Engineer

Datapipeline

en följd av steg som för data från källa till användning

Vad det är

En datapipeline, ofta bara kallad pipeline, är en uppsättning jobb som körs i en fast ordning, där varje jobb tar emot det som jobbet före producerade, så att råa poster från de system som skapar dem hamnar rensade, sammanfogade och lagrade där rapporter, dashboards eller andra program kan läsa dem. Namnet kommer från Unix-skalet, där | matar utdata från ett kommando in i nästa. De flesta pipelines körs utan tillsyn, enligt ett schema eller så fort en ny fil kommer in.

Få pipelines är en rak linje. Ett jobb som läser in dagens försäljning kan behöva både valutakurserna och kundlistan först, medan de två kan köras sida vid sida. Jobben och pilarna för ”måste bli klart först” mellan dem bildar en riktad acyklisk graf: varje pil pekar åt ett håll, och ett jobb kan aldrig hamna i att vänta på sig självt. Apache Airflow, en plattform med öppen källkod för att schemalägga och övervaka sådana arbetsflöden, kallar varje graf för en Dag, skrivet så och inte DAG eftersom betydelsen i Airflow har vuxit förbi den rena matematiska strukturen.

En batchpipeline vaknar enligt ett schema eller vid en händelse, bearbetar en avgränsad del av datan, till exempel ett dygns mätaravläsningar, och stannar. En strömmande pipeline fortsätter att köras och skickar vidare varje händelse när den kommer in. Airflow beskriver sig som en plattform för att orkestrera batcharbetsflöden, bäst lämpad för arbete med en tydlig början och ett tydligt slut som körs enligt ett schema.

Den enklaste pipelinen är en rad i en crontab som startar ett skript klockan två på natten. Det håller tills flera skript måste köras i ordning, ett av dem går sönder och någon måste reda ut vilka som kördes och vad som ska startas om. En orkestrerare tar över den bokföringen: den håller grafen, startar varje jobb när

jobben uppströms har lyckats, gör om de jobb som går sönder, sparar en logg per jobb och per körning och noterar vilka perioder av data varje körning täckte. I Airflow täcker varje schemalagd körning ett dataintervall. En daglig körning för den 1 januari startar strax efter midnatt den 2 januari, när dygnet är slut, och dess jobb kan läsa gränser som `data_interval_start` för att veta exakt vilken del av datan som är deras.

Använd begreppet i ett beslut

Använd ”Vad är en datapipeline?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Siffror som kommer fram dit de behövs, på fasta tider, utan att någon kopierar ett kalkylblad i slutet av månaden.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. En infrastrukturplan för 14 avdelningar — 14 forskningsavdelningar under en och samma infrastrukturplan

En infrastrukturplan för 14 avdelningar Hör av er →

Varför det spelar roll

En pipeline gör en rutin till kod. Ordningen mellan jobben, kontrollerna mellan dem, vad som händer när ett går sönder och hur ofta allt körs flyttar ut ur en enskild persons minne och in i filer som kan granskas, versionshanteras och testas som all annan programvara. Airflow definierar sina arbetsflöden helt i Python, ett angreppssätt som projektet kallar workflows as code, och räknar versionshantering och testning bland fördelarna.

Ett fel får också en adress. När hela processen är ett långt skript betyder en krasch vid minut fyrtio att man börjar om från minut noll. Uppdelad i jobb med sparat tillstånd visar en trasig körning vilket jobb som stannade och varför, och bara det jobbet och de efter det behöver köras igen. Airflow låter en operatör köra om enbart de misslyckade uppgifterna, eller fylla i körningar i efterhand för en tidigare period, och det är så en rättning som görs på onsdagen når måndagens data.

Upprepning blir säker först när varje jobb är skrivet för den. Guiden Best Practices i Airflow-dokumentationen ber att varje uppgift beter sig som en databastransaktion som aldrig lämnar halvfärdiga resultat efter sig, och att den ger samma utfall vid varje omkörning. Ett jobb som är byggt så, ett idempotent jobb, kan göras om av schemaläggaren klockan tre på natten, köras om efter en rättning eller spelas upp igen för förra månaden, utan att någon först behöver granska målet.

Priset är programvara att äga. En orkestrerare är ytterligare en tjänst med egen databas, egna uppgraderingar och egna behörigheter, och varje pipeline i den är kod som behöver granskning och tester. För två skript på samma schema kan cron och en loggfil vara det ärliga svaret. Argumenten för en orkestrerare växer med antalet jobb som beror på varandra, antalet personer som behöver se vad som kördes och kostnaden för en morgonrapport som bygger på en inläsning som stannade halvvägs.

Så ser det ut i praktiken

Tjugo pipelines som måste bete sig likadant, en som måste tåla att köras om och en som måste vara snabb ställer tre olika krav på samma sorts kedja.

På en plattform levererades och övervakades tjugo GIS-datapipelines och ETL-flödet för applikationsdata för hand, med en ständig risk att en av dem skulle stanna i det tysta och lämna fel data längre ned i kedjan innan någon märkte det. Alla fick automatiserad leverans, med schemaläggning, loggning och felhantering standardiserade så att varje pipeline betedde sig likadant och så att det syntes när en gick sönder, och de lades in i den befintliga infrastrukturautomatiseringen och rapporteringen. Alla tjugo kördes sedan automatiskt och förutsägbart, med aktuella data och utan att någon behövde lotsa dem igenom för hand.

En pipeline kan också bygga något komplett med ett enda kommando. I ett dokumenterat utvecklingsskede hämtade en databasdriven generator för CV och portfolio varje dokument ur en enda SQLite-databas med 23 tabeller, som byggdes upp av en pipeline i 19 steg som gick från en tom fil till en komplett databas. Stegen följde en fast ordning och vart och ett var skrivet så att det kunde upprepas säkert, så att pipelinen kunde köras mot en befintlig databas utan att någon rad dubblerades. Fyrtioen Python-moduler på sammanlagt 10 580 rader renderade sedan sex utdataformat från den enda källan, med bara två beroenden vid körning. Dessa siffror beskriver implementationen i det dokumenterade utvecklingsskedet.

Inne i ett enskilt jobb spelar hastighet lika stor roll som ordning. En pipeline som aggregerar elförbrukning per timme tar JSONL-data som rapporteras varje timme, var 15:e minut eller vid oregelbundna tidpunkter, justerar dem till strikta timintervall, summerar produktionsmixen, lägger till import och drar av export och producerar en rad per timme och zon. Den byggdes på tre sätt, i Python, i JQ och i SQL på PostgreSQL, och varje variant benchmarkades på ett dygns och

trettio dygns data. Python-varianten var snabbast, med 34 ms för ett dygn och 180 ms för trettio, och koden skrevs så att den kunde byggas ut till fler zoner eller integreras i en ETL-pipeline.

Var det går fel

En omkörning ger ett annat svar. Ett dagligt jobb lägger till sina summor med `INSERT`, går sönder halvvägs, och schemaläggaren gör om det, så att raderna som skrevs före avbrottet nu finns två gånger. Ett jobb som väljer allt sedan aktuell tid minus ett dygn läser en annan del vid omförsöket än vid första försöket, och ett jobb som läser den senaste versionen av sin indata får med sig det som hann ändras däremellan. Guiden Best Practices tar upp alla tre: ersätt `INSERT` med en upsert, läs och skriv en bestämd partition med körningens `data_interval_start` som nyckel, och håll `now()` utanför varje kritisk beräkning. En enkel lösning för en sammanfattningstabell per timme i PostgreSQL är att ersätta hela intervallet i en enda transaktion.

```
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;
```

Körd som `psql -v ON_ERROR_STOP=1 -v start=... -v end=... -f hourly_load.sql`, med början och slutet av körningens dataintervall inskickade av orkestreraren, raderar jobbet och skriver om exakt det intervallets rader, så att en andra körning lämnar tabellen som den första gjorde. Formen `:'start'` är psql-syntax för att infoga en variabel citerad som en SQL-literal.

Ett jobb blir grönt och datan är fel. En orkestrerare ser om processen för ett jobb avslutades rent, och det är allt den ser. Verktyget psql är ett tydligt exempel: som standard fortsätter det efter ett fel i ett skript och returnerar 0 till skalet, så ett steg som kör `psql -f` rapporterar framgång även när en sats i det gick sönder. Med `ON_ERROR_STOP` satt stannar psql vid det första felet och avslutas med slutkod 3, så att steget fallerar där det ska. En ren avslutning bevisar ändå bara att programmet kördes, och därför föreslår guiden självkontroller: en uppgift längre ned i kedjan som bekräftar att partitionen skrevs och att datan ser

rimlig ut innan något läser den. För en körning som hänger sig i stället för att gå sönder utlöser Deadline Alerts, som kom i Airflow 3.1 och ersätter SLA-funktionen som togs bort i 3.0, en callback, till exempel ett mejl, när en körning passerar en tidsgräns.

En pipeline som slås på igen startar en flod av körningar. Med `catchup=True` skapar schemaläggaren en körning för varje dataintervall som den hoppade över sedan startdatumet, och dokumentationen för Airflow tillägger att catchup också utlöses när en Dag stängs av en tid och sedan aktiveras igen. En daglig pipeline som pausats i en månad kan komma tillbaka med trettio körningar som väntar mot samma källa. Standardvärdet, `catchup_by_default=False`, gör att schemaläggaren bara skapar en körning för det senaste intervallet, och dokumentationen säger att en Dag vars jobb läser aktuell tid i stället för sitt intervall bör ha catchup avstängt. Historik fylls hellre i med avsikt genom `airflow backfill create`, som tar ett datumintervall och en gräns för parallella körningar genom `--max-active-runs`.

Tung kod överst i en Dag-fil gör schemaläggaren långsam. Airflow tolkar varje Dag-fil om och om igen, med minst `min_file_process_interval` sekunder mellan tolkningarna, så kod på översta nivån i filen, utanför alla uppgifter, körs långt oftare än själva pipelinen. En databasfråga, ett nätverksanrop eller en långsam import där kostar tid vid varje tolkning, och guiden Best Practices visar ett exempel där ett enda sådant anrop gör att en Dag tar ytterligare 1 000 sekunder att tolka. Begränsa översta nivån till att deklarera uppgifter och deras ordning, och flytta dyra importer och anrop in i de uppgiftsfunktioner som behöver dem.

Läs mer

- Apache Airflow — projektets egen översikt: arbetsflöden som Python-kod, orkestrering av batcher, backfill och omkörning av enbart de misslyckade uppgifterna.
- Best Practices — uppgifter som behandlas som transaktioner, samma utfall vid varje omkörning, kod på översta nivån och självkontroller.
- Dags — hur en Dag deklarerar sitt schema, sina uppgifter, beroenden och standardargument som omförsök, och var Deadline Alerts passar in.
- Dag Runs — dataintervall, catchup, backfill och rensning av uppgifter så att de körs igen.
- Tasks — beroenden mellan uppgifter, tidsgränser, policyer för omförsök och övergången från SLA till Deadline Alerts.
- psql — variabler och hur de citeras, `ON_ERROR_STOP` och de slutkoder som en orkestrerare läser.

Ordlista

Välj utifrån era faktiska förutsättningar

Ta med ett representativt dataset, frågorna det ska besvara och dagens flaskhals. Vi kan avtala mått på kvalitet, genomströmning och återställning.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är en datapipeline?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

welcome@engineer.company +45 50 10 61 84 Köpenhamn, Danmark

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/pipeline/