Engineer

パイプライン

データを取得元から利用先へ運ぶ処理の流れ

どのようなものか

パイプライン、正式にはデータパイプラインは、決まった順序で実行するジョブの集合です。それぞれが前のジョブの出力を受け取り、元のシステムが作った未加工のレコードを整え、組み合わせ、レポート、ダッシュボード、ほかのプログラムが読める場所へ保存します。名前の由来はUnixシェルで、一つのコマンドの出力を次へ渡す | です。大半のパイプラインは人の操作を必要とせず、予定した時刻や新しいファイルの到着時に動きます。

一直線のパイプラインは多くありません。その日の売上を取り込むジョブには、先に為替レートと顧客一覧の両方が必要かもしれませんが、この二つは並行して処理できます。ジョブと、その間の「先に終わる必要がある」という矢印が、有向非巡回グラフを作ります。矢印は一方向を向き、自分自身の完了を待つジョブはありません。こうしたワークフローのスケジュールと監視を行うオープンソースのプラットフォームApache Airflowは、一つのグラフをDagと呼びます。DAGと大文字で書かないのは、Airflowでの意味が、数学的な構造だけではなくなったためです。

バッチ処理のパイプラインは、予定した時刻やイベントをきっかけに起動し、一日分のメーターの読み取り値など、範囲を限定したデータを処理して停止します。ストリーミング処理のパイプラインは動き続け、イベントが届くたびに次へ渡します。Airflowは自身をバッチワークフローの実行を調整するプラットフォームと説明しており、開始と終了が明確で、定期的に実行する仕事に向いています。

最も単純なパイプラインは、午前二時にスクリプトを起動するcrontabの一行です。しかし、複数のスクリプトを順番に動かす必要が生まれ、一つが失敗すると、どこまで動いたのか、何を再開すべきかを誰かが調べなければなりません。オーケストレーターは、その記録と管理を引き受けます。グラフを保持し、上流のジョブが成功してから各ジョブを起動し、失敗したものを再試行し、ジョブご

と、実行ごとのログを残し、それぞれがどの期間のデータを処理したか記録します。Airflowでは、予定された各実行がデータ区間を受け持ちます。1月1日分の日次実行は、その日が終わった2日の午前零時過ぎに始まり、ジョブは `data_interval_start` などの境界値を読んで、担当するデータの範囲を正確に把握できます。

概念を判断に生かす

「パイプラインとは？」を実際の判断に生かしましょう。その概念を、開発または購入するシステムに結び付けて考えるお手伝いをします。 月末に誰かが表計算ファイルをコピーしなくても、決まった時刻に必要な場所へ数値が届く仕組みをつくります。

サービスを見る お問い合わせ →

用語の背後にある前提を検証する

解説とリンク先の実績を参考に、プロジェクトについての質問を整理してください。ツールを選ぶ前に、前提を一緒に検討できます。 14部門を支える共通インフラ計画 — 14 共通インフラ計画の対象となった研究部門

14部門を支える共通インフラ計画 お問い合わせ →

なぜ重要なのか

パイプラインは手順をコードにします。ジョブの順序、途中の検査、失敗したときの動作、実行頻度を、一人の記憶から、ほかのソフトウェアと同じようにレビュー、バージョン管理、テストができるファイルへ移します。Airflowはワークフロー全体をPythonで定義します。この方法を「コードとしてのワークフロー」と呼び、バージョン管理とテストを利点に挙げています。

障害が起きた場所も分かります。全体が一本の長いスクリプトなら、四十分目の異常終了で、零分目からやり直します。状態を記録するジョブに分けると、失敗した実行から、どのジョブがなぜ止まったか分かり、そのジョブと後続だけを動かし直せます。Airflowでは、運用者が失敗したタスクだけを再実行したり、過去の期間の実行を補ったりできます。水曜日の修正を月曜日のデータに適用するのは、この仕組みです。

繰り返しても安全になるのは、それを前提に各ジョブを書いた場合だけです。Airflowのベストプラクティスガイドは、タスクが途中結果を残さないデータベーストランザクションのように動き、「再実行のたびに同じ結果を出す」ことを求めています。そうしたジョブなら、スケジューラーが午前三時に再試行しても、修正後に再実行しても、先月分をやり直しても、先に保存先を誰かが点検する必要はありません。

代わりに、管理するソフトウェアが増えます。オーケストレーターを導入すると、独自のデータベース、アップグレード、権限を持つサービスが一つ増え、そこに置くパイプラインもすべて、レビューとテストが必要なコードです。二本の

スクリプトを同じ予定で実行するだけなら、cronとログファイルが適切な答えかもしれません。依存関係のあるジョブの数、実行結果を見る必要がある人の数、途中で止まった取り込みを基に朝のレポートを作るコストが増えるほど、オーケストレーターを導入する理由は強くなります。

実際の活用例

あるプラットフォームでは、多数のパイプラインを同じ方法で動かすことが課題でした。二十本のGISデータパイプラインとアプリケーションデータのETLを手作業で配信し、監視していたため、一つが気付かれずに止まり、発見する前に下流のデータが誤る危険が常にありました。すべてを自動配信の対象にし、実行予定、ログ、失敗処理を標準化して、同じ動きになり、失敗が見えるようにしました。既存のインフラ自動化とレポートにも組み込みました。その後は二十本すべてが、人の手で一つずつ進めることなく、データを最新に保ちながら、自動的かつ予測可能に動きました。

パイプラインは、一つのコマンドで全体を構築することにも使えます。記録されたある開発段階では、データベース駆動のCV・ポートフォリオ生成ツールが、23テーブルの一つのSQLiteデータベースからすべての文書を作っていました。そのデータベースは、空のファイルから完全なデータベースまでを構築する19ステップのパイプラインで組み立てていました。各ステップには順序があり、それぞれを安全に繰り返せるように書いていたため、既存のデータベースに対して実行しても、行が重複しませんでした。その後、合計10,580行の四十一個のPythonモジュールが、一つのデータ源から六つの出力形式を生成し、実行時依存関係は二つだけでした。この数値は、記録されたその段階の実装を示すものです。

一つのジョブの中では、順序と同じくらい速度も重要です。電力データを一時間ごとに集計するパイプラインは、毎時、15分ごと、または不規則な時刻に届くJSONLデータを取り込み、厳密な一時間の区間に合わせ、発電構成を合計し、輸入を加え、輸出を差し引いて、一時間、一地域につき一行を作ります。Python、JQ、PostgreSQL上のSQLという三つの方法で作り、それぞれ一日分と三十日分のデータでベンチマークしました。最速はPython版で、一日分が34 ms、三十日分が180 msでした。地域を増やしたり、ETLパイプラインへ組み込んだりできるようにコードを書いています。

問題が起きるところ

再実行すると答えが変わります。 毎日のジョブが `INSERT` で集計値を追記し、途中で失敗してスケジューラーが再試行すると、失敗前に書いた行が二重になります。現在時刻の一日前以降のデータを選ぶジョブは、初回と再試行で異なる範囲を読み、入力の最新版を読むジョブは、その間の変更を取り込みます。Airflowのベストプラクティスガイドは、この三つすべてを挙げています。 `INSERT` をupsertに替え、実行の `data_interval_start` をキーとする特定のパーティションを読み書きし、重要な計算には `now()` を使わないことです。PostgreSQLの毎時集計テーブルなら、一つのトランザクションで区間全体を置き換えるのが単純な方法です。

```
BEGIN;
DELETE FROM hourly_load
WHERE hour_start >= :'start' AND hour_start < :'end';
INSERT INTO hourly_load (zone, hour_start, mwh)
SELECT zone, date_trunc('hour', read_at), sum(mwh)
FROM reading
WHERE read_at >= :'start' AND read_at < :'end'
GROUP BY zone, date_trunc('hour', read_at);
COMMIT;
```

オーケストレーターから実行のデータ区間の開始と終了を渡し、 `psql -v ON_ERROR_STOP=1 -v start=... -v end=... -f hourly_load.sql` として動かすと、その区間の行だけを正確に削除して書き直します。二度目の実行後も、初回と同じテーブルになります。 `:'start'` は、変数をSQLリテラルとして引用符で囲んで挿入する、psql独自の構文です。

ジョブは成功しますが、データは間違っています。 オーケストレーターが見ているのは、ジョブのプロセスが正常終了したかどうかだけです。psqlは分かりやすい例で、既定ではスクリプト内のエラー後も処理を続け、シェルへ0を返します。そのため、 `psql -f` を動かす段階は、内部の文が失敗しても成功を報告します。 `ON_ERROR_STOP` を設定すると、最初のエラーで停止し、コード3で終了するので、失敗すべき段階で失敗します。正常終了だけでは、プログラムが動いたことしか分かりません。そこでAirflowのガイドは、後続タスクでパーティションへの書き込みとデータの妥当性を、ほかの処理が読む前に確認する、自己検査を勧めています。異常終了せず止まったままの実行には、Airflow 3.0で削除されたSLA機能に代わって3.1で追加されたDeadline Alertsが、時間のしきい値を超えると、メール送信などのコールバックを起動します。

パイプラインを有効に戻すと、実行が大量に始まります。 `catchup=True` では、スケジューラーが開始日以降に飛ばしたデータ区間ごとに実行を作ります。Airflowのドキュメントによると、Dagをしばらく無効にしてから有効に戻した場合も、この追いつき処理が働きます。毎日のパイプラインを一か月止めると、再開時に同じデータ源に対する三十回分がキューへ入ることがあります。既定の `catchup_by_default=False` なら、最新の区間についてだけ実行を作ります。自身の区間ではなく現在時刻を読むジョブを持つDagは、追いつき処理を無効にすべきだと、ドキュメントは説明しています。過去の分を埋めるなら、 `airflow backfill create` で意図して行う方が適切です。このコマンドは、日付範囲と、 `--max-active-runs` による並行実行数の上限を受け取ります。

Dagファイルのトップレベルにある重いコードが、スケジューラーを遅くします。 Airflowは、少なくとも `min_file_process_interval` 秒の間隔を空けながら、各Dagファイルを繰り返し解析します。そのため、タスクの外にあるファイルのトップレベルのコードは、パイプライン本体よりはるかに頻繁に動きます。そこにあるデータベースクエリー、ネットワーク呼び出し、遅いインポートは、解析のたびに時間を使います。ベストプラクティスガイドには、そうした一つの呼び出しで、Dagの解析に1,000秒が追加される例があります。トップレベルではタスクと順序の宣言だけを行い、コストの大きいインポートと呼び出しは、必要なタスク関数の内部へ移します。

参考資料

- Apache Airflow — Pythonコードとしてのワークフロー、バッチの実行調整、過去の期間の補完、失敗したタスクだけの再実行について、プロジェクト自身が説明しています。
- ベストプラクティス — トランザクションとして扱うタスク、毎回同じ再実行結果、トップレベルのコード、自己検査です。
- Dag — 実行予定、タスク、依存関係、再試行などの既定の引数をDagで宣言する方法と、Deadline Alertsの位置付けです。
- Dagの実行 — データ区間、追いつき処理、過去の期間の補完、タスクを再実行するための状態の消去です。
- タスク — タスクの依存関係、タイムアウト、再試行方針、SLAからDeadline Alertsへの移行です。
- psql — 変数と引用符の扱い、 `ON_ERROR_STOP` 、オーケストレーターが読む終了コードです。



用語集

実際の制約に合わせて選ぶ

代表的なデータセット、そこから答えを得たい問い、現在のボトルネックをお知らせください。品質、処理量、復旧の測り方を合意できます。

料金を見る お問い合わせ →

用語を実務に結び付ける

「パイプラインとは？」がお客様の組織にどう当てはまるかをご相談ください。現在のシステムの具体例が一つあれば、話を始められます。

サービスを見る

welcome@engineer.company +45 50 10 61 84 デンマーク、コペンハーゲン



このページを持ち歩く



連絡先カード

心を込めて丁寧に制作： D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

著作権 © 2025 – 現在 · Engineer ApS · すべての権利を留保します。

印刷元： https://engineer.company/ja/glossary/pipeline/