Engineer

Pipeline

en række trin der fører data fra kilde til brug

Hvad det er

En pipeline, fuldt ud en datapipeline, er en kæde af job, der kører i fast rækkefølge, hvor hvert job arbejder videre på det, jobbet før det afleverede, indtil rådata fra de systemer, der skaber dem, ligger renset, samlet og gemt dér, hvor rapporter, dashboards og andre programmer kan læse dem. Ordet stammer fra Unix-shellen, hvor tegnet | sender én kommandos output videre som den næstes input. De fleste pipelines kører uden opsyn, efter en tidsplan eller når en ny fil dukker op.

Sjældent er en pipeline en lige linje. Et job, der indlæser dagens salg, kan have brug for både valutakurserne og kundelisten først, mens de to sagtens kan køre samtidig. Jobbene og pilene imellem dem, der betyder »skal være færdig før«, danner en rettet acyklisk graf: alle pile peger én vej, og intet job kan ende med at vente på sig selv. Apache Airflow, en open source-platform til at planlægge og overvåge den slags arbejdsgange, kalder hver graf en Dag, skrevet sådan frem for DAG, fordi begrebet i Airflow er vokset ud over den rene matematiske struktur.

En batch-pipeline vågner på et fast tidspunkt eller ved en hændelse, behandler et afgrænset udsnit af data, for eksempel én dags målerdata, og stopper igen. En streaming-pipeline kører hele tiden og sender hver hændelse videre, efterhånden som den kommer. Airflow beskriver sig selv som en platform til at orkestrere batch-arbejdsgange, bedst egnet til arbejde med en klar start og slutning, der kører efter en tidsplan.

Den enkleste pipeline er en linje i en crontab, der starter et script klokken to om natten. Det holder, så længe scriptene er uafhængige. Skal flere af dem køre i rækkefølge, og fejler ét af dem, må nogen finde ud af, hvad der nåede at køre, og hvad der skal startes igen. En orkestrator overtager det bogholderi: den kender

grafen, starter hvert job, når jobbene før det er lykkedes, prøver igen, når et job fejler, fører log pr. job og pr. kørsel og noterer, hvilke perioder af data hver kørsel dækkede. I Airflow dækker hver planlagt kørsel et datainterval. En daglig kørsel for 1. januar starter lige efter midnat den 2., når dagen er omme, og dens job kan læse grænser som `data_interval_start` og vide præcis, hvilket udsnit af data der er deres.

Brug begrebet i en beslutning

Brug »Hvad er en pipeline?« i en konkret beslutning. Vi kan knytte begrebet til det system, I bygger eller køber. Tal, der lander, hvor der er brug for dem, efter en fast plan, uden at nogen kopierer et regneark ved månedens slutning.

Se vores ydelser Kontakt os →

Undersøg antagelserne bag ordet

Brug forklaringen og det tilknyttede arbejde til at formulere spørgsmål til jeres projekt. Vi kan undersøge antagelserne sammen, før et værktøj vælges. Én infrastrukturplan til 14 afdelinger — 14 forskningsafdelinger på én infrastrukturplan

Én infrastrukturplan til 14 afdelinger Kontakt os →

Hvorfor det betyder noget

En pipeline gør en fremgangsmåde til kode. Rækkefølgen af job, kontrollerne imellem dem, hvad der sker, når et af dem fejler, og hvor ofte det hele kører, flytter ud af én persons hukommelse og ind i filer, der kan gennemgås, versionsstyres og testes som al anden software. Airflow definerer sine arbejdsgange udelukkende i Python, en tilgang projektet kalder workflows as code, og nævner versionsstyring og test blandt fordelene.

En fejl får også en adresse. Er hele processen ét langt script, betyder et nedbrud efter fyrre minutter, at man starter forfra fra minut nul. Deles den op i job med registreret tilstand, viser en mislykket kørsel, hvilket job der stoppede og hvorfor, og kun det job og dem efter det skal køre igen. Airflow lader en operatør køre kun de fejlede opgaver igen eller lave backfill for en tidligere periode, og sådan når en rettelse fra onsdag frem til mandagens data.

Gentagelse bliver først sikker, når hvert job er skrevet til det. Airflows vejledning i god praksis beder om, at hver opgave opfører sig som en databasetransaktion, der aldrig efterlader halve resultater, og giver samme udfald ved hver genkørsel. Et job bygget sådan kan scheduleren prøve igen klokken tre om natten, en person kan køre det igen efter en rettelse, og det kan genafspilles for sidste måned, uden at nogen først skal undersøge målet.

Prisen er software, der skal passes. En orkestrator er én tjeneste mere med egen database, opgraderinger og rettigheder, og hver pipeline i den er kode, der skal gennemgås og testes. Til to scripts på én tidsplan kan cron og en logfil være det ærlige svar. Argumentet for en orkestrator vokser med antallet af job, der afhænger af hinanden, antallet af mennesker, der skal kunne se, hvad der kørte, og prisen for en morgenrapport bygget på en indlæsning, der stoppede halvvejs.

Sådan ser det ud i praksis

Tyve GIS-datapipelines og ETL-processerne for appdata på én platform blev startet og fulgt manuelt, og der var hele tiden en risiko for, at én af dem gik i stå i det stille, så data længere nede i kæden var forkerte, før nogen opdagede det. Alle sammen blev lagt over på automatiseret levering. Tidsplaner, logning og fejlhåndtering fik én fælles standard, så pipelines opførte sig ens, og en, der fejlede, kunne ses, og de indgik derefter i den infrastrukturautomatisering og rapportering, der allerede fandtes. Herefter kørte alle tyve af sig selv og forudsigeligt, og dataene var aktuelle og til at stole på, uden at nogen skulle skubbe dem igennem.

En pipeline kan også bygge noget helt med én kommando. På et dokumenteret stadium i udviklingen hentede en databasedrevet generator af CV og portefølje hvert dokument fra én SQLite-database med 23 tabeller, som blev samlet af en pipeline på 19 trin, der gik fra en tom fil til en færdig database. Trinnene fulgte en fast rækkefølge, og hvert af dem var skrevet, så det kunne køres sikkert igen. Derfor kunne hele pipelinen køres mod en database, der allerede fandtes, uden at en eneste række blev duplikeret. Ud fra den ene kilde lavede 41 Python-moduler på tilsammen 10.580 linjer derefter seks outputformater, og programmet havde kun to afhængigheder under kørsel. Disse tal beskriver implementeringen på det dokumenterede udviklingsstadium.

Inde i et enkelt job tæller hastighed lige så meget som rækkefølge. En pipeline, der samler elforbrug time for time, tager JSONL-data, der kommer hver time, hvert kvarter eller på uregelmæssige tidspunkter, lægger dem til rette i faste timeintervaller, summerer produktionsmikset, lægger import til og trækker eksport fra og afleverer én række pr. time pr. zone. Den blev lavet på tre måder, i Python, i JQ og i SQL i PostgreSQL, og hver udgave blev målt på data for én dag og for 30 dage. Python-udgaven var hurtigst, med 34 ms for én dag og 180 ms for 30 dage, og koden kunne let udvides til flere zoner eller bygges ind i en ETL-pipeline.

Hvor det går galt

En genkørsel giver et andet resultat. Et dagligt job tilføjer sine totaler med `INSERT`, går ned halvvejs, og scheduleren prøver igen, så de rækker, der blev skrevet før nedbruddet, nu findes to gange. Et job, der vælger alt fra nu minus et døgn, læser et andet udsnit ved andet forsøg end ved det første, og et job, der læser den seneste udgave af et input, får det med, der er ændret i mellemtiden. Airflows vejledning i god praksis nævner alle tre: erstat `INSERT` med en upsert, læs og skriv en bestemt partition ud fra kørslens `data_interval_start`, og hold `now()` ude af enhver kritisk beregning. For en tabel med timetotaler i PostgreSQL er én ligetil løsning at udskifte hele intervallet i én 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;
```

Kørt som `psql -v ON_ERROR_STOP=1 -v start=... -v end=... -f hourly_load.sql`, hvor orkestratoren sender start og slutning på kørslens datainterval med, sletter og genskriver jobbet præcis det intervals rækker, så en anden kørsel efterlader tabellen, som den første gjorde. Formen `:'start'` er psql's egen syntaks for at indsætte en variabel i anførselstegn som en SQL-literal.

Et job ender grønt, og dataene er forkerte. En orkestrator ser, om et jobs proces sluttede rent, og det er alt, den ser. psql er et skarpt eksempel: som standard fortsætter den efter en fejl i et script og returnerer 0 til shellen, så et trin, der kører `psql -f`, melder succes, selv om en sætning undervejs fejlede. Med `ON_ERROR_STOP` sat stopper psql ved første fejl og afslutter med kode 3, så trinnet fejler, hvor det skal. En ren afslutning beviser stadig kun, at programmet kørte, og derfor foreslår Airflows vejledning selvtjek: en opgave længere nede i grafen, der bekræfter, at partitionen blev skrevet, og at dataene ser rigtige ud, før noget læser dem. For en kørsel, der hænger i stedet for at fejle, findes Deadline Alerts, som kom i Airflow 3.1 og afløser SLA-funktionen, der blev fjernet i 3.0. De udløser et callback, for eksempel en e-mail, når en kørsel passerer en tidsgrænse.

En pipeline, der slås til igen, sender en strøm af kørsler afsted. Med `catchup=True` opretter scheduleren en kørsel for hvert datainterval, den har sprunget over siden startdatoen, og Airflow-dokumentationen tilføjer, at catchup også sætter ind, når en Dag har været slået fra en tid og derefter slås til igen. En daglig pipeline, der har stået på pause i en måned, kan vende tilbage med tredive kørsler i kø mod den samme kilde. Standarden, `catchup_by_default=False`, får scheduleren til kun at oprette en kørsel for det seneste interval, og dokumentationen siger, at en Dag, hvis job læser det aktuelle tidspunkt frem for deres interval, bør holde catchup slået fra. Historik fyldes bedre op med vilje med `airflow backfill create`, som tager et datointerval og et loft over parallelle kørsler via `--max-active-runs`.

Tung kode øverst i en Dag-fil gør scheduleren langsom. Airflow parser hver Dag-fil igen og igen, med mindst `min_file_process_interval` sekunder mellem gangene, så kode på filens øverste niveau, uden for enhver opgave, kører langt oftere end selve pipelinen. En databaseforespørgsel, et netværkskald eller en langsom import dér koster tid ved hver parsing, og vejledningen i god praksis viser et eksempel, hvor ét sådant kald får en Dag til at bruge 1.000 sekunder ekstra på at blive parset. Hold det øverste niveau til at erklære opgaver og deres rækkefølge, og flyt dyre importer og kald ind i de opgavefunktioner, der bruger dem.

Læs mere

- Apache Airflow — projektets eget overblik: arbejdsgange som Python-kode, orkestrering af batch, backfill og genkørsel af kun de fejlede opgaver.
- Best Practices — opgaver behandlet som transaktioner, samme udfald ved hver genkørsel, kode på øverste niveau og selvtjek.
- Dags — hvordan en Dag erklærer sin tidsplan, sine opgaver, afhængigheder og standardargumenter som retries, og hvor Deadline Alerts hører hjemme.
- Dag Runs — dataintervaller, catchup, backfill og rydning af opgaver, så de kører igen.
- Tasks — afhængigheder mellem opgaver, timeouts, politikker for nye forsøg og skiftet fra SLA'er til Deadline Alerts.
- psql — variabler og deres citering, `ON_ERROR_STOP` og de exitkoder, en orkestrator læser.

Ordliste

Vælg ud fra jeres faktiske vilkår

Medbring et repræsentativt datasæt, de spørgsmål det skal besvare, og den aktuelle flaskehals. Vi kan aftale målinger af kvalitet, gennemløb og gendannelse.

Se, hvad det koster Kontakt os →

Gør begreberne praktiske

Spørg, hvordan »Hvad er en pipeline?« gælder i jeres organisation. Et konkret eksempel fra jeres nuværende system er nok til at begynde samtalen.

Se vores ydelser

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

Tag denne side med dig

Visitkort

Lavet med kærlighed af D.

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

Copyright © 2025 – nu · Engineer ApS · Alle rettigheder forbeholdes.

Udskrevet fra https://engineer.company/da/glossary/pipeline/