Data Engineering Design Patterns

Bartosz Konieczny

Published by O'Reilly Media, Inc.

Chapter 6. Data Flow Design Patterns

Generating business value from raw data enables a fact-based decision process, and the data value design patterns from Chapter 5 will help you create this smart process. However, at this stage of our exploration of data engineering design patterns, the generated data insight remains local to you. It is indeed beneficial, but what if I tell you that you can create even more benefits by opening it up to a much wider scale than just local?

For example, you might expose one of your valuable datasets to other teams within the organization to enable them to enrich their local use cases and consequently increase their data value assets. It works the opposite way too, as other teams could share their valuable datasets that would increase the value of your data! Although this sounds like a data value patterns family, there's a different set of rules to apply. That's why you'll retrieve them as data flow design patterns.

The goal of data flow design patterns is to design and coordinate all steps required to generate a dataset. This involves actions like chaining various tasks in a pipeline, creating parallel or exclusive execution branches, or even managing the dependency of physically separated pipelines.

Data flow design patterns operate at two different levels. The first level is data orchestration, where they work in one or many data pipelines. This is particularly useful when you want to address the cross-teams collaboration issue. The second level is the data processing layer, which is the environment of your job. Here, data flow design patterns help to better organize business logic to make it more obvious and easier to maintain over time.

How can you achieve all that with data flow design patterns? First, with sequence patterns, you can coordinate tasks or pipelines within a single pipeline or across many pipelines. In the next section, you'll see how to enhance pipelines with two other types of patterns. The first of them, fan-in, handles the merge situation where one step depends on many others. The second category does the opposite, meaning it creates two or more branches from the same task. Finally, you will discover orchestration patterns that you can leverage to manage the concurrency of your pipelines.

Curious to see the patterns? Let's get started!

Sequence

The first category of patterns that you'll inevitably encounter while designing data flows concerns the sequence of steps. This is an important factor that will impact the complexity,

performance, and maintenance of the pipelines. An example here is a data processing job writing the processed dataset to multiple places. If you represent it as a single unit in your workflow, whenever you need to replay only the loading part for one of the databases, you'll have to restart the whole execution. The patterns from this section will show you how to address this issue.

Pattern: Local Sequencer

The first design pattern from this section is probably the easiest one as it orchestrates tasks locally (i.e., within the same pipeline or data processing job).

Problem

You're in charge of one of the oldest jobs in your data analytics department. Over the years, the codebase has grown from dozens to hundreds of lines, and the number of transformations has increased three times. The job itself fails very often, and each time, it must start again from the beginning, leading to long debugging journeys.

You were tasked with simplifying the code and improving daily maintenance. Unfortunately, you can't remove any business logic.

Solution

A good software engineering practice to simplify complex logic consists of decomposing it into smaller and therefore more approachable steps. This is also valid for data engineering, where reorganization improves readability and also highlights the separation of concerns. That's where the Local Sequencer pattern shines.

The end goal is to decouple one big component into multiple smaller but connected items that will be run sequentially (i.e., one after another). The dependency between tasks should be organized according to the dataset dependencies. For example, if task B needs the data generated by task A, then the sequence should be defined as B executing after A completes. To help you understand this better, let's think about a full data ingestion and the patterns covered in Chapter 2.

To implement the ingestion, you'll need to create two tasks: one implementing the Readiness Marker pattern to determine whether the data is there and another using the Full Loader pattern to physically load the data. (We discussed both patterns in Chapter 2.) As you can see in Figure 6-1, you can implement them either as dependent tasks (with data orchestration layer dependency) or as a single task (with data processing layer dependency).

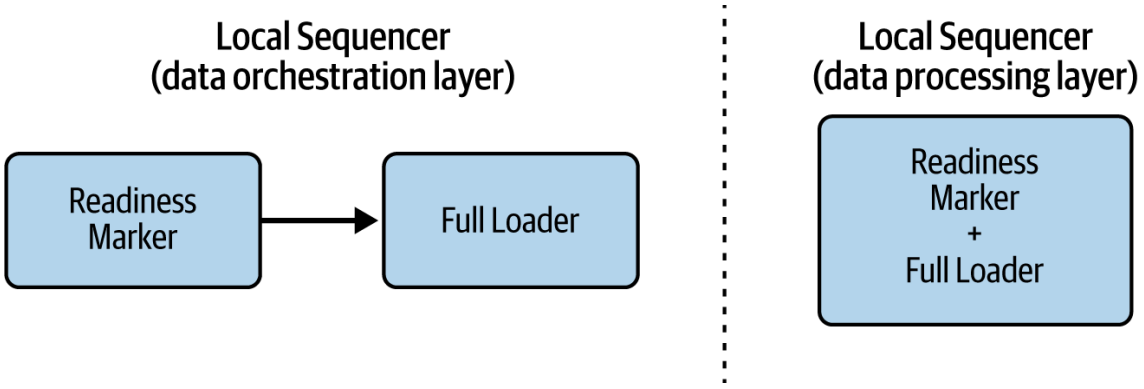

Figure 6-1. Local Sequencer versus single task

The following three criteria should help you decide whether the sequence should be based on the data orchestration layer or the data processing layer:

Separation of concerns

Putting all operations in the same item can make things harder to understand. A good indicator here is naming difficulty. If you struggle to find the name or if the name is too long in your opinion, this may be an indicator that you put too many operations into the single task on the data processing layer.

Maintainability

Relying on data processing sequentiality is also challenging for maintenance. In cases of backfilling or automatic retries, you will recompute all successful tasks prior to the failed one. For example, if your readiness check involves calling 10 paid APIs and you need to restart a job that failed three times, you'll quickly see the cost increase if you include the API call to your job.

Implementation effort

The data orchestrator may provide different abstractions to perform common tasks out of the box, such as running a SQL query or executing an API call. If you combine all tasks into a single unit, you won't be able to leverage this facility and you'll probably need to reinvent the wheel.

Consequences

Even though the tips we've presented thus far can be helpful, they may not guarantee a perfect organization from day one. For that reason, defining boundaries remains a challenge, among many others presented in the following sections.

Boundaries

If you define boundaries incorrectly, execution time may grow too much or even impact other pipelines if you can't scale the scheduler in your data orchestration logic. It's therefore important to find a good balance between the scope and the number of tasks.

A good rule of thumb that applies to both the data processing layer and the data orchestration layer is to think about restart boundaries (i.e., what are the tasks that should be able to restart individually?). In our data ingestion example, the Readiness Marker and Full Loader tasks should fail individually and shouldn't impact each other. Put differently, the Readiness Marker shouldn't run if the loading step fails.[1] The loading execution already implies the presence of data.

Regarding the data processing job, you'll often put the boundary between the most compute-expensive operations. For example, if generating an intermediary dataset before transforming it into the final one is costly, you might put the separation line there in your job. Thanks to this, whenever your final transformation fails, you won't need to execute this costly intermediate dataset generation at each retry.

Besides the restart boundaries, you can also reason for the logic separation in terms of transactions. If two or more operations must be performed as a single unit, it makes sense

to keep them together. An example of thisis the data processing job implementing the Dynamic Late Data Integrator pattern (code available on GitHub), which, besides processing the data, also updates the last processed version in the state table. The processing logic and the processed version storage depend on each other, so it makes sense to execute them as a single unit.

Examples

Giving an example of the Local Sequencer is relatively easy. Both the libraries for data orchestration and the data processing layers used in this book provide native abstraction to chain tasks. Let's start with Apache Airflow, meaning the orchestration part. Combining tasks consists of using the >> sign to express the dependency because the left side must run before the right one. The code in Example 6-1 shows this.

Example 6-1. Local Sequencer in Apache Airflow

```
input_data_sensor >> load_data_to_table >> expose_new_table
```

If you needed to translate the snippet from Example 6-1 into a human language, you could say, "The pipeline starts by waiting for the data to be available. Once this condition is met, the next task loads the input data into an internal table. Later, the last task exposes the internal table to the end users." But I hope you agree with me that Airflow's language is more concise!

The same concise implementation is available in other orchestrators, including the cloud ones. If you use AWS EMR to run your data procesisng jobs, you can use the Step API to add tasks to run one after another. Example 6-2 shows the last two tasks from Example 6-1 implemented as sequential steps of the cluster.

Example 6-2. Local Sequencer with AWS EMR

```
aws emr add-steps --cluster-id j=cluster_id --steps Type=Spark,Name="Spark Program",

 ActionOnFailure=TERMINATE_CLUSTER,Args=[--class com.waitingforcode.DataLoader]

aws emr add-steps --cluster-id j=cluster_id --steps Type=Spark,Name="Spark Program",

 ActionOnFailure=TERMINATE_CLUSTER,Args=[--class
com.waitingforcode.DataPublisher]
```

An important thing here is the dependency configuration on failure. As you can see, the code asks to terminate the cluster in case of any task failure. Otherwise, you could introduce some inconsistency by exposing partial data and considering the job to be successful. Such configuration is not required for Apache Airflow, which by default enables sequential processing based on the success of the upstream job (i.e., one task can't advance as long as its parent didn't succeed).

Besides the data orchestration, the pattern also works at the data processing level. You can express it in SQL or a programmatic API, like Python for PySpark. The idea here is to consider each previously defined variable as an input for the next step, until you reach the data writing stage. Example 6-3 shows that while there is no explicit chaining argument, the usage remains intuitive.

Example 6-3. Local Sequencer with PySpark and programmatic API

input_dataset: DataFrame = spark_session.read...

valid_and_enriched_dataset_to_write: DataFrame = input_dataset..

valid_and_enriched_dataset_to_write.write...

More Than CRON

The Local Sequencer pattern shows the power a data orchestrator has compared to a simple CRON expression when it comes to building advanced processing logic. However, a CRON expression can still be a valid solution if you have an isolated use case that doesn't require any dependencies.

Pattern: Isolated Sequencer

Often, the pipelines implementing the Local Sequencer are not the final steps. Instead, they are part of a more complex workflow where multiple isolated workflows must collaborate to generate the final insight. Here, although the rules may sound similar, there are completely different triggering conditions and constraints.

Problem

Your team is responsible for cleaning and enriching raw datasets to expose them as various views used in a data visualization tool. After the technical meeting with the data visualization team, you agreed that it's not a good idea to include the dashboards dataset transformation directly in your data preparation pipeline, as your team won't be responsible for that part. Instead, the data visualization team asked you to provide the cleansed and enriched dataset only. The data visualization team will handle the transformation on its own.

Solution

The problem statement introduces two pipelines where one provides data to another. However, due to organizational separation, it's not possible to merge them into a single process. That's the perfect scenario in which to use the Isolated Sequencer pattern.

The objective here is to find a way to combine physically isolated pipelines. As with the Local Sequencer, the most important concern is the identification of boundaries. The easiest solution consists of dividing the pipeline in terms of consumers and providers, or teams. If your team provides the dataset to a different team, then naturally, you can draw a boundary to create two isolated pipelines.

On the other hand, you may also face a situation in which you are the provider and a consumer at the same time. This may happen when the processed dataset is used by other pipelines within your team's scope to generate other datasets. To define boundaries in that context, you can analyze the complexity of the pipeline. If combining all producer and consumer logic in a single place makes things unreadable, you should consider splitting it into multiple pipelines following the producer and consumer approach. Otherwise, you might prefer to keep them together and thus use the Local Sequencer pattern.

Defining boundaries is only the first step, though. The second step consists of finding the triggering mechanism. There are two strategies here: data based and task based.

The data-based strategy is based on the Readiness Marker pattern from Chapter 2. In other words, the data producer generates the dataset and a marker file to indicate it's ready for processing. The consumer listens for this marker file and starts the work as soon as it detects the creation.

In the task-based strategy, the data producer doesn't create a marker file. Instead, it directly triggers the pipeline responsible for consuming the generated dataset. As you can see, the two approaches have different couplings and shared responsibilities:

- Pipelines from the data-based solution are loosely coupled, meaning there is more room for evolution. The only requirement is to respect the marker file. On the other hand, the pipelines that form the task-based solution are tightly coupled. This means that, albeit physically separated, they can't live alone. This is particularly visible on the data producer side because a simple task or pipeline renaming operation on the consumer side will make the whole dependency chain fail, since the producer will try to trigger a task that doesn't exist anymore.
- This brings up another topic: which side can change? In the data-based approach, the dataset consumer has more freedom. It can even decide to use a different dataset without notifying the data provider. On the other hand, in the task-based strategy, the consumer can't simply decide to skip the dataset as the producer has the direct trigger mechanism. If the dataset is removed, the producer's pipeline can fail.

Figure 6-2 summarizes both approaches.

As you'll notice, the data-based dependency approach relies on the presence of the dataset generated by the data provider pipeline. The consumer leverages the Readiness Marker pattern to start its pipeline as soon as the data is there. On the other hand, the task-based dependency approach shown on the bottom considers the dataset to be a local asset of the data provider. In that case, the provider directly triggers processing on the data consumer side with a dedicated task. That can be a good approach if there is no shared dataset (for example, when the consumer must send some notification to the users instead of working on the provider's data).

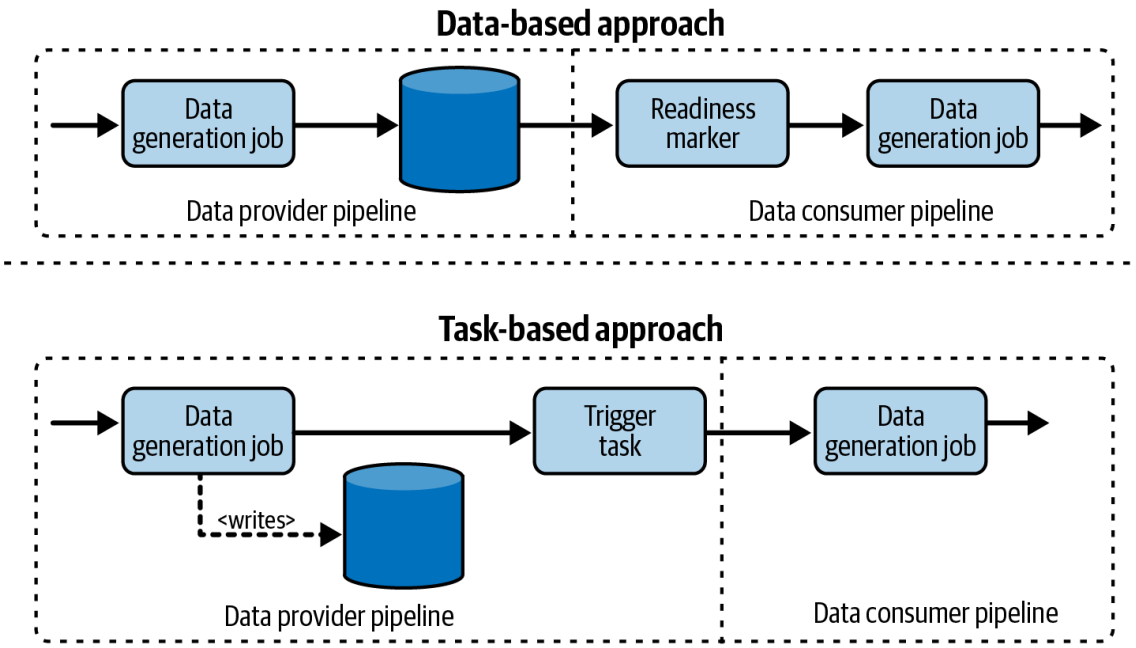

Figure 6-2. Two strategies to implement the dependencies between two pipelines

Consequences

As you've already learned in the Solution section, the biggest challenge is to keep all the pipelines in sync. Plus, the biggest drawback of the pattern is that by adding some logical isolation, it also adds an extra operational constraint.

Scheduling

The task-based solution doesn't impact just the evolution part. It also involves something important, which is the scheduling frequency. Ideally, the two pipelines should share the same schedule so that the producer can directly trigger the consumer. If that's not the case, one of them will need to introduce more complexity.

If it's the producer, it will need to add a condition to skip the triggering part for some of the planned execution schedules. If it's the consumer, it'll have to do the same but by adapting its schedule to the producer and running the physical data processing only when needed.

Communication

The Isolated Sequencer addresses pipelines managed by different teams, among other things. That may be a problem if the organization doesn't have a good communication culture.

And it's not easy to overcome this issue. Either as a producer or as a consumer, you can add a mechanism that checks if the condition on the other side didn't change. That requires a lot of effort and may even be impossible to implement if the other side doesn't accept any incoming connections for security reasons!

A more resilient approach requires more effort and time. It involves an organizational change to make communication efficient. Each isolated team must be aware of its inputs and outputs, so that it can effectively communicate with all dependent parties planning work, particularly the one introducing any breaking changes. You may be the catalyst of this evolution, but the implementation goes far beyond the technical aspect of your work.

Examples

But let's remain in the code and see now how to implement the data-based and trigger-based solutions with Apache Airflow. The data-based approach relies on the same waiter components (sensors) as the Readiness Marker pattern. The first example from this section shows two dataset-dependent pipelines.

Example 6-4 demonstrates an implicit dependency between the devices_loader and devices_aggregator pipelines. The input_data_sensor from the devices_aggregator is a kind of dependency enforcer. If it fails, this means the data provider couldn't deliver the data as agreed before. However, there is no way to know whether the two pipelines are interdependent.

Example 6-4. Dataset dependency for the Isolated Sequencer

```
# devices_loader

@task

def load_new_devices_to_internal_storage():

 ctx = get_current_context()

 partitioned_dir = f'{devices_file_location}/{ctx["ds_nodash"]}'

 internal_file_location = f'{partitioned_dir}/dataset.csv'

 shutil.copyfile(input_devices_file, internal_file_location)


input_data_sensor >> load_new_devices_to_internal_storage()


# devices_aggregator

input_data_sensor = FileSensor(

 task_id='input_data_sensor',

 filepath=devices_file_location + '/{{ ds_nodash }}/dataset.csv',

)

input_data_sensor >> load_data_to_table >> refresh_aggregates
```

Data Lineage

An important component in complex data system is the data lineage tool you use to represent dependencies between datasets. In our example, you could use it to see what datasets or pipelines are consuming the devices_loader's output. Chapter 10 covers two data lineage design patterns. You can also learn more about the lineage from OpenLineage,[2] which is an open source standard supporting major data tools used nowadays.

Implicitness is not the case for the next, trigger-based example. Example 6-5 shows two dependent pipelines that

use ExternalTaskMarker and ExternalTaskSensor operators, respectively, instead of data dependencies. The sensor is there to represent the consumer pipeline waiting for the data provider's task execution. The marker automates backfilling, and the marker task detects any backfilling made on the devices_loader pipeline and automatically runs it locally.

Example 6-5. Trigger-based dependency for the Isolated Sequencer

```
# devices_loader
success_execution_marker = ExternalTaskMarker(
 task_id='trigger_downstream_consumers',
 external_dag_id='devices_aggregator',
 external_task_id='downstream_trigger_sensor',
)
(input_data_sensor >> load_new_devices_to_internal_storage()
  >> success_execution_marker)


# devices_aggregator
parent_dag_sensor = ExternalTaskSensor(
 task_id='downstream_trigger_sensor',
 external_dag_id='devices_loader',
 external_task_id='trigger_downstream_consumers',
 allowed_states=['success'],
 failed_states=['failed', 'skipped']
)
parent_dag_sensor >> load_data_to_table >> refresh_aggregates
```

Fan-In

The two previous patterns involve a sequence, meaning steps that follow each other in a particular order. But data pipelines are not that simple every time. Often, they create branches that eventually merge again at some point. This point is where the fan-in patterns family gets involved.

Pattern: Aligned Fan-In

The easiest fan-in pattern to explain is the Aligned Fan-In. It simply assumes that all direct parent tasks must succeed before continuing.

Problem

Your pipeline generates a daily aggregate of your blog visits from raw visit events. However, the dataset is partitioned by the hour as that organizational logic fits most of the use cases within your organization.

It doesn't make sense to process the pipeline daily on your own as the consumers of your dataset are interested in the full view only. However, you would like to leverage the hourly partitioning to avoid too much data being processed in a single job.

Solution

The use case introduced previously involves a dependency between multiple parent tasks and one task that runs after them. To solve this dependency at either the data orchestration layer or the data processing layer, you can use the Aligned Fan-In pattern.

The implementation is rather straightforward for the data orchestration. It involves defining separate branches that merge into a common task. In our example, these branches represent data processing jobs generating partial aggregates for each hour of the day. Once all are completed, the merge task takes these hourly results to compute the final output for each day. As a result, your orchestration layer generates a flow like the one presented in Figure 6-3.

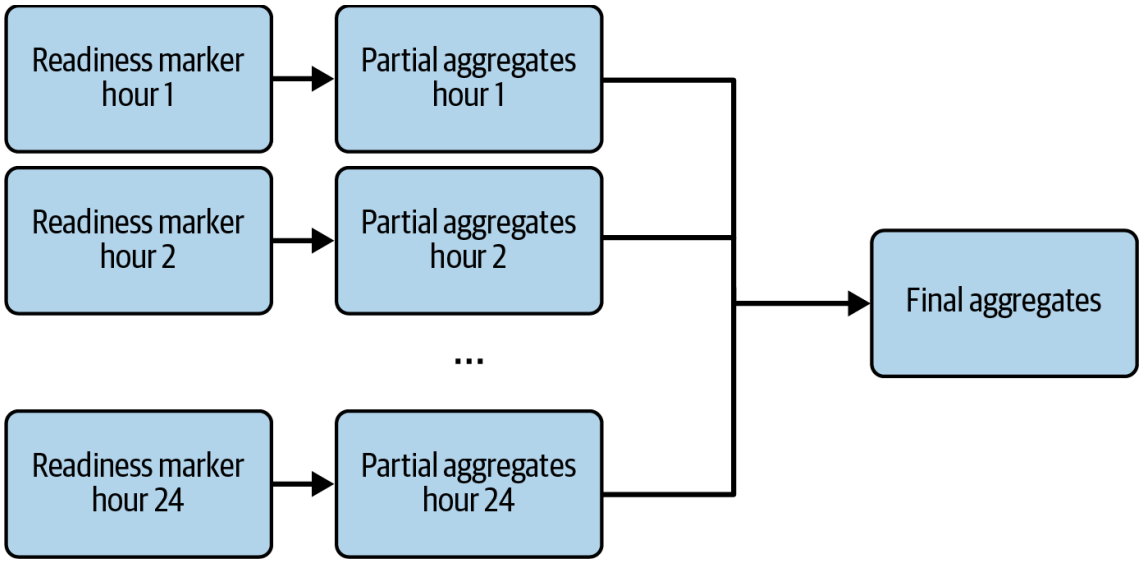

Figure 6-3. Aligned Fan-In in the example of partial aggregates

However, things are not that simple when it comes to the data processing layer. Here, you still have to define the branches, but additionally, you must define how they interact with each other. The best way to understand this is to use SQL operations.

The first interaction outcome will be a single data processing abstraction produced by multiple outputs. In this context, you will use the UNION operator and get a dataset with more rows but the same number of columns. An alternative is the JOIN operation, which will combine the datasets row-wise, thus adding extra columns. The JOIN approach will always store fewer rows than the UNION-based solution. You can think of them as vertical (UNION) and horizontal (JOIN) alignments.

A big advantage of the Aligned Fan-In pattern is the feedback loop optimization. Let's take a look at our partial aggregates schema from Figure 6-3 to understand this better. If you don't leverage the pattern and there is an issue in the processing for the last hour, the pipeline will

return an error, probably by the end of the run since it's processing one day of data. On the other hand, with decoupled logic, you might have 24 smaller jobs that would work on smaller volumes of data simultaneously and as a result return the issue faster.

Besides this feedback loop, there is another advantage related to failure. If only the last of the 24 hours fails, as is the case in our example, you only need to fix and replay this hour. Therefore, without the Aligned Fan-In pattern and with a single big data processing task for the 24 hours, you would need to process the other 23 hours too. Also, as you've probably noticed, in this pattern, you'll use the separation logic based on the ease of backfilling, as was the case with the Local Sequencer pattern.

Consequences

By now, we've probably convinced you that decoupling can be beneficial for your pipelines. However, despite this positive impact, the Aligned Fan-In pattern has some gotchas.

Infrastructure spikes

One drawback of the Aligned Fan-In pattern is the infrastructure load. As you can see in the problem statement, our daily aggregation job will run 24 jobs simultaneously. That shouldn't be an issue if you have elastic provisioning capacity. If you don't, you'll need to find a good balance and reduce the allowed concurrent runs to some smaller number.

An alternative approach to reducing the concurrency is to run branches as soon as possible, in a fully incremental way. For our 24 hourly branches, this would mean running each branch hourly and aggregating the results only in the last run of the day.

Scheduling skew

Since the child tasks require all parents to be successfully executed, having unbalanced execution time for the parents will lead to a scheduling skew. When that happens, all the successful tasks will wait for the slowest one. As a result, the child task's triggering time depends on the longest parent tasks.

Scheduling overhead

Longer parent tasks aren't the only things that can be problematic. You must also be aware that pipelines that are too granular also involve scheduling overhead. The data orchestrator will need to allocate most of their resources to schedule and coordinate tasks. However, that's the price you must pay if you want to decouple and improve readability.

Complexity

The more you decouple, the longer the pipeline you will create. This may lead to some negative effects like reduced readability and understanding. Unfortunately, there is no one-size-fits-all solution, so you must identify correct boundaries for each pipeline individually.

Questions this chapter raises that may help you are "What tasks belong to a single unit of execution?" and "What operations should be backfilled individually?"

Examples

What about examples? First, let's see how to implement the Aligned Fan-In with Apache Airflow, meaning at the data orchestration layer. The implementation leverages the dynamic character of pipeline creation. As you can see, Example 6-6 defines the pipeline sequence

only once. However, the definition is inside a for loop. Apache Airflow will interpret this declaration, create a dedicated task for each iteration, and connect them to the common clear_context parent task.

Example 6-6. Aligned Fan-In in Apache Airflow

```
clear_context = PostgresOperator(...)

generate_trends = PostgresOperator(...)


for hour_to_load in [f"{hour:02d}" for hour in range(24)]:

 file_sensor = FileSensor(

  task_id=f'wait_for_{hour_to_load}',

  filepath=input_dir +'/date={{ ds_nodash }}/hour=' + hour_to_load+'/dataset.csv'

)

 visits_loader = PostgresOperator(

  task_id=f'load_hourly_visits_{hour_to_load}',

  params={'hour': hour_to_load}

 )


 clear_context >> file_sensor >> visits_loader >> generate_trends
```

If you have a static list of tasks, you can also declare all tasks located at the same level with "[...]," (for example, [load_data_1, load_data_2] >> process_data). This is more verbose but also more readable as the connection between branches and the common child task is explicit.

In addition to the data orchestration level, you can implement the pattern at the data processing level. A great operation representing the Aligned Fan-In is UNION. As its name indicates, it combines two separate datasets into a single one so that you can apply one transformation on top of it. Example 6-7 shows an example for PySpark.

Example 6-7. UNION **in PySpark**

```
input_dataset_1: DataFrame = ...

input_dataset_2: DataFrame = ...

output_dataset = input_dataset_1.unionByName(input_dataset_2)
```

PySpark has an interesting solution for dealing with one of the trickiest parts of the UNION operation. By default, UNION is position based, meaning that you can combine incompatible fields, though this will result in an error (see Example 6-8).

Example 6-8. Position-based error

```
SELECT a, b, c FROM abc
```

UNION

SELECT c, b, a FROM cba

The operation in Example 6-8 will result in an incompatible dataset with a + c, b + b, and c + a combined. To mitigate this issue, Apache Spark provides a unionByName method that combines the datasets by column name rather than by position. It's not that popular, though, so be aware of it when you use the default UNION implementation in SQL or other data processing tools.

Pattern: Unaligned Fan-In

Sometimes, having a condition that all the parents must succeed can not only add some latency but can also be semantically wrong. Therefore, you may need a variation on the Aligned Fan-In pattern.

Problem

Your hourly-based processing of visits aggregates with the Aligned Fan-In pattern has been working great for several weeks. However, the implementation lacks a proper method for managing failed tasks.

A few times, an hour hasn't been correctly processed, and you therefore haven't produced the aggregated views for your downstream consumers. After a meeting, you agreed that it would be better to release even a partial dataset and fill the gaps later. You're looking now for a solution that will help you evolve the pipeline based on the Aligned Fan-In pattern.

Solution

The solution consists of relaxing the dependency on the parents' outcome and transforming the Aligned Fan-In pattern into the Unaligned Fan-In pattern.

With this pattern, a child task can run even when when some of the parents don't succeed. This enables different scenarios:

- If some parents succeed and the remaining failures are acceptable, you can trigger the child task anyway. Consequently, you may be working on a partial input that will lead to the partial dataset problem, as in the problem statement.
- If all the parents fail, instead of running the task based on the success criteria, you can schedule a task based on the failures criteria. For example, the task could be a fallback or error management task.

To implement the pattern, you'll need to configure the trigger conditions that are available in your data orchestration tool. In Apache Airflow, you'll rely on the trigger_condition attribute and set it to one of the available states. In less declarative environments, you may need to explicitly evaluate the parent tasks' outcomes.

Consequences

Although the Unaligned Fan-In pattern shares the drawbacks of the Aligned Fan-In, it also adds new ones.

Readability

The Unaligned Fan-In pattern may decrease the readability and general understanding of the data flow. This is particularly visible when you add two types of downstream tasks, one executed when all parents succeed and another to execute when there are some failures. Your pipeline will be confusing as you will not be able to easily determine the execution flow without going into code (see Figure 6-4).

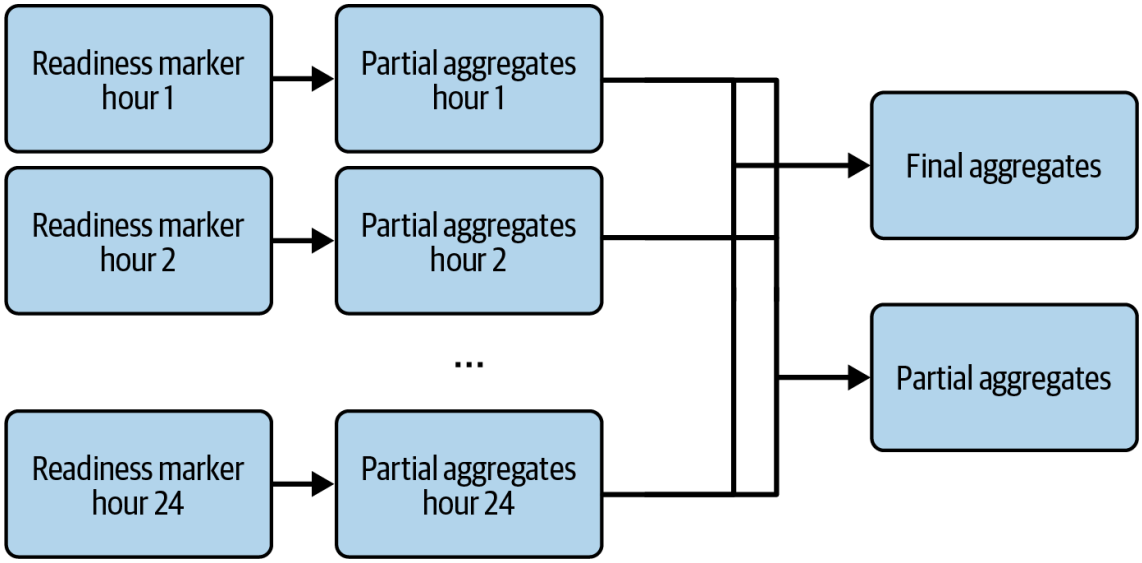

Figure 6-4. Confusing Unaligned Fan-In example

To mitigate this issue it is always better to check if the data orchestration tool provides custom functions to handle the "other" scenario. When it comes to dealing with errors, Apache Airflow has an on_failure_callback function to manage any failed invocation, while AWS Step Functions supports this with a Catch field type where you can trigger a failure handler directly.

Partial data

If you decide to generate the dataset from partially successful parents, it's important to share this fact with the consumers of the dataset. Otherwise, they may make incorrect assumptions about the completeness status.

There are different ways to achieve this. The simplest way is to store the completeness metric in a completeness table that's a companion to the generated dataset and then do some math. For example, if only 12 of 24 parents succeed, then data will be complete at 50%; if only 6 succeed, it'll be 25%, etc. Another way is to add this information to the metadata layer (e.g., as tags for objects written in cloud object storage). You can also complete this static information with a notification shared with downstream consumers (for example, via an email).

You can also decide not to share the partial dataset but instead keep it private. In that case, you will need to determine the completeness first and write the results to an internal table or folder if the completeness status is not 100%.

Examples

If your data orchestration solution lets you define outcome dependencies between tasks, the Unaligned Fan-In implementation should be easy. After all, it's just a matter of using the correct trigger configuration. Example 6-9 shows how to do this for Apache Airflow.

Example 6-9. Triggering condition for the Unaligned Fan-In in Apache Airflow

```
clear_context = PostgresOperator(...)
generate_cube = PostgresOperator(
 # ...
 trigger_rule=TriggerRule.ALL_DONE
)

for hour_to_load in [f"{hour:02d}" for hour in range(24)]:
file_sensor = FileSensor(
  task_id=f'wait_for_{hour_to_load}',
  filepath=input_dir +'/date={{ ds_nodash }}/hour=' + hour_to_load+'/dataset.csv'
)
visits_loader = PostgresOperator(
  task_id=f'load_hourly_visits_{hour_to_load}',
  params={'hour': hour_to_load}
)

 clear_context >> file_sensor >> visits_loader >> generate_cube
```

Example 6-9 is identical to Example 6-6, except for one little detail: the trigger_rule. By default, Apache Airflow starts a task only if all its parents succeed. Here, the pipeline uses a different condition. It simply expects all parent tasks to complete, regardless of the outcome. That's a pretty easy way to express the dependencies, isn't it? The only problem is that the rule remains hidden in the code, and you won't see it if you analyze the pipeline graph visually.

In addition to this purely orchestration-related change, the pipeline == includes a new flag for the output table. The flag marks a given row as approximate if the number of processed hours for the day is different from 24 (see Example 6-10).

Example 6-10. Approximate flag computation in SQL

```
INSERT INTO dedp.visits_cube (...,is_approximate)
SELECT ...,
 (SELECT CASE WHEN hours_subquery.all_hours = 24 THEN false ELSE true END FROM
  (SELECT COUNT(DISTINCT execution_time_hour_id) AS all_hours FROM dedp.visits_raw
    WHERE execution_time_id = '{{ ds }}')
 AS hours_subquery)
```

```
FROM dedp.visits_raw GROUP BY CUBE(...);
```

The code in Example 6-10 uses a subquery to determine whether the table is based on complete or incomplete input data.

AWS Step Functions is a serverless data orchestration offering on the public cloud and also an example of a less declarative implementation of the Unaligned Fan-In pattern. To demonstrate this, we're going to use three lambda functions (see Example 6-11).

Example 6-11. Returned elements of lambda **functions**

```
# lambda-partitions-detector

def lambda_handler(event, context):

 # ...

 partitions_to_process = []

 return partitions_to_process


# lambda-partitions-processor

def lambda_handler(event, context):

  # ...

 processing_result: bool

 return processing_result


# lambda-table-creator

def lambda_handler(event, context):

 table_metadata = {}

 if False in event['ProcessorResults']:

  table_metadata['is_partial'] = True

 # ...

 return True
```

Example 6-11 shows the three functions used in the Step Functions workflow. The first one detects new partitions to process, and it has only one occurrence. The second one is defined as a Map task in the workflow. It runs individually for each of the items returned by the lambda-partitions-detector and generates a boolean flag to mark the processing as successful or failed. The last function, lambda-table-creator, takes all the processing_result flags and detects whether there were any errors in processing. If there were, it annotates the table with an is_partial attribute. The full code of the workflow is available in the GitHub repo, and I'm omitting it here to stay focused on the less declarative part of the implementation.

Fan-Out

So far, you have seen pipelines that always merge into a common task. But we're still missing the last type, in which one task is the input for others. This approach can be useful when one dataset is used by multiple teams for different purposes, such as data analytics or data science.

Pattern: Parallel Split

In the first fan-out pattern, one parent task is a requirement for at least two child tasks. They can run in parallel because their logic is isolated and the single common point is the same parent requirement.

Problem

You're about to replace a legacy data processing framework written in the C# programming language, which nobody in your organization knows anymore. All the maintainers left the company without leaving any useful documentation. You've performed a reverse-engineering step, and now, you are rewriting the logic with a modern open source Python library. At this point, you need to migrate the pipelines, but since your reverse-engineering approach may not be perfect, you prefer to keep the old pipelines running until their consumers have to switch to the new solution. Therefore, during the migration, you'll need to write the processed dataset in two different places.

Solution

The problem states that two different operations depend on a common parent task. That's the type of situation in which you should use fan-out patterns. The first of them is the Parallel Split pattern, and as the name indicates, it breaks the work into parallel parts.

The implementation on top of the data orchestration layer is straightforward. It relies on the data flow definition API that can use a dedicated domain-specific language (DSL) or high-level programming abstractions, like functions.

The Parallel Split also applies to the data processing world, but the implementation is not that simple as it involves several points to keep in mind. First, the split processing shouldn't trigger separate data reading operations. Instead, all common computations should run only once. You can achieve this by materializing the intermediary dataset with either a temporary table in SQL or a .persist() method in Apache Spark.

Next, you need to ensure that the parallel branches don't interfere with each other. Therefore, if your codebase uses any global and shared variables, they should either be read-only or have modifications that are compatible across all writing processes.

Finally, if your computation is execution time sensitive, you should allocate dedicated compute resources or define the auto-scaling to accommodate the split workload in parallel.

Consequences

The hardware problems from the Solution section are valid concerns, but there are even worse things that might happen. Let's see which ones.

Blocked execution

This drawback is valid for the data orchestration layer and time-dependent pipelines, where each execution runs only if the previous one succeeded. As you may have already guessed, in the case of a Parallel Split, the triggering condition will be based on the slowest branch created in the Parallel Split pattern. Put differently, each pipeline will have to wait for the slowest branch to complete. If there are failures, the consequences will be even worse as the subsequent executions will not run at all.

If there is such a risk, you might consider exporting the slow branch to a dedicated pipeline whose execution will be conditioned on one of the trigger rules presented in the Isolated Sequencer pattern, namely, the dataset- or task-based dependency.

Hardware

Let's return to the data processing layer. If the main job needs to generate an intermediary dataset for two other jobs, then they should have the same hardware expectations. For example, if one of them is CPU heavy and another is memory heavy, you won't be able to use the adapted infrastructure.

To mitigate the issue, you'll need to divide this job as you did when solving the problem stated in the Local Sequencer pattern, by generating the intermediary dataset and later starting the parallel jobs on their dedicated hardware. Figure 6-5 shows this operation in a schema.

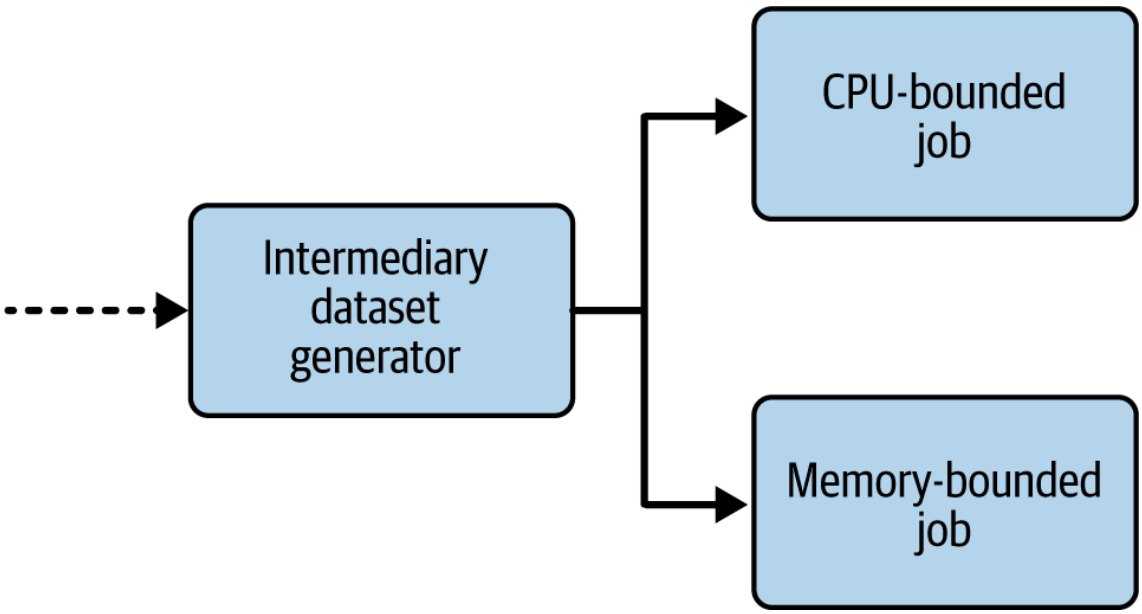

Figure 6-5. Splitting a job working on the same base dataset but requiring different compute capacity (dashed arrow indicates previous tasks in the pipeline not relevant to this example)

Examples

How should you implement the Parallel Split pattern? In the examples you've seen up until now, you can leverage the DSL capabilities. First, let's take a look at Apache Airflow (see Example 6-12).

Example 6-12. Parallel Split in Apache Airflow

file_sensor = FileSensor(#...

```
 task_id='input_dataset_waiter')


for output_format in ['delta', 'csv']:
 load_job_trigger = SparkKubernetesOperator(# ...
  task_id=f'load_job_trigger_{output_format}',
  params={'output_format': output_format}
 )
 load_job_sensor = SparkKubernetesSensor(#...
  task_id=f'load_job_sensor_{output_format}')


 file_sensor >> load_job_trigger >> load_job_sensor
```

Example 6-12 uses the same model as the examples of the fan-in patterns. However, there is a subtle difference in the end of the sequence that doesn't end with a common task. Instead, the single common point is the first sensor operation (file_sensor).

The data processing layer also supports the Parallel Split. Even though the implementation is also straightforward, you must keep in mind one thing. You're going to apply a different processing logic on top of the same input dataset, so it's enough to read it only once! To achieve this in Apache Spark, you must call the persist() function for the read dataset (see Example 6-13).

Example 6-13. Caching the input dataset in PySpark

```
input_dataset = (spark_session.read
  .schema('type STRING, full_name STRING, version STRING').format('json')
  .load(DemoConfiguration.INPUT_PATH))
input_dataset.persist(StorageLevel.MEMORY_ONLY)
input_dataset.write...
input_dataset.write...
```

This code memorizes the read dataset in memory. If you're worried about not having enough capacity in RAM, you can also configure the function to store the dataset in memory and eventually on disk, if all available space is taken.

Besides this caching aspect, you may also want to avoid writing the data twice in case of retries. Multiple executions will increase the dataset volume but also may introduce data quality issues, such as duplicates. But the good news is that some data formats have protections against the retried writes. That's the case with Delta Lake, in which you can leverage the txnVersion and txnAppId options (see Example 6-14).

Example 6-14. txnVersion **and** txnAppId **with Delta Lake**

```
batch_id = 1
app_id = 'devices-loader-v1'

input_dataset.write.mode('append').format('delta')
 .option('txnVersion', batch_id).option('txnAppId', app_id)
 .save(DemoConfiguration.DEVICES_TABLE))

(input_dataset.withColumn('loading_time', functions.current_timestamp())
  .withColumn('full_name',
    functions.concat_ws(' ', input_dataset.full_name, input_dataset.version))
  .write.mode('append').format('delta')
  .option('txnVersion', batch_id).option('txnAppId', app_id)
  .save(DemoConfiguration.DEVICES_TABLE_ENRICHED))
```

If, for whatever reason, you need to replay the code from Example 6-14 multiple times, it'll physically write the data for the first run only. Since the values for the txnVersion and txnAppId properties don't change, all subsequent runs will be ignored. If this write deduplication feature is not natively available in your tool, you can still leverage one of the data idempotency design patterns from Chapter 4.

Pattern: Exclusive Choice

The second fan-out pattern also relies on a common parent, but instead of running parallel downstream tasks, it chooses only one.

Problem

The migration you performed with the Parallel Split pattern works perfectly. Now, you need to evolve the pipeline and start executing the new job version on January 1, 2024, only. In case of backfilling, prior days should still run the previous job. You want to make this evolution without creating a new pipeline to keep the full execution history.

Solution

As you can see from the problem statement, there are still two child tasks, but this time, only one should run at a time. These are great conditions in which to use the Exclusive Choice pattern.

The implementation is pretty similar to the Parallel Split as it consists of declaring at least two downstream processes. For their definition, you can still rely on the DSL of the data orchestrator or the functions of your programming language.

Things then change one step before branching. Instead of declaring downstream tasks directly after the last common point, you should add a condition evaluator task to decide which path to follow next.

All modern data orchestration frameworks, such as Apache Airflow (which is open source) and serverless cloud offerings (AWS Step Functions and Azure Data Factory) come with built-in support for condition evaluations with branching. Apache Airflow does it with a dedicated task type, which is a *branch operator*. When it comes to the cloud offerings, Azure Data Factory, for example, implements the Exclusive Choice pattern with an *if condition activity*.

Figure 6-6 shows the pipeline from the problem statement adapted to the Exclusive Choice pattern.

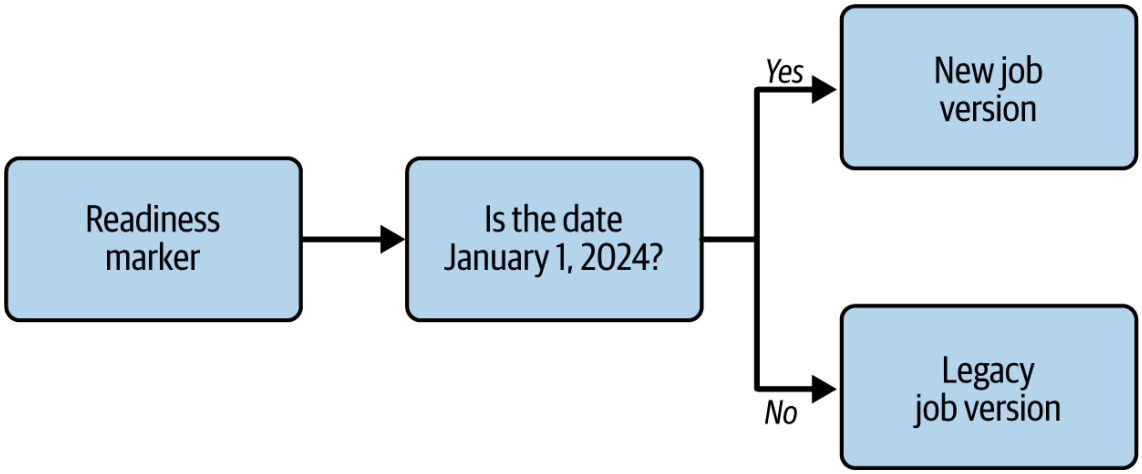

Figure 6-6. Exclusive Choice pattern on top of the data orchestration layer

In addition to the data orchestration layer, it's possible to use the Exclusive Choice pattern in the data processing layer. The implementation is even simpler since you'll inevitably use the if-else or switch statements from the programming language of your compute layer.

Consequences

The greatest danger posed by the pattern comes from its flexibility. You can add as many branches as you want, and having a large number of branches will pretty quickly degrade understanding of the pipeline.

Complexity factory

Since it's not uncommon to see applications with multiple if-else statements, it may be tempting to repeat this practice in the data orchestration layer. However , in that context, multiple conditions mean multiple execution branches that eventually merge together or split even more. Your components will then become barely readable.

There is no single accepted threshold for if-else conditions, though. To find out if there is something wrong, you can try to explain the pipeline to an imaginary friend who has joined your project recently. If the explanation is not concise and leads to lots of questions and answers, the pipeline may be too complex.

Rubber Duck Debugging

The method of explaining things to an imaginary friend is also known as *rubber duck debugging*. Even though it's often presented as a debugging method, you can use a similar approach to analyze complexity.

Hidden logic

This applies to the data processing implementation. If your job includes conditional statements and branches, possibly generating different datasets or interacting with different output data stores, this can become problematic in the future.

Often, you'll remember and understand the code for a few days or weeks after you write it. Then, over time, you'll write more code, solve other problems, and probably forget that this particular job has many possible outcomes.

For that reason, it can be better to apply the Exclusive Choice pattern on top of the data orchestration layer too for a given pipeline. That probably sounds confusing, so let's take a look at Example 6-15.

Example 6-15. Conditional execution in the data processing layer

```
if condition_a:
  process_condition_a()
else:
  process_default()
```

If your code does what the code in the example does, then you have two choices. Either you can trigger the job directly and delegate condition evaluation to the processing layer, or you can evaluate the condition in the data orchestration layer and, depending on the outcome, invoke a different entry point, as in Example 6-16.

Example 6-16. Conditional execution in the data orchestration layer

```
if condition_a:
  python job_for_condition_a.py
else:
  python job_for_default.py
```

Heavy conditions

If you implement the pattern in the data processing layer and your condition needs to process the data, be aware that this will impact the execution time of the job. For that reason, it's always better to prefer metadata-based conditions, which are faster to run as they don't interact with the dataset.

When you can't avoid processing the data, optimize this step as best you can. Ensure that you don't process it many times and that there is a way to implement an incremental processing that doesn't read the whole dataset each time.

Examples

First, let's see how to implement the Exclusive Choice pattern on top of the data orchestration layer, in our case, with Apache Airflow. Previously, you have seen how to parallelize the work just by declaring the pipeline's sequence multiple times. Unfortunately, this technique won't work for the Parallel Split. Instead, you need to rely on

a BranchPythonOperator that will route the execution to the correct branch (see Example 6-17).

Example 6-17. Conditional router for the Exclusive Choice pattern in Apache Airflow

```
def get_output_format_route(**context):
 migration_date = pendulum.datetime(2024, 2, 3)
 execution_date = context['execution_date']
 if execution_date >= migration_date:
  return 'load_job_trigger_delta'
 else:
  return 'load_job_trigger_csv'


format_router = BranchPythonOperator(
 task_id='format_router',
 python_callable=get_output_format_route,
 provide_context=True
)
```

Example 6-17 shows a router implementation based on the execution date. As you can see, it generates CSV files or a Delta Lake table, depending on the execution date. Although the example includes two routes, you can always add more. However, keep in mind that more is not always better because it might make your pipeline less readable. For that reason, instead of branching, you could simply create another pipeline with the start_date and end_date parameters to control the time validity.

Additionally, the pattern also works in the data processing layer. It may come in one of two flavors. The first one comes from an external configuration, and you can see it in Example 6-18.

Example 6-18. The Exclusive Choice implemented as job parameters in PySpark

```
class OutputType(str, Enum):
 delta_lake = 'delta'
 csv = 'csv'


parser = argparse.ArgumentParser(prog='...')
parser.add_argument('--output_type', required=True, type=OutputType)
args = parser.parse_args()
```

```
output_generation_factory = OutputGenerationFactory(args.output_type)


spark_session = output_generation_factory.get_spark_session()
raw_data = (spark_session.read...)


output_generation_factory.write_devices_data(raw_data, args.output_dir)
```

Example 6-18 shows a batch job that, depending on the output_type input parameter, generates a Delta Lake table or a set of CSV files. But wait, where is this logic? Here, we're leveraging software engineering (SWE) design patterns and, more specifically, the Factory pattern.[3] In a nutshell, this specific design pattern hides the creation logic behind an interface, which is the single element exposed to the user. In our case, the OutputGenerationFactory is such an interface and, at the same time, the class responsible for generating correct objects to the expected input (see Example 6-19).

Example 6-19. SWE Factory design pattern supporting the Exclusive Choice in PySpark

```
class OutputGenerationFactory:

 def __init__(self, output_type: OutputType):
  self.type = output_type

 def get_spark_session(self) -> SparkSession:
  if self.type == OutputType.delta_lake:
   return (configure_spark_with_delta_pip(SparkSession....)
  else:
   return SparkSession.builder...getOrCreate()

 def write_devices_data(self, devices_data: DataFrame, output_location: str):
  if self.type == OutputType.delta_lake:
   devices_data.write.format('delta')...
  else:
   devices_data.coalesce(1).write...format('csv')...
```

As you can see in Example 6-19, the client code receives a different SparkSession and the write action is selected automatically, based on the output type argument. The only drawbacks here are the conditions repeated in each method. That's fine for our simple example, but it might not be for your use case. The good news is that you can avoid them

with dedicated classes for Delta Lake and CSV writers returned by the factory, instead of particular functions.

In addition to the external parameters, you can rely on the dataset characteristics to control the execution flow (see Example 6-20).

Example 6-20. The Exclusive Choice pattern driven by a schema change

```
input_dataset = ...

input_schema = detect_schema(input_dataset)

output_location = DemoConfiguration.DEVICES_TABLE_LEGACY

if len(input_schema.fields) >= 3:

 output_location = DemoConfiguration.DEVICES_TABLE_SCHEMA_CHANGED
```

This example adapts the job to the input schema of the file. If it has at least three fields, the job writes it to a new location. If not, it's considered as a legacy table. Here, we're relying on the metadata layer to get this information, and as you already know from Chapter 4, metadata operations are often much faster than data operations. However, since you can use anything as the conditions, including the data, you should be aware that it may be more costly.

Orchestration

So far, you have only been organizing individual flows alongside their internal and external dependencies. At this stage, they are simple static resources sitting on top of your data orchestration layer. With the last family of patterns, they will become dynamic components running data processing tasks.

Pattern: Single Runner

The data orchestrator must run each of the declared pipelines. The question is, how? The first pattern is the most universal one, but this universality comes with some runtime costs. Let's see which ones.

Problem

In your most recent project, you implemented the sessionization pipeline with the Incremental Sessionizer pattern from Chapter 5. Since it was a proof of concept (POC), the orchestration was not in the scope. To validate the sessions with your business owners, you have been running the job manually, on demand. As the project is about to enter into the release cycle, you need to work on the data orchestration.

You already have the pipeline graph but are still looking for a way to execute it.

Solution

The solution is simple: you need a runner! The sessionization problem involves incremental, thus sequential, execution. As a result, you cannot run more than one pipeline at a time.

The Single Runner pattern ensures there is always a single execution of a given pipeline. The implementation consists of configuring the concurrency level of the data flow, and this can be easily achieved with Apache Airflow and Azure Data Factory, thanks to their

configuration-driven approach. Both orchestrators support a concurrency attribute that can be set to 1 if you want to always run at most one occurrence at a time.

If the native capability is not supported, you might need to implement a Readiness Marker pattern that will be waiting for the previous execution to complete. A small gotcha here is that the solution doesn't prevent the triggering of future runs that will all be waiting for their predecessors to complete before processing the data.

Consequences

The limited concurrency does make sense from a logic standpoint. However, it may have some important implications in the daily life of the pipeline.

Backfilling

The limited concurrency probably will not be problematic as long as you don't need to backfill. However, if you do, then reprocessing will be very slow because of the sequential character of the pipeline. Unfortunately, there is not a lot you can do since the single concurrency is a business requirement that cannot be relaxed. In other words, if you run multiple pipelines in parallel, the results will be wrong.

The only thing you can do is to see whether you really need to backfill the whole pipeline each time. For example, if your data flow is composed of data processing and data loading steps, and if you only need to insert a generated dataset into a different location after changing your configuration, then you can avoid the data processing part.

Latency

Backfilling is the worst-case scenario for the Single Runner pattern. A less serious case is one with stragglers, in which some pipelines run slower than others. Let's take a look at an example of an hourly scheduled data flow. Usually, the execution of all tasks takes 30 minutes, but recently, it increased to 1.5 hours. As you can see, data will be increasingly delayed. And the consequences can be very serious. Not only will your pipeline get slower and slower, but all your downstream consumers will also suffer from this extra latency.

This shouldn't be the most common scenario, though. You can always mitigate it, assuming that you're running your workloads on scalable infrastructure, by adding more compute power or improving the processing logic.

Examples

Now, let's take a look at some examples. The first one is an Apache Airflow pipeline that compares the current day's data with the previous day's data. You can already see the sequential dependency between the steps, which would be broken if you ran multiple simultaneous executions in case of backfilling. How should you enforce this order? As with many other things, you do it with the configuration (see Example 6-21).

Example 6-21. Concurrency limits in Apache Airflow

```
with DAG('visits_trend_generator', max_active_runs=1, default_args={

 'depends_on_past': True,

 # ...
```

Example 6-21 configures the concurrency with the max_active_runs and depends_on_past properties. The first attribute determines how many concurrent pipelines are allowed. Here, as the logic relies on the sequential execution, the value must be 1. The depends_on_past attribute applies to each task and enforces execution upon success of the previous run. Put differently, if task A from the previous run fails, the same task for the current run won't start.

You can also control the concurrency cluster-wise. The aforementioned AWS EMR service not only provides steps to control the execution order but also provides the StepConcurrencyLevel parameter to determine how many jobs can run in parallel on a cluster.

The concurrency limits are present in data orchestration services on the cloud too. For example, if you create an Azure Data Factory pipeline, you can configure the concurrent executions with the Concurrency setting. However, the feature has some gotchas. While Apache Airflow won't schedule a new run if there is one active, Data Factory will create them and store them in a bounded queue that supports up to 100 concurrent runs. The service will still schedule other runs, but if the queue is full, the scheduling action will return a 429 error, which stands for "Too many requests."

Pattern: Concurrent Runner

The backfilling and latency issues of the Single Runner pattern can be easily addressed with the next pattern. All you need to do is relax the concurrency constraint.

Problem

You're on the data ingestion team, and your goal is to bring data from external at-rest sources to your internal database as soon as possible. Usually, the ingestion frequency goes from 30 minutes to 1 hour, but sometimes the whole process takes longer. Since it's running with the Single Runner pattern, all subsequent deliveries are delayed. You're wondering whether sequential execution is required, because the loaded datasets are not dependent on each other.

Solution

The datasets ingested by your team are independent. That means you can ingest in any and therefore with a relaxed concurrency constraint. This is where you can use the Concurrent Runner pattern.

The implementation is rather simple. It consists of defining a concurrency that's higher than 1. However, with great power comes great responsibility. You need to strike a proper balance that will take other pipelines into account in the context of your infrastructure.

Once this concurrency step is defined, the data orchestrator will pick the next pipeline execution available for scheduling as long as the currently running instances don't reach the allowed maximum concurrency level.

Consequences

Running multiple pipelines at the same time may have some important impacts, especially if your concurrency level is too restrictive.

Resource starvation

This is particularly true if you work in a multitenant environment (i.e., if when your orchestrator is used by many different teams). If a bunch of pipelines have a pretty high concurrency level and they run backfilling at the same time, the scheduler may not have enough capacity to start other pipelines.

The best way to handle this is with concurrency control of the pipelines with the *workload management feature*. This consists of allocating a specific compute capacity to a group of users running their pipelines. That way, even if a team sets the allowed concurrency to a very high level, it won't be able to use more capacity than the threshold assigned in the workload.

Resource starvation may not be an issue for serverless orchestrators, which often offer a pretty high concurrency level. For example, as of this writing, AWS Step Functions allows up to 10,000 parallel child workflow executions.

Shared state

The shared state is a pretty common gotcha for everything that allows concurrent execution. If the pipeline works on a shared component, concurrent and therefore nondeterministic execution can be a source of many problems due to unexpected side effects. One example could be the Dynamic Late Data Integrator pattern from Chapter 4, in which concurrent runs could at best trigger backfilling multiple times or, at worst, not trigger it at all for some execution dates.

Examples

The concurrent execution example is straightforward. You simply need to configure allowed concurrency to a value of more than 1 (see Example 6-22).

Example 6-22. Concurrent orchestration in Apache Airflow

```
with DAG('devices_loader', max_active_runs=5,
 default_args={
 'depends_on_past': False,
# ...
```

Although the example defines depends_on_past as False, you can change this value if particular tasks should depend on their prior executions. That won't affect the concurrency configuration, but it might prevent the pipeline from moving when it encounters a task with failed past execution.

This flexibility level doesn't exist in all data orchestrators, though. In Azure Data Factory, which we introduced in the previous section, you can define the concurrency and also trigger-based dependency but not task-based dependency.

Summary

In this chapter, you learned different ways to organize data flow dependencies. First, we discussed sequences. There, you saw two patterns you can use to coordinate a workflow, either at a local level with the Local Sequencer or globally with the Isolated Sequencer. Using a good approach here will help you define the pipeline boundaries to keep them readable and maintainable.

In the next section, you discovered fan-in patterns. Their purpose is to merge isolated execution branches, and using them is a great way to parallelize the work of isolated parts of a pipeline. Depending on your business constraints, you can use the aligned or unaligned versions. The former requires all parallel branches to succeed, whereas the latter relaxes this constraint and is a good candidate for generating partial results.

At the opposite end of the spectrum from fan-in patterns are fan-out patterns that create the branches. They also come in two flavors. The Parallel Split pattern starts two or more branches from a single task. Put differently, the output of one task is the input for many other tasks that can be isolated on purpose due to hardware or business reasons. An alternative to this simultaneous execution is the Exclusive Choice pattern that follows only one of the many declared branches, a little bit like the code you write for if-else statements.

In the last section, you saw two different orchestration patterns. The first of them was the Single Runner, whose goal is to always run one instance of the pipeline at a time. That's crucial for incremental data processing, in which the current instance's logic depends on the previous instance's output. On the other hand, you have the Concurrent Runner. As the name implies, it allows multiple instances to run in parallel. It's a good candidate for accelerating isolated executions (for example, during backfilling).

But your responsibilities as a data engineer don't stop there, at defining data flow. There is another, maybe even more important aspect: data security. The next chapter will show you data engineering design patterns that should help you approach this essential area.

1 This assumes immutable data at the source. If the producer can remove the dataset, you should of course run the Readiness Marker step again.

2 To get started with OpenLineage, visit the official website.

3 You can learn more about this and other patterns from the Refactoring Guru website and *Design Patterns: Elements of Reusable Object-Oriented Software*.