AN ASSESSMENT OF PIPELINE ORCHESTRATION APPROACHES

Ascend.io

Table of Contents

Intro

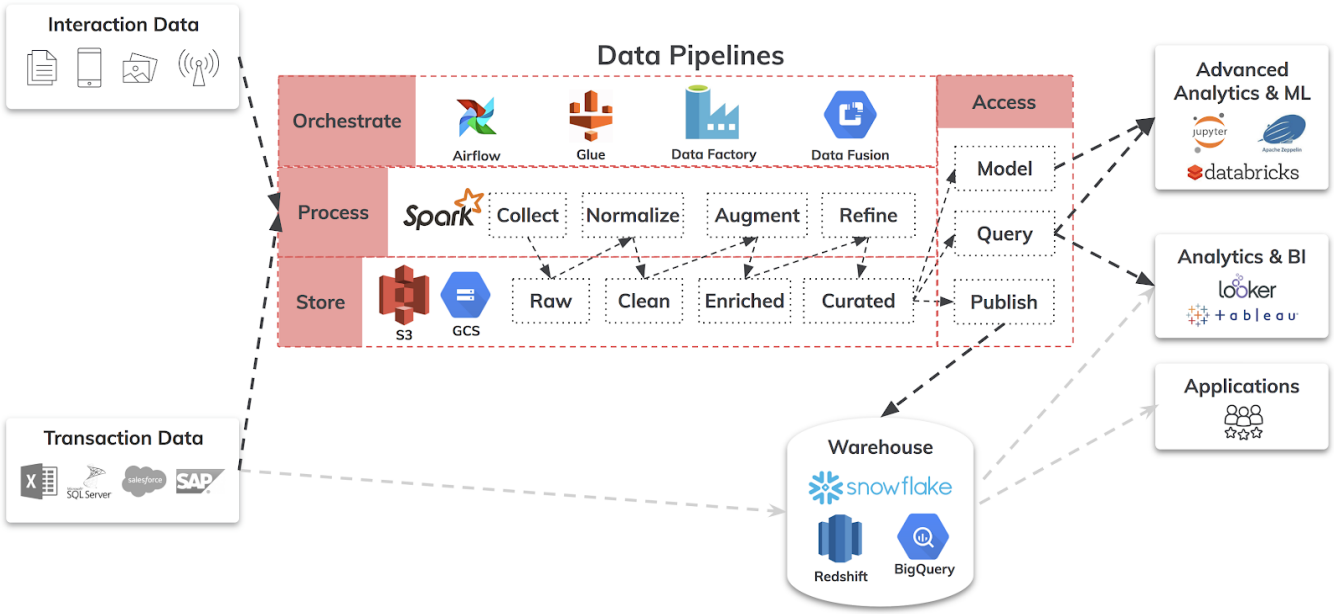
The modern data architecture holds a lot of promise for enterprises looking to tap into more data and fuel more downstream consumers, all while operating more efficiently and cost-effectively. However, for any of us who have worked with data over the past few years, we know that utopian state is often far from the reality.

While there's been no shortage of innovation - from the cloud services processing your data, to the elastic warehouses, all the way to the BI and data science tools - connecting and orchestrating the movement of data between these systems has been conspicuously absent from these advancements.

The current state of the art in orchestration are workflow automation systems. Using these systems, you are responsible for designing the tasks, connecting them together via dependency relations, and passing them to a workflow automation system that manages the execution. This model puts the manual burden on you to manage the enumerable details within (and between) each task and ensure data is correct, often resulting in brittle pipelines that are expensive to run and maintain.

There is an alternative: dataflow automation systems. This approach leverages algorithms to translate high-level specs into tasks and to schedule and execute those tasks. You're responsible for creating and curating the high-level spec and now the dataflow automation system manages the legion of tasks required to implement your spec.

In advancing orchestration, Ascend took the dataflow automation approach. We've seen this approach lower the design and maintenance costs while also improving the quality and reliability of resulting data pipelines, just as moving from assembly language to higher-level development languages has done for software. We'll explore why we opted for this route by looking at more of the challenges and limitations within workflow automation systems, as well as compare and contrast both approaches overall.



Modern Data Architecture

What is Workflow Automation?

Workflow automation systems schedule and execute a directed acyclic graph (DAG) of tasks. The system guarantees that tasks are only executed when their upstream tasks have successfully completed.

In addition to this entirely asynchronous scheduling, a set of tasks—those having no upstream dependencies—are often scheduled to run on a fixed, static schedule, such as once an hour or once a day.

Today, a large part of a data engineer's job is to design tasks that a workflow automation system can run. The engineer then designs a dependency graph such that these tasks run in an acceptable order. Finally, the engineer is responsible for configuring this automation system so that it runs periodically scheduled tasks when data should be available. In some cases, they may also have to manually trigger execution of tasks and graphs on historical data (ie backfills).

The Emergence Of Workflow Automation

Coinciding with the ubiquity of technologies that handle virtually unlimited volumes of data - such as scalable compute engines like Apache Spark and Apache Presto and flexible object storage like AWS S3 and Google Cloud Store - came more complexity with how you work with all this data. For instance, when you query against distributed data, you need to manually partition data and then simplify your queries based on this partitioning (again, manually). Workflow automation systems gained popularity to help restitch these partitions. You can configure these systems to schedule the simplified queries to reflect your manual partitioning and your understanding of when new data drops. However, there remains a strong reliance on what can be defined and executed manually. Let's break down the steps needed for pipeline development using workflow automation systems to better understand how this manual effort gets more intractable over time.

Partitioning Design

When working with data against these scalable technologies, the first step is to partition data so new and updated data is separated from older, unchanging or rarely changing data. The default here is usually date-based partitioning.

Partitioning considerations start at raw ingestion. You need to know where the data will land, how events are separated into files, and the pattern used to name those files. Tradeoffs can arise here based on downstream usage requirements. For instance, if some data consumers require low latency insights, your ingest pipeline may create many small files. This, however, is often an anti-pattern for aggregation jobs and means you have to manually implement additional tasks to aggregate sets of small files before further processing.

Additionally, events are typically stored in files based on the time the event was received by an ingestion pipeline. Since it takes time to transfer and land data, event timestamps and files do not line up. It's virtually always the case that data for the end of the day (e.g., 11:59PM on December 31st) will end up in a file for the next day (e.g., the midnight hour bucket for January 1st). You are responsible for anticipating this and designing for it, which typically means writing tasks that read wider windows of data and filter out events not relevant for a particular aggregation. This is always a heuristic: it implicitly embeds your assumptions based on expected patterns and schedules for worst case latency, even if typical actual latency is far less.

You also need to anticipate how downstream dependent tasks will consume the results of upstream

tasks when designing partitioning. Let's take a simple example: an internet commerce provider that wants to track total transaction value per customer on a daily basis as well as rolling week-over-week total transaction value per customer. In this example, you'll want to run two jobs: one for the daily aggregation and one for the week-over-week difference. That means when designing the daily job, you need to consider how the weekly job will read that data and the irregularity of the calendar (different months with different numbers of days, different numbers of days in different years). Some path layouts make this easier than others.

Foreseeing and accommodating downstream task requirements gets progressively more difficult as data use grows and diversifies.

Storage Plan for Intermediate Results

For a pipeline, the location of the raw input data and location of resulting output data are both provided. But allocating and managing the location of intermediate data falls on you and is handled as part of the task and DAG design.

In the prior example, you'll need to run daily aggregations, store them temporarily, and then run week-over-week aggregations. Because of the running-week requirement, this data must be computed every day on the basis of multiple days of input.

In practice, intermediate storage is often not well documented and can be difficult to manage, debug, etc. It's not uncommon to have large amounts of data where dependencies are not easily identifiable, which can make it difficult to determine when data can be safely removed and more complex when future backfills must be accomodated.

Write and Connect Tasks via Dependency Graph

Once the above partitioning and storage are designed, you write the task templates and configure the workflow automation system to run them in the correct order.

This effort entails:

1. Reducing the high-level specification to a task template that produces a single output partition based on a subset of the input set. This can take several steps to select only the necessary partitions and simplify it down so only a particular task runs.

2. Connecting the tasks with the parameters necessary to pass data from input to output locations

3. Configuring the triggers that will cause the task to be scheduled, either

    a. Configuring the dependencies on upstream tasks so that the task is run when the upstreams complete successfully

    b. Defining a time-based execution so the task is run at regular intervals

4. Creating the code necessary to send the task to the compute cluster, e.g., submit to a Spark cluster

Scheduling is often subtly complicated. In the case of the daily aggregation task, you need to decide when to trigger the task based on the expected time that data should be available (remember all those ingest assumptions made at the start?). If data is dropped at the end of every hour, you might schedule the daily aggregation task to start at 2AM every day to account for:

- Late data arriving in at midnight of the following day
- The midnight hour data not being finalized until just before 01:00 that day
- Some delay for finalization after the 01:00 hour starts

The weekly rollup example has similar scheduling requirements plus it must wait until the daily rollup for each day is available. Again, this is based on an assumption of when the daily data will be available, often with an additional check that the data has, in fact, been completed with a retry mechanism if the data is not ready.

The weekly rollup has the additional complication that it depends on the daily results for seven days, not just the most recent day. This dependency is often difficult to express in workflow automation systems, so it's often just left out. This creates an implicit assumption that if yesterday's data is available, all the previous days' data is available. In normal operation, this is expected to be true. When anomalies occur, this may not be the case. Because the implicit assumption is not exposed through the workflow system, the system cannot enforce correct task execution order and incorrect data may be produced.

Moreso, even normal operations do not fulfill this assumption, in particular, backfills. When backfilling historical data, it would be very common to compute daily aggregations in reverse chronological order - with the most valuable data likely to be the most recent. However, this is completely backwards of what the implicit dependency requires. This makes backfills laborious and brittle with the result that, though the data would be valuable, most of us choose not to do it.

Monitor and Respond to Failure

Once everything is deployed and running, both the entire workflow automation process as well as the individual tasks must be monitored, and alerts are created if things go awry. It's a fact of life in big data systems that failures happen. "It is possible to fail in many ways…while to succeed is possible only in one way" rings true in workflow systems:

- Data sometimes arrives late
- Previously provided data gets restated
- An external service necessary for task execution becomes unavailable
- Input data organization changes
- Input data schemas change
- One or more more task definitions change and downstream tasks subsequently fail

Anyone that has worked in scalable data systems has experienced at least one of these. Many of us have experienced all of them. Multiple times.

In the simple case of an ephemeral error (e.g., a required external service is briefly unavailable), corrective action can just be retrying a task. Many workflow systems can retry failed tasks, though without the context of the specific error, they will retry persistent errors (e.g., bad SQL) just as they do ephemeral errors. For persistent errors, it's up to you to determine root cause and implement the task or dependency graph changes necessary to correct the failure.

On its own, correcting for a single task failure can be laborious but is generally tractable. The complexity increases vastly as the size and number of workflows increases.

Restated data (data that was incorrect on first access) is particularly insidious. When first dropped, it is considered valid data so all downstream tasks can run to completion. When the incorrect data is detected (which can take days or sometimes weeks), it's necessary to identify and rerun tasks that have read that data. It's also necessary to identify the transitive downstreams of these tasks and rerun them, as well, while preserving the necessary dependencies. In practice, this process is so laborious, if it's done at all, it is done only for a limited set of manually identified, high-value results. This means the accuracy of the rest of the data in the system remains unknown indefinitely.

Writing dependencies that will hold correct under all conditions is difficult. In response, we consider the

most likely case (new data comes in accurately and on time, upstream tasks are triggered and complete successfully within an expected time window) and write the dependencies necessary to get correct results under these conditions. While this simplifies immediate design tasks, it makes it difficult when atypical situations arise. The resulting system can be brittle in the face of these anomalies, requiring increasing manual remediation over time.

Why Not Just Improve Workflow Automation Systems?

Workflow automation systems only know that a specific task can be run if and only if its upstream tasks have run successfully. These systems do not know why they must run the tasks this way. They don't know why it's useful to run these tasks. They don't even know what the (side)effects of a single task are.

Formally, this workflow model is ***referentially*** *and* ***semantically opaque****.*

Referentially opaque means the workflow system does not know what a task reads as inputs or what it writes as outputs.

Semantically opaque means the workflow system does not know how the inputs of a task are transformed into its outputs. As a result, it cannot consider task optimization.

Opacity also exists between the system and the individual tasks. Just as the workflow system has no representation of what each task does, the tasks have no representation of the surrounding context in which they run. They don't know what tasks create the data they're reading or what tasks will read the data they're writing. This prevents the individual tasks from optimizing themselves. For example, if an upstream task uses a GROUP BY clause to create a dataset, the values selected from that GROUP BY must be unique. But any downstream tasks don't see those semantics and thus cannot optimize its query plan based on this property.

Data Engineers as the Compilers

Due to this opacity, the process of taking the high-level description of the problem, reducing it to a number of small steps, and scheduling those steps falls on you as the data engineer. If this process sounds familiar, it's what compilers and query planners have been doing for decades. In this workflow automation world, you are actually acting as optimizing compilers.

But even the best of us make mistakes, especially during anomalous conditions. This is no slam on engineers. This whole development process is based on hard-coded heuristics and assumptions, and manual compilation is:

- Tedious
- Error-prone
- Difficult to do optimally

On the other hand, computers make great compilers. They don't mind tedium. They don't mind considering lots (and lots) of execution alternatives. Additionally, when the problem specs change, the compiler can easily be rerun. Whereas, with manual design, it's up to you to understand the current system and the implications of the change in order to respond.

Writing less code and leaving more to compilers will virtually always result in more correct code in a shorter amount of time (and thus at a lower cost). It doesn't hurt that computers are way cheaper to scale as well.

Dataflow Systems: Focus on Data, Not Tasks

Instead of leaning on engineers to act as task compilers, dataflow automation systems leverage algorithms to handle the translation of higher-level specs into scheduled tasks. What's compelling about this approach is not only the manual relief it provides to us, but also that these systems can view *data* as the primary entity and, thus, only generate tasks as a means to an end.

In contrast to the previous workflow systems, dataflow automation systems are ***referentially*** and ***semantically transparent***, with the specification completely describing the inputs, outputs, and transformations that computation requires. All transformations are seen as high-level representations of data and these systems extract the semantics from the specifications provided.

The transparent descriptions mean these systems have awareness of the context regarding dependencies and input data, so they can now optimally generate and schedule tasks to materialize the correct result set. Moreso, these systems can do this in the face of changing inputs and transformations. Similar to what we've seen with database query planners, these systems will make conservative assumptions during execution by default if the necessary semantics cannot be extracted.

With this approach:

- The correctness of resulting task graphs is (modulo bugs) guaranteed
- The optimality of generated task graphs is dependent on the algorithms used. Generated tasks graphs may not always match the optimality of manually designed task graph just as compiled code may not be as optimal as hand-generated assembly language. While algorithms will improve with time, the cost of manual generation will not and will become an increasing bottleneck.

Let's now look at what pipeline development is like when using dataflow automation systems.

Partitioning Design

Since a dataflow automation system models the semantics of the transformation provided, it's well positioned to automatically choose the partitioning scheme.

The dataflow automation system uses algorithms to analyze the semantics of the query, determine the execution plans to produce the correct results, and select the lowest cost path. These systems can also pull in available metadata to make these choices more optimally based on a larger consideration set.

For example, based on known semantics, these systems can choose to run a cheaper map operation over a costlier full reduction operation when it's known that the map will also produce the correct result. But when data volumes are small, the cost of these two operations switches and these systems can dynamically opt for a full reduction over the now costlier map operation.

Storage Plan for Intermediate Results

Similar to how Spark is responsible for managing the storage of intermediate results during the execution of a Spark job, a dataflow automation system is now responsible for managing storage of results between tasks. Given broad configurations, such as buckets in object stores, the dataflow system handles the actual configuration of data location for each task. The format of the locations or paths within a bucket have no impact on your development, which frees you from this complex and error-prone responsibility. You no longer need to code where every task output should materialize or configure dependent tasks to write/read from these locations.

When the specs change, a dataflow system creates new locations for the resulting transforms and handles

purging the old data at a future point. This guarantees the currently stored value is the desired value, eliminating the need for manual tracking of which task versions were most recently run to determine as much.

Additionally, since dataflow automation systems are semantically transparent and understand the context of the transformations being performed, it's possible in these systems to trivially deduplicate redundant operations. For example, if someone on your team designs a transformation that you already built, these systems can identify this, materialize the existing result set, and not execute requests for the same computation multiple times. This operation is extremely useful during development cycles since you can copy the transformation specs and make small changes without triggering costly recomputation of unaffected data. This is only possible when storage planning is also automated.

Write and Connect Tasks via Dependency Graph

Dataflow automation systems "compile" high-level data transformation specifications into tasks that implement them and produce the correct data results. We've already described that, as part of that process, they automatically perform the partitioning design. Closely tied to this is automatically generating the necessary task templates that then implement the chosen partitioning.

More concretely, in a dataflow automation system with SQL as the high-level specification language, the system will examine the spec in order to create an optimal query plan. Depending on the semantics of the SQL, it will determine whether to run:

- A separate task for each individual input partition/file as they are created or updated (Map Operation)
- A single task whenever any input partition/file is created or updated (Full Reduction)
- A task for affected output partitions when an input that affects the result is created or updated (Partial Reduction)

In workflow automation systems, the burden of translating this into resulting task templates and deciding optimal query plans based on the tasks at hand falls on you. To give credit where credit's due, Spark is able to do this within a single task. However, only dataflow automation systems are able to do the same type of transformation between or across tasks, because they are referentially transparent - making it relatively trivial to extract the dependency relations and optimally generate execution plans based on inter-task relations.

Monitor and Respond to Failure

While you still need to monitor and mitigate errors with dataflow automation systems, the automation of task creation and execution means these systems can offload a vast majority of the monitoring.

A dataflow automation system is responsible for monitoring all the tasks it schedules for execution. Through its semantic understanding of the transforms, it is in a better position to actually interpret the cause and impact of errors, and then respond to a range of recoverable errors accordingly.

For example, if a task reports that a required input file is not present, the dataflow automation system can automatically schedule recomputation of the missing element and, upon success, trigger re-execution of the failed task. It doesn't need to know the cause of the missing input data; it just focuses on generating correct data outputs for the high-level transform.

When task failures are detected that require manual intervention, it is also the responsibility of these systems to interpret the failures in the context of the high-level specification provided and report them back to you at that level, not the task level.

For example, if incorrect data is received, it can cause a task to fail. With a workflow automation system, you get informed of the task failure and then you examine

the task to determine how it failed, the root cause of the failure, and assess the impact of this failure to the data and downstream dependencies. In a dataflow automation system, a task failure is simply attributed to the data it impacts and you are directly informed of the inability to compute a dataset. Details on the task failure are inconsequential here, so you get to focus solely on the impact to the data. Moreover, since all dataset dependencies are tracked by the dataflow automation system, the impact of this failure on downstream data computation can be immediately provided for faster remediation.

Comparing and Contrasting the Approaches

Workflow Automation Systems

Pros	Cons
1. Maturity These systems have been around for decades with new ones arising regularly. This approach is well understood and incremental improvements continue to be developed. **2. Flexibility** These systems can be configured to automate virtually any task that can be expressed programmatically. Since they do not require any representation of the semantics, inputs, or outputs of tasks, there is often no limitation on the programming language, model, or tool used to implement tasks. **3. Simplicity of Dependency Semantics** Dependency representations in workflow automation systems are simple and straightforward to write. Many systems can render dependency graphs as images to help you clearly understand the relationships between tasks.	**1. Task-Based State Management** In these systems, graphs are expressed via the state and dependencies of individual tasks. This means the burden of matching that to the datasets is left to you as the engineer. Since tasks are continuously being created and executed, the volume of this state information grows rapidly with time. This also may require you to prune state data regularly to avoid performance issues. **2. Data Provenance and Lineage Limitations** Since workflow systems track task state only, the state and lineage of data produced by these tasks must be manually inferred from task state, raw task logs, and task definitions. Determining data state this way is time consuming, tedious, and error prone. This is especially true as the size and number of graphs grow. At larger scale, these systems don't track the interdependence of all tasks but instead track the dependencies between entire graphs. Generally the dependency tracking in these cases provides even less detail, making it even more challenging to manually infer data state. **3. No Task or Resource Optimizations**

As all tasks are opaque to workflow automation systems, there is no ability to implement algorithms to optimize computation or execution. For example, they cannot rewrite task dependencies to pre-filter data before a transformation. Since these systems don't model the computation a task is performing, it also has very little ability to estimate the computational resources required by a task and thus optimize concurrent tasks. This means it's up to you to implement scheduling policies, which can result in under-utilizing or over-committing resources.

4. **High Supportability Costs**

   Failure is inevitable at scale. When errors do occur, they can only be detected by observing failures after individual tasks run. Diagnosis then requires manual effort and intervention in locating and reading task logs. Additionally, the hard-coded assumptions made based on data at the point-of-initial development will likely change over time. This not only causes an increasing number of failures but also becomes more difficult and costly to track and resolve these errors in growing, legacy codebase.

5. **Extensibility Limitations**

   Workflow systems do not model the semantics of tasks, thus, they cannot provide any features that validate the consistency and correctness of the relationship between two tasks. Even with an open model of the transforms and dependencies, the precise semantics of the transforms are often not reflected in the model in a consumable way, which blocks the extensibility to more tools.

Dataflow Automation Systems

Pros	Cons
1. Data-Focused Management and Lineage Dataflow systems reflect entire data sets as an organized set of objects and manage the relationships between the objects both within individual tasks and across the overall dataset. This means that you generally only need to consider the overall dataset when designing and managing specs and let the dataflow system manage the lower level details, allowing you to design and maintain far larger, more complex systems. **2. Automation Optimizations** With a transparent understanding of transformations, dataflow automation systems can algorithmically rewrite graphs to produce the same results faster or at lower costs. Similarly, since they automatically generate the individual tasks necessary to implement the high-level transforms, they can dynamically create cost models to manage the execution of tasks based on available compute resources to optimize utilization and performance tradeoffs. **3. Reduced Supportability Costs** These systems are able to monitor task execution and automatically remediate a large proportion of task failures without requiring manual intervention. With less code needed to create the graphs, this also decreases brittleness and the number of potential errors, and makes it more manageable to maintain over time. **4. Rich, Extensible Metadata Model** To achieve much of the automation and optimizations, these systems must maintain rich sets of metadata representing both the high-level system specifications as well as the low-level storage objects and tasks. The richness of this model allows these systems to do extensive correctness and consistency checks. This metadata is also available for other tools to tap into or develop against.	**1. Difficult Manual Optimizations** Similar to other high-level systems, it's more difficult to specify low-level optimizations that the dataflow automation system cannot itself generate. Often the abstraction layer provided by these systems makes it challenging to express lower level optimizations. In general, the quality of optimization will vary across dataflow systems, just as it does across databases and compilers. This can be expected to improve over time but may not presently match manual task design 100% of the time. **2. Limited Flexibility** These systems are purpose-built to move data through a graph, not run arbitrary tasks. While a workflow automation system can be equally capable of computing a set of data transformation tasks or acting as a next gen cron, a dataflow automation system is only well-tuned for the former. **3. Effort to Represent High-Level Task Semantics** Much of the power of dataflow automation systems comes from the semantic transparency. In many cases, such as with SQL, this is simple to define in terms of inputs and semantics. For arbitrary code, it can be more difficult to extract semantics. To maintain the benefits of transparency, it can be necessary to manually provide a description of the I/O and behavior of tasks, which can increase the resulting effort required.

Conclusion

At Ascend, we believe it's the data that matters. Not the tasks, or the scheduling, or the maintenance. This is why we opted to develop a dataflow automation system. This allows you to combine declarative configurations with automation to build and run pipelines with less code and less breaks. If you're interested in giving it a try, sign up for a free trial at https://www.ascend.io/get-started/

About Ascend

Ascend provides the world's first Autonomous Dataflow Service, enabling data engineers to build, scale, and operate continuously optimized, Apache Spark-based pipelines with 85% less code. Running natively in Microsoft Azure, Amazon Web Services, and Google Cloud Platform, Ascend combines declarative configurations and automation to manage the underlying cloud infrastructure, optimize pipelines, and eliminate maintenance across the entire data lifecycle. For more information about Ascend, visit www.ascend.io.