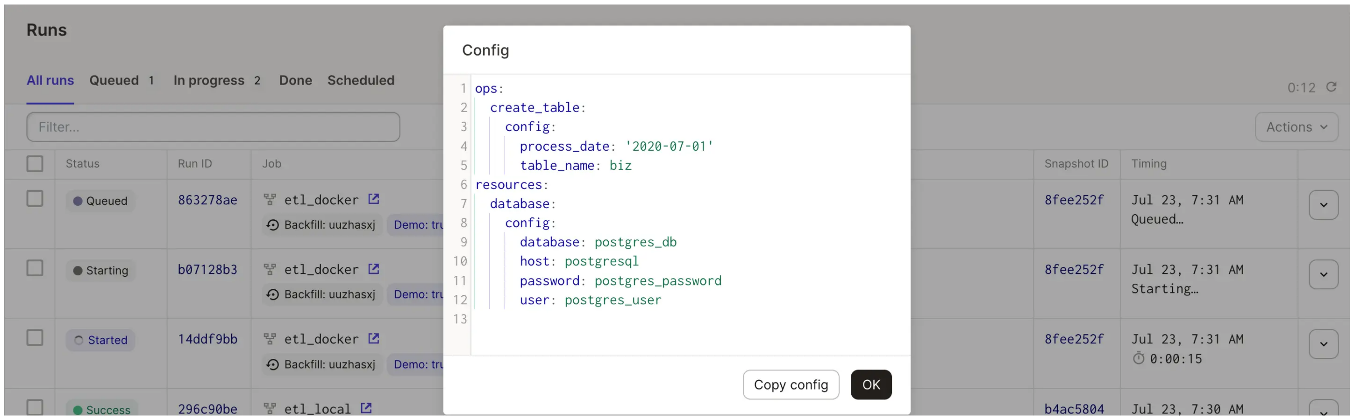
If we look in Dagit, we will see our partitions are now our three table names, rather than dates:



3 Total partitions	0 Failed partitions	3 Missing partitions

	foo	biz	bar
Number of Runs	1	1	1
create_table			
insert_into_table			

Everything else remains the same. We can perform a backfill based on the partition (our table name). And if we look at the config of the jobs running, we will see that our ETL pipeline will create and insert rows into the corresponding tables.



```
ops:
  create_table:
    config:
      process_date: '2020-07-01'
      table_name: biz
resources:
  database:
    config:
      database: postgres_db
      host: postgresql
      password: postgres_password
      user: postgres_user
```

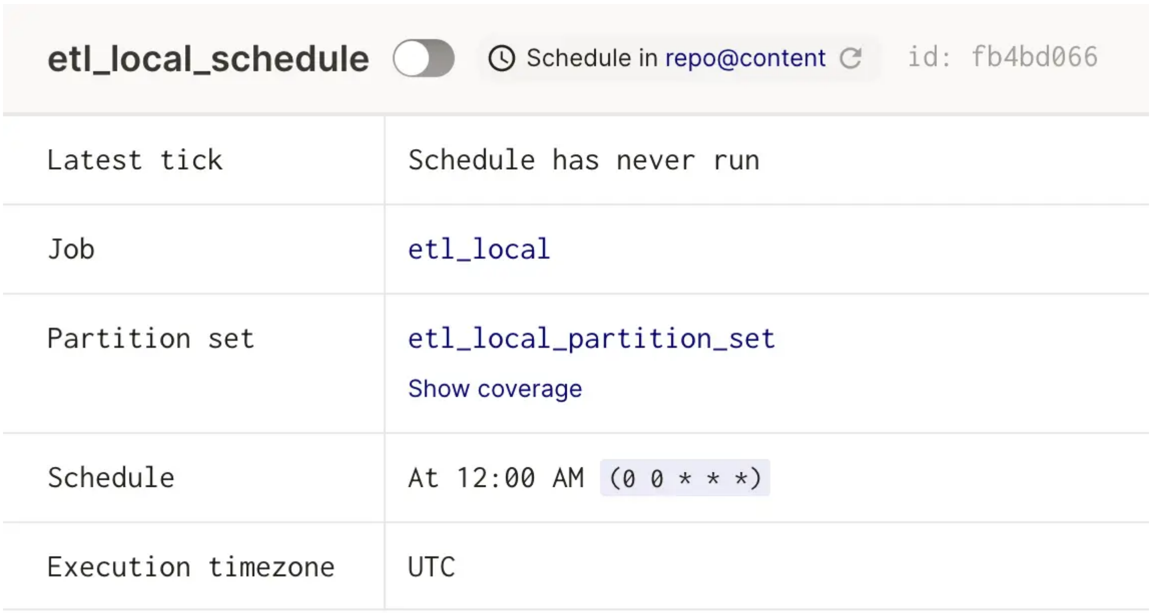
Status	Run ID	Job	Snapshot ID	Timing
Queued	863278ae	etl_docker Backfill: uuzhasxj Demo: tr	8fee252f	Jul 23, 7:31 AM Queued...
Starting	b07128b3	etl_docker Backfill: uuzhasxj Demo: tr	8fee252f	Jul 23, 7:31 AM Starting...
Started	14ddf9bb	etl_docker Backfill: uuzhasxj Demo: tr	8fee252f	Jul 23, 7:31 AM 0:00:15
Success	296c90be	etl_local	b4ac5804	Jul 23, 7:30 AM

Schedules and Partitions

Schedules are the last topic we'll cover while discussing partitions. We have already talked about creating schedules with cron syntax. If your job is configured with a date partition, you can easily generate a schedule for it on its partition by using `build_schedule_from_partitioned_job`:

```
etl_local_partitioned_schedule =
build_schedule_from_partitioned_job(etl_local)
```

This schedule will run daily and update your partitions with each execution.


etl_local_schedule Schedule in repo@content id: fb4bd066

Latest tick	Schedule has never run
Job	etl_local
Partition set	etl_local_partition_set Show coverage
Schedule	At 12:00 AM (0 0 * * *)
Execution timezone	UTC

Summary

QUICK QUIZ (SINGLE SELECTION)

You receive files from a data vendor that arrive in S3 in the following format:

- bucket/data_vendor/2022/01/data_1.csv
- bucket/data_vendor/2022/01/data_2.csv
- bucket/data_vendor/2022/02/data_1.csv
- bucket/data_vendor/2022/02/data_2.csv
- ...

What might be a good partition decorator to use?

- ◯ static_partitioned_config
- ◯ daily_partitioned_config
- ◯ hourly_partitioned_config

○ monthly_partitioned_config

Submit → Clear

Partitions are another common practice in data and can help make your jobs production-ready. Before building a pipeline, it can be useful to think of how its underlying data is organized and how your jobs can be constructed to match that.

Object	Relationship	Description
partition	A job can have a single partition strategy configured.	A partitioned job organizes its runs and run history by a set data pattern (either time-based or by a common characteristic).