O'REILLY®

Wzorce projektowe w inżynierii danych

Sprawdzone rozwiązania i dobre praktyki

Helion

Bartosz Konieczny

Tytuł oryginału: Data Engineering Design Patterns:
Recipes for Solving the Most Common Data Engineering Problems

Tłumaczenie: Jacek Janusz

ISBN: 978-83-289-3179-4

© 2026 Helion S.A.

Authorized Polish translation of the English edition of *Data Engineering Design Patterns*
ISBN 9781098165819 © 2025 Bartosz Konieczny.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/wzprin
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

ROZDZIAŁ 2.

Wzorce projektowe pozyskiwania danych

Systemy inżynierii danych rzadko same generują dane. Najczęściej ich pierwszym etapem działania jest pozyskiwanie danych z różnych źródeł. Współpraca z tymi źródłami bywa trudna — mogą to być inne potoki danych w Twoim zespole, zespoły w obrębie firmy, a nawet całkowicie odrębne organizacje. Każde źródło wiąże się z własnymi ograniczeniami technicznymi i biznesowymi, co czyni interakcję z nimi szczególnie wymagającą.

Nie ma jednak alternatywnej opcji — trzeba się dostosować. W przeciwnym razie nie otrzymasz żadnych danych, a Twoje procesy analityczne lub naukowe nie będą mogły zostać uruchomione. Co gorsza, możesz otrzymać pewien zbiór danych, udostępnić go odbiorcom, a po kilku dniach spotkać się ze skargami. Ich przyczyną może być niekompletność danych, ich nieefektywna organizacja albo wręcz uszkodzenie zbioru, które wymaga wewnętrznych procesów odtwarzania i uzupełniania.

Jak widać, doprowadzenie danych do systemu to zadanie o kluczowym znaczeniu — zarówno dla Ciebie, jak i dla Twoich użytkowników. Z tego względu książkę otwiera analiza wzorców projektowych związanych z pozyskiwaniem danych.

Wzorce przedstawione w tym rozdziale odnoszą się do scenariuszy i wyzwań, jakie mogą wystąpić podczas integracji danych pochodzących od zewnętrznych dostawców lub z innych potoków danych. Rozpoczniemy od dwóch typowych procesów wczytywania danych — pełnego oraz przyrostowego — wykorzystywanych do pozyskiwania całości lub części zbioru danych. Następnie przedstawimy szczególny przypadek pozyskiwania danych, czyli replikację. W tej części znajdziesz dwa inne wzorce kopiowania danych — z transformacją oraz bez niej — które mogą pomóc rozwiązać problemy związane z prywatnością.

Ponieważ pozyskiwanie danych obejmuje również zagadnienia wykraczające poza samo ich przenoszenie, poruszymy także aspekty techniczne. Po pierwsze, dobrze wiedzieć, kiedy należy rozpocząć proces pozyskiwania danych — w tym pomoże fragment o gotowości danych. Po drugie, warto się również dowiedzieć, jak poprawić doświadczenia użytkowników oraz jak sobie poradzić z jednym z najczęstszych wyzwań w inżynierii danych, czyli małymi plikami. W tym kontekście przydatny będzie punkt dotyczący kompresji danych. Na zakończenie rozdziału pokażemy, że pozyskiwanie danych nie zawsze jest procesem przewidywalnym. Wzorzec Wyzwalacz Zewnętrzny, który przedstawimy na końcu, pozwoli lepiej sobie radzić z tą niepewnością.

Ponieważ wiesz już, czego należy oczekiwać, czas poznać pierwsze wzorce pozyskiwania danych — dla scenariuszy wczytywania pełnego oraz przyrostowego!

Kup książkę Poleć książkę

Wczytywanie pełne

Wzorzec wczytywania pełnego (ang. *full load*) odnosi się do scenariusza pobierania danych, w którym za każdym razem jest przetwarzany cały zestaw danych. Może być przydatny w wielu sytuacjach, takich jak inicjalizacja bazy danych czy generowanie *zbioru danych referencyjnych*.

Wzorzec: Wczytywanie Pełne

Implementacja wzorca Wczytywanie Pełne jest jednym z najprostszych zagadnień przedstawionych w tej książce. Jednak mimo swojej prostej, dwuetapowej konstrukcji ma ona pewne pułapki.

Problem

Dla naszego przypadku użycia przygotowujesz warstwę Silver. Jedno z zadań transformacji wymaga uzyskania dodatkowych informacji o urządzeniu od zewnętrznego dostawcy danych. Zbiór danych związany z tym urządzeniem zmienia się tylko kilka razy w tygodniu. Jest to również bardzo wolno zmieniający się zestaw, w którym całkowita liczba wierszy nie przekracza miliona. Niestety, dostawca danych nie udostępnia żadnego atrybutu, który pozwalałby wykryć wiersze zmienione od czasu ostatniego wczytania danych.

Rozwiązanie

Brak informacji o czasie ostatniej aktualizacji w zbiorze danych sprawia, że idealnym rozwiązaniem tego problemu jest zastosowanie wzorca pełnego wczytywania.

Najprostsza implementacja opiera się na dwóch krokach: wyodrębnieniu i wczytaniu (EL). W tym podejściu wykorzystuje się natywne polecenia baz danych do eksportowania danych z jednej bazy i ich importowania do drugiej. Metoda EL jest idealna dla jednorodnych baz danych, ponieważ nie wymaga żadnych transformacji danych.

Zadania przelotowe

Zadania wyodrębniania i wczytywania są również znane jako **zadania przelotowe**, ponieważ dane po prostu przechodzą przez potok — od źródła do miejsca docelowego.

Niestety, stosowanie potoków EL nie zawsze jest możliwe. Jeśli trzeba przenosić dane między różnorodnymi bazami danych, konieczne jest dostosowanie formatu danych wejściowych do wymagań systemu docelowego za pomocą cienkiej warstwy transformacji pomiędzy etapami wyodrębniania i wczytywania. W takim przypadku potok przekształca się w zadanie wyodrębniania, transformacji i wczytywania (ETL), w którym można wykorzystać bibliotekę do przetwarzania danych, często wyposażoną w natywne interfejsy do komunikacji z różnymi bazami danych.

Kup książkę Poleć książkę

Następstwa

Mimo że przenoszenie zbioru danych między dwoma magazynami danych może się wydawać prostym zadaniem, stosowanie wzorca wczytywania pełnego wiąże się z pewnymi wyzwaniami.

Objętość danych. Implementacje wzorca wczytywania pełnego często mają postać zadań wsadowych uruchamianych według ustalonego harmonogramu. Jeśli objętość wczytywanego zbioru danych rośnie powoli, infrastruktura do wczytywania danych prawdopodobnie będzie działać stabilnie przez długi czas, ponieważ zapotrzebowanie na moc obliczeniową pozostanie niemal niezmienne.

Z drugiej strony, bardziej dynamicznie rozwijający się zbiór danych może powodować problemy, zwłaszcza jeśli do jego przetwarzania wykorzystujesz statyczne zasoby obliczeniowe. Na przykład, gdy rozmiar zbioru danych z dnia na dzień się podwoi, proces wczytywania może się wydłużyć, a nawet zakończyć niepowodzeniem z powodu ograniczeń sprzętowych.

Aby ograniczyć wpływ zmienności danych na proces wczytywania, możesz skorzystać z możliwości automatycznego skalowania warstwy przetwarzania danych.

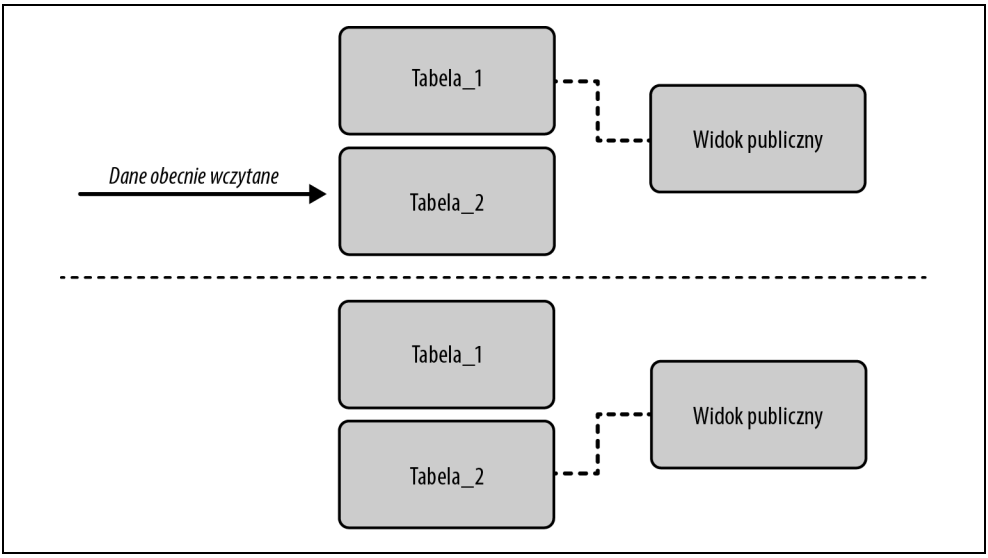
Spójność danych. Drugim ryzykiem związanym ze wzorcem wczytywania pełnego jest możliwość utraty spójności danych. Ponieważ dane są całkowicie nadpisywane, może pojawić się pokusa, by w każdym przebiegu usuwać je i wstawiać na nowo. Decydując się na takie podejście, trzeba być świadomym jego konsekwencji.

Po pierwsze, należy wziąć pod uwagę spójność danych z perspektywy konsumenta. Co się stanie, jeśli proces wczytywania zostanie uruchomiony w tym samym czasie, gdy potoki próbują odczytać zbiór danych? Jeśli krok wstawiania nie został jeszcze zakończony, konsumenci mogą przetwarzać niekompletne dane albo w ogóle nie mieć do nich dostępu. Transakcje automatycznie zarządzają widocznością danych i są najprostszym sposobem rozwiązania tego problemu współbieżności. Jeżeli jednak Twój magazyn danych nie obsługuje transakcji, możesz zastosować **pojedynczą abstrakcję udostępniania danych**[1], taką jak widok, i manipulować wyłącznie ukrytymi strukturami technicznymi. Na rysunku 2.1 jest przedstawiony sposób przełączania między dwiema technicznymi tabelami, który pozwala zachować ciągłą dostępność danych.

Po drugie, pamiętaj, że w razie wystąpienia nieoczekiwanych problemów być może będzie trzeba skorzystać z poprzedniej wersji zbioru danych. Jeśli nadpiszesz go w całości, możesz utracić możliwość przywrócenia wcześniejszej wersji — chyba że używasz formatu obsługującego funkcję „podróży w czasie", takiego jak Delta Lake, Apache Iceberg czy BigQuery w Google Cloud Platform (GCP). Alternatywnie możesz samodzielnie zaimplementować taką funkcjonalność, opierając się na koncepcji pojedynczej abstrakcji udostępniania danych, pokazanej na rysunku 2.1.

[1] Więcej informacji na ten temat znajdziesz w podrozdziale poświęconym wzorcowi Pełnomocnik w rozdziale 4.

Kup książkę Poleć książkę



Rysunek 2.1. Przykład pojedynczej abstrakcji udostępniania danych z użyciem widoku bazy danych

Nie tylko pozyskiwanie danych

Mimo że w tym rozdziale omawiamy pobieranie danych, pamiętaj, że wszystkie przedstawione tu wzorce mają bezpośredni wpływ na zadania związane z analizą danych i danetyką, ponieważ to one wprowadzają dane do systemu.

Przykłady

Zobaczmy teraz, jak zaimplementować ten wzorzec w różnych kontekstach technicznych. Po pierwsze, jeśli musisz przenieść zbiór danych między dwoma identycznymi lub kompatybilnymi magazynami danych, możesz po prostu napisać skrypt i wdrożyć go w swoim środowisku uruchomieniowym. Na listingu 2.1 pokazano, jak przenosić pliki z jednego folderu Amazon S3 do drugiego. Polecenie automatycznie synchronizuje zawartość folderów i usuwa wszystkie obiekty, których brakuje w źródle, ale które są obecne w miejscu docelowym (argument `--delete`).

Listing 2.1. Synchronizacja folderów

```
aws s3 sync s3://input-bucket s3://output-bucket --delete
```

Polecenia takie jak `aws s3 sync` są świetnym sposobem na proste przenoszenie zbiorów danych, ale czasami operacja wczytywania może wymagać pewnych modyfikacji, takich jak dodanie przetwarzania równoległego lub rozproszonego. Przykładem takiej implementacji jest Apache Spark.

Apache Spark — biblioteka do rozproszonego przetwarzania danych, może być płynnie skalowana, dzięki czemu nawet drastyczne zmiany w objętości danych nie powinny negatywnie wpływać na ich wczytywanie, o ile odpowiednio zwiększymy moc obliczeniową infrastruktury.

Kup książkę Poleć książkę

Ponadto, jeśli używamy tej biblioteki z formatem tabel takim jak Delta Lake, dzięki możliwościom transakcyjnym i wersjonowaniu automatycznie rozwiązujemy przedstawione wcześniej problemy spójności. Wisienką na torcie jest to, że kod służący do pełnego wczytywania danych jest dość prosty. Na listingu 2.2 pokazano, jak wykorzystać API odczytu i zapisu z biblioteki Apache Spark do zapisywania rekordów JSON jako tabeli Delta Lake.

Listing 2.2. Implementacja procesu wczytywania danych z użyciem Apache Spark i Delta Lake

```
input_data = spark.read.schema(input_data_schema).json("s3://devices/list")
input_data.write.format("delta").save("s3://master/devices")
```

Możesz również zaimplementować ten wzorzec dla baz danych, które nie mają wbudowanej obsługi wersjonowania. Weźmy jako przykład Apache Airflow i PostgreSQL. Implementacja jest bardziej złożona, ponieważ wymaga oddzielnych zadań do wprowadzania i udostępniania danych.

Zadanie wprowadzania danych zapisuje zbiór danych do jawnie wersjonowanej tabeli. Odpowiada to instrukcji `COPY` z listingu 2.3, w którym `${version}` jest parametrem dostarczanym przez operator Apache Airflow.

Listing 2.3. Wczytywanie danych do tabeli z obsługą wersjonowania

```
COPY devices_${version} FROM '/data_to_load/dataset.csv' CSV DELIMITER ';' HEADER;
```

Zadanie udostępniania zmienia następnie referencję widoku prezentowanego użytkownikom końcowym. W tym celu może wykorzystać operację aktualizacji widoku przedstawioną na listingu 2.4.

Listing 2.4. Publiczne udostępnianie wersjonowanej tabeli

```
CREATE OR REPLACE VIEW devices AS SELECT * FROM devices_${version}
```

Proces może wymagać wykonania dodatkowych kroków, takich jak pobranie zbioru danych wejściowych i utworzenie tabel z kontrolą wersji. W celu zachowania zwięzłości pomijam je tutaj, ale pełny przykład znajdziesz w repozytorium GitHuba (*https://oreil.ly/eAG5j*).

Wczytywanie przyrostowe

Pełne wczytywanie danych to prosty scenariusz, jednak w przypadku ciągle rosnących zbiorów danych może się on okazać kosztowny. Lepszym rozwiązaniem jest wczytywanie przyrostowe, które polega na przetwarzaniu (często z większą częstotliwością) mniejszych fragmentów fizycznie lub logicznie podzielonego zbioru danych.

Wzorzec: Wczytywanie Przyrostowe

Pierwszy wzorzec projektowy stopniowego przetwarzania danych przetwarza nowe części zbioru danych, stąd jego nazwa — Wczytywanie Przyrostowe.

Kup książkę Poleć książkę

Problem

Większość zdarzeń związanych z odwiedzinami bloga pochodzi z brokera strumieniowego, który działa w czasie rzeczywistym. Pewna ich część wciąż trafia jednak do transakcyjnej bazy danych, obsługiwanej przez starsze systemy.

Aby można było uwzględnić te dane w analizie, potrzebny jest specjalny proces pozyskiwania, który przekaże starsze zdarzenia odwiedzin do warstwy Bronze. Ponieważ liczba odwiedzin stale rośnie, proces ten powinien obejmować wyłącznie nowe dane, dodane od ostatniego uruchomienia. Warto przy tym podkreślić, że każde zdarzenie odwiedzin jest trwałe i niezmienne.

Rozwiązanie

Ciągle rosnący zbiór danych to typowy przypadek, w którym warto zastosować wzorzec wczytywania przyrostowego. Istnieją dwie główne metody jego implementacji — wybór odpowiedniego podejścia zależy od struktury danych wejściowych:

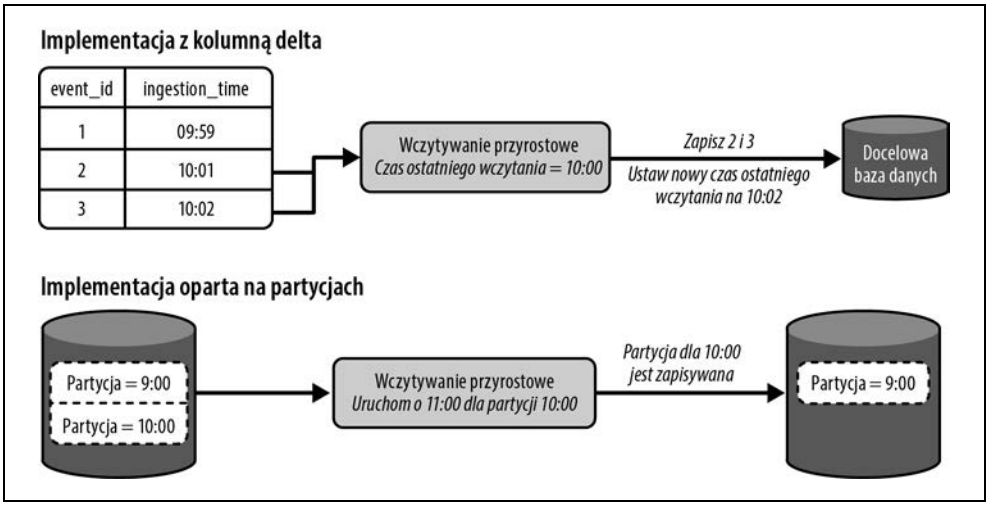
- Pierwsze podejście opiera się na wykorzystaniu tak zwanej **kolumny delta**, która umożliwia identyfikację rekordów dodanych od ostatniego uruchomienia procesu. W przypadku danych zdarzeniowych, takich jak niezmienne wizyty użytkowników, kolumna ta zazwyczaj zawiera znacznik czasu pobrania.
- Drugie podejście zakłada, że dane są już fizycznie podzielone na *partycje czasowe*. W takim scenariuszu zadanie pozyskiwania danych korzysta z tych partycji do wykrywania nowych rekordów. Dzięki temu dane są już wstępnie odfiltrowane i logicznie zorganizowane w warstwie przechowywania, co znacząco upraszcza i przyspiesza cały proces. Aby się upewnić, że dana partycja jest gotowa do pobrania, można zastosować wzorzec o nazwie Znacznik Gotowości.

Uwaga na problemy związane z czasem rzeczywistym

Używanie *czasu zdarzenia* jako kolumny delta jest ryzykowne. Twój proces pozyskiwania danych może pominąć niektóre rekordy, jeśli producent danych emituje dane z opóźnieniem (patrz dalszy podrozdział „Dane opóźnione") dla czasu zdarzenia, który został już przetworzony.

Obie implementacje są zilustrowane na rysunku 2.2. Jak można zauważyć, podejście oparte na kolumnie delta wymaga zapamiętania ostatniej wartości czasu pobrania, aby kod mógł przyrostowo przetwarzać nowe wiersze. Z kolei implementacja oparta na partycjach nie stawia takiego wymogu — może automatycznie wyznaczyć partycję do przetworzenia na podstawie daty uruchomienia. Na przykład, jeśli proces wczytywania danych rozpocznie się o 11:00, może od razu się skierować do partycji odpowiadającej godzinie 10:00.

Kup książkę Poleć książkę



Rysunek 2.2. Dwie możliwe implementacje wzorca Wczytywanie Przyrostowe

Następstwa

Wczytywanie przyrostowe sprzyja zmniejszeniu objętości pobieranych danych, ale może też stanowić wyzwanie.

Twarde usuwanie. Stosowanie tego wzorca może być problematyczne w przypadku danych zmieniających się. Załóżmy, że zamiast niezmiennych odwiedzin musisz obsługiwać zdarzenia, które mogą być aktualizowane lub usuwane. Jeśli proces wczytywania danych opiera się na kolumnie delta, jest w stanie wykryć zaktualizowane wiersze i skopiować ich najnowszą wersję. Niestety, w przypadku wierszy usuniętych sytuacja nie jest już tak prosta.

Gdy dostawca danych usuwa wiersz, informacja fizycznie znika ze zbioru wejściowego. Nadal jest jednak obecna w Twojej wersji zbioru, ponieważ dla usuniętego wiersza *kolumna delta* nie istnieje. Aby rozwiązać ten problem, można zastosować **miękkie usuwanie**, w przypadku którego producent zamiast fizycznie usuwać dane, po prostu oznacza je jako usunięte. Innymi słowy, używa operacji `UPDATE` zamiast `DELETE`.

Tabele tylko do wstawiania

Innym rozwiązaniem problemu zmienności mogą być zbiory danych *tylko do wstawiania*. Jak sugeruje nazwa, akceptują one wyłącznie nowe wiersze poprzez operację `INSERT`. Przenosi to odpowiedzialność za rekonstrukcję danych na konsumentów, którzy muszą poprawnie wykrywać wszelkie usunięte i zmodyfikowane wpisy. Tabele tylko do wstawiania są znane również pod nazwą tabeli tylko do dopisywania.

Uzupełnianie brakujących informacji. Nawet tak podstawowe zadania związane z pozyskiwaniem danych wiążą się z ryzykiem, że będzie konieczne uzupełnienie brakujących informacji. W takim scenariuszu wzorzec może zaskakująco negatywnie wpłynąć na działanie potoków wczytywania danych.

Kup książkę Poleć książkę

Wyobraź sobie potok oparty na implementacji wykorzystującej kolumnę delta. Po przetworzeniu danych z dwóch miesięcy otrzymujesz polecenie rozpoczęcia uzupełniania brakujących informacji. W takiej sytuacji uruchomiony proces wczytywania danych wykona pełne wczytanie zamiast przyrostowego. Oznacza to, że zadanie będzie potrzebować większych zasobów, aby obsłużyć dodatkowe wiersze.

Na szczęście problem ten można złagodzić przez ograniczenie okna czasowego wczytywania. Na przykład, jeśli zadanie wczytywania jest uruchamiane co godzinę, można ograniczyć je tylko do danych z tej jednej godziny. W SQL można to wyrazić jako `delta_column BETWEEN ingestion_time AND ingestion_time + INTERVAL '1 HOUR'`. Ta operacja przynosi dwie korzyści:

- Lepszą kontrolę nad objętością danych. Nawet w przypadku uzupełniania brakujących informacji nie zaskoczy Cię zwiększone zapotrzebowanie na zasoby obliczeniowe.
- Równoczesne wczytywanie danych. Możesz uruchomić wiele jednoczesnych zadań uzupełniania brakujących informacji, o ile źródłowy magazyn danych to umożliwia.

Problem z rozmiarem zbioru danych nie występuje w implementacji opartej na partycjach, jeśli zadanie wczytywania danych przetwarza tylko jedną partycję naraz.

Przykłady

Przykład oparty na skrypcie używanym we wzorcu wczytywania pełnego można również zastosować w przypadku wczytywania przyrostowego. Operacja z listingu 2.5 po prostu przenosi wszystkie obiekty z prefiksem `date=2024-01-01` do innego folderu. Choć działanie kodu może się wydawać proste, istnieje pewna pułapka. Jeśli po prawej stronie polecenia pominiesz prefiks `date=2024-01-01`, zadanie importu spłaszczy strukturę przechowywania danych wyjściowych.

Listing 2.5. Synchronizacja folderów S3

```
aws s3 sync s3://input/date=2024-01-01 s3://output/date=2024-01-01 --delete
```

Czasami implementacja musi być czymś więcej niż tylko prostym skryptem. Podobnie jak w przypadku pełnego wczytywania danych, można się oprzeć na warstwach przetwarzania i orkiestracji. Na listingu 2.6 przedstawiono implementację opartą na partycjach, zrealizowaną z użyciem platform Apache Airflow i Apache Spark. Proces w Apache Airflow (najczęściej określany skrótowcem DAG — skierowany graf acykliczny) rozpoczyna się od wywołania funkcji `FileSensor`, która oczekuje na dostępność kolejnej partycji. Ten krok jest niezbędny, aby można było uniknąć wczytywania niekompletnych danych i propagowania nieprawidłowego zbioru. Fragment kodu przedstawia prostą weryfikację kolejnej partycji. Platforma Airflow obsługuje także inne sensory, takie jak AWS Glue (`AwsGlueCatalogPartitionSensor`), GCP BigQuery (`BigQueryTablePartitionExistenceSensor`) czy Databricks (`DatabricksPartitionSensor`). Gdy partycja jest już gotowa, potok uruchamia zadanie pozyskiwania danych.

Listing 2.6. Przykład procesu DAG w przypadku wczytywania przyrostowego

```
next_partition_sensor = FileSensor(
  task_id='input_partition_sensor',
  filepath=get_data_location_base_dir() + '/{{ data_interval_end | ds }}',
  mode='reschedule',
)
```


Kup książkę Poleć książkę

```
load_job_trigger = SparkKubernetesOperator(application_file='load_job_spec.yaml',
  # ... pominięte dla zachowania zwięzłości
)
load_job_sensor = SparkKubernetesSensor(
  # ... pominięte dla zachowania zwięzłości
)
next_partition_sensor >> load_job_trigger >> load_job_sensor
```

Zadanie pozyskiwania danych przyjmuje argumenty określające lokalizacje wejściowe i wyjściowe, zdefiniowane na listingu 2.7. Definicja zadania opiera się na stałym czasie wykonania, wyrażonym za pomocą makra `{{ ds }}`. Wykorzystanie tej niemutowalnej właściwości znacznie upraszcza proces uzupełniania brakujących danych, ponieważ wartości nigdy się nie zmieniają. Zadanie `EventsLoader` wykorzystuje do odczytywania i zapisywania zbioru danych określone argumenty czasowe wraz z interfejsem API Apache Spark.

Listing 2.7. Wczytywanie zdarzeń z podziałem na partycje

```
# ...
  mainClass: com.waitingforcode.EventsLoader
  mainApplicationFile: "local:///tmp/dedp-1.0-SNAPSHOT-jar-with-dependencies.jar"
  arguments:
    - "/data_for_demo/input/date={{ ds }}"
    - "/data_for_demo/output/date={{ ds }}"
```

Skoro wcześniej omówiliśmy już sposób wykorzystania interfejsu API platformy Apache Spark, możemy przejść bezpośrednio do implementacji przyrostowego wczytywania danych opartego na kolumnie delta. Wersja ta różni się od podejścia partycjonowanego — pomija etap z użyciem sensora i od razu wykonuje zadanie importu danych (patrz listing 2.8).

Listing 2.8. Wczytywanie przyrostowe zbioru danych transakcyjnych (bez partycjonowania)

```
load_job_trigger = SparkKubernetesOperator(
  # ...
  application_file='load_job_spec_for_delta_column.yaml',
)
load_job_sensor = SparkKubernetesSensor(
  # ...
)
load_job_trigger >> load_job_sensor
```

To zadanie wykorzystuje dodatkowy etap filtrowania z użyciem kolumny delta w celu pobrania wyłącznie tych wierszy, które mieszczą się w zakresie czasowym skonfigurowanym dla danego procesu. Nawet jeśli będzie potrzebne ponowne wykonanie zadania, dzięki filtrowi nie zostaną pobrane żadne nadmiarowe rekordy, co zapewni spójność zbioru danych. Zadanie wykorzystujące implementację opartą na kolumnie delta przedstawiono na listingu 2.9.

Listing 2.9. Zadanie wczytywania danych z wykorzystaniem kolumny delta i ograniczeń czasowych

```
in_data = (spark_session.read.text(input_path).select('value',
  functions.from_json(functions.col('value'), 'ingestion_time TIMESTAMP')))

input_to_write = in_data.filter(
  f'ingestion_time BETWEEN "{date_from}" AND "{date_to}"'
)
input_to_write.mode('append').select('value').write.text(output_path)
```


Kup książkę Poleć książkę

Wzorzec: Wykrywacz Zmian w Danych

Przyrostowe wczytywanie danych nie sprawdzi się we wszystkich przypadkach. Jeśli potrzebujesz krótszego czasu przetwarzania lub wbudowanej obsługi fizycznego usuwania rekordów, lepszym wyborem będzie kolejny wzorzec.

Problem

Niestety, dane o wizytach pochodzące ze starszych systemów, które zostały zintegrowane z procesem przyrostowego wczytywania, wymagają zmodyfikowania. Ich tempo pozyskiwania jest zbyt wolne, a odbiorcy danych z kolejnych etapów przetwarzania zaczęli zgłaszać zastrzeżenia dotyczące zbyt długiego czasu oczekiwania na dostępność informacji. Aby rozwiązać ten problem, Twój menedżer produktu poprosił o możliwie szybkie zintegrowanie transakcyjnych rekordów z brokerem strumieniowym. Proces wczytywania danych musi wychwytywać każdą zmianę w tabeli w ciągu maksymalnie 30 sekund i udostępniać ją innym odbiorcom za pośrednictwem głównego strumienia danych.

Rozwiązanie

Wymóg niskiego opóźnienia uniemożliwia zastosowanie wzorca wczytywania przyrostowego. Jego użycie wiąże się z narzutami wynikającymi z harmonogramowania zadań i wykonywania zapytań, co może utrudniać osiągnięcie oczekiwanego poziomu opóźnień.

Lepszym rozwiązaniem jest zastosowanie wzorca Wykrywacz Zmian w Danych. Zapewnia on mniejsze opóźnienie dzięki wewnętrznemu mechanizmowi wczytywania. Jego działanie polega na ciągłym wczytywaniu wszystkich zmodyfikowanych wierszy bezpośrednio z dziennika zatwierdzeń bazy danych. Umożliwia to dostęp do rekordów na niższym poziomie i z większą prędkością niż jakiekolwiek zapytanie lub zadanie przetwarzające na poziomie wyższym.

Dziennik zatwierdzeń to struktura umożliwiająca jedynie dopisywanie, na której końcu są zapisywane wszystkie operacje wykonane na istniejących wierszach. Konsument wzorca Wykrywacz Zmian w Danych strumieniuje te zmiany i przesyła je do brokera strumieniowego lub innego skonfigurowanego kanału danych wyjściowych. Od tego momentu konsumenci mogą dowolnie przetwarzać te dane — na przykład przechowywać pełną historię zmian albo zachowywać jedynie najnowszą wartość każdego wiersza.

Oprócz tego, że zapewnia niskie opóźnienie, wzorzec Wykrywacz Zmian w Danych przechwytuje wszystkie typy operacji na danych, w tym usunięcia twarde. Oznacza to, że nie trzeba prosić producentów danych o stosowanie usuwania miękkiego w celu oznaczania rekordów do skasowania.

Następstwa

Obietnica niskich opóźnień jest kusząca, ale — jak każdy komponent inżynieryjny — również ten wzorzec niesie ze sobą pewne wyzwania.

Kup książkę Poleć książkę

Złożoność. Wzorzec Wykrywacz Zmian w Danych różni się od dwóch wcześniej omówionych, ponieważ wymaga innych umiejętności wdrożeniowych. Wzorce Wczytywanie Pełne i Wczytywanie Przyrostowe inżynier danych może zaimplementować samodzielnie, o ile są dostępne odpowiednie warstwy obliczeniowa i orkiestracyjna. Wzorzec Wykrywacz Zmian w Danych może jednak wymagać współpracy z zespołem operacyjnym — na przykład w celu włączenia dziennika zatwierdzeń na serwerach.

Zakres danych. Należy uważnie określić zakres danych, które mają zostać objęte tym wzorcem. W zależności od implementacji, możliwe jest pozyskiwanie jedynie zmian wprowadzonych po uruchomieniu klienta. Jeśli potrzebujesz również wcześniejszych, konieczne będzie połączenie wzorca z innymi metodami pozyskiwania danych opisanymi w tym rozdziale.

Dane wyjściowe. Wzorzec Wykrywacz Zmian w Danych różni się od wzorca Wczytywanie Przyrostowe nie tylko opóźnieniem, ale także danymi wyjściowymi. Ten pierwszy dostarcza dodatkowych metadanych wraz z rekordami, takimi jak typ operacji (aktualizacja, dodanie, usunięcie), czas modyfikacji czy typ kolumny. Jako odbiorca takich danych możesz chcieć dostosować logikę przetwarzania, aby ignorować nieistotne atrybuty.

Semantyka danych. Nie zrozum tego źle — wzorzec Wykrywacz Zmian w Danych pobiera dane statyczne. Efektem ubocznym tego procesu jest jednak to, że statyczne wiersze stają się danymi w ruchu. Dlaczego warto to podkreślić? Dane w ruchu charakteryzują się zupełnie inną semantyką przetwarzania niż dane statyczne — nawet w przypadku operacji, które na pierwszy rzut oka wydają się trywialne.

Spójrzmy na przykład operacji `JOIN`. Jeśli wykonujesz ją na tabelach statycznych, zarządzanych przez warstwę orkiestracyjną, a nie otrzymujesz wyniku, oznacza to po prostu, że nie istnieje pasujące dopasowanie. Jeśli natomiast uruchamiasz zapytanie na dwóch dynamicznych źródłach strumieniowych i nie uzyskujesz dopasowania, przyczyną może być to, że dane *jeszcze* się nie pojawiły. Jeden ze strumieni może być opóźniony względem drugiego i operacja `JOIN` może się zakończyć powodzeniem dopiero w przyszłości. Z tego też powodu danych pozyskiwanych przez konsumenta wzorca Wykrywacz Zmian w Danych nie należy traktować jako danych statycznych.

Przykłady

Istnieje wiele sposobów implementacji tego wzorca. Możesz stworzyć własny mechanizm odczytu dziennika zatwierdzeń albo skorzystać z gotowych rozwiązań. Jednym z najpopularniejszych rozwiązań o otwartym oprogramowaniu jest Debezium[2]. Ta biblioteka obsługuje wiele baz danych relacyjnych i NoSQL, a do połączenia świata danych statycznych (ang. *data-at-rest*) z danymi w ruchu (ang. *data-in-motion*) wykorzystuje narzędzie Kafka Connect. Sposób działania tego rozwiązania jest w dużej mierze oparty na parametrach konfiguracyjnych (patrz listing 2.10).

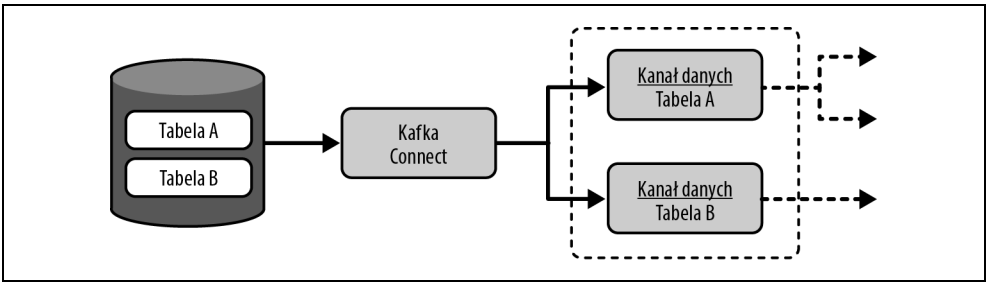
[2] Oprócz tego, że współpracuje z narzędziem Kafka Connect, Debezium oferuje jeszcze dwie inne implementacje: Debezium Embedded Engine oraz Debezium Server.

Kup książkę Poleć książkę

Listing 2.10. Konfiguracja biblioteki Debezium z narzędziem Kafka Connect dla bazy PostgreSQL

```
{
  "name": "visits-connector",
  "config": {
    "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "postgres", "database.port": "5432",
    "database.user": "postgres", "database.password": "postgres",
    "database.dbname" : "postgres", "database.server.name": "dbserver1",
    "schema.include.list": "dedp_schema",
    "topic.prefix": "dedp"
  }
}
```

Ten fragment przedstawia plik konfiguracyjny dla bazy danych PostgreSQL. Zawiera on parametry połączenia, listę wszystkich schematów objętych operacją monitorowania oraz prefiks, który zostanie użyty w nazwach kanałów danych tworzonych dla każdej synchronizowanej tabeli. W rezultacie, jeśli istnieje tabela `dedp_schema.events`, konektor zapisze wszystkie zmiany do kanału danych `dedp.dedp_schema.events`. Zależność ta jest zilustrowana na rysunku 2.3.



Rysunek 2.3. Architektura Debezium w skrócie — rysunek przedstawia konsumenta Kafka Connect, który wykorzystuje dzienniki zatwierdzeń bazy danych i zapisuje wszystkie pasujące zbiory danych do oddzielnych kanałów danych Apache Kafka

Oprócz utworzenia nowego zadania Kafka Connect niezbędne jest odpowiednie przygotowanie bazy danych. PostgreSQL wymaga włączenia strumienia replikacji logicznej z wykorzystaniem wtyczki `pgoutput` oraz konfiguracji użytkownika posiadającego wszystkie wymagane uprawnienia. W tym kontekście staje się jasne, dlaczego wzorzec Wykrywacz Zmian w Danych jest bardziej złożony w implementacji niż wczytywanie przyrostowe — szczególnie pod kątem początkowej konfiguracji.

Dobrą wiadomością jest to, że natywne formaty jezior danych obsługują powyższy wzorzec w prostszy sposób. Delta Lake oferuje wbudowaną funkcję **strumieniowania zmian** (ang. *Change Data Feed*, w skrócie *CDF*), która umożliwia przesyłanie zmodyfikowanych wierszy. Można ją włączyć globalnie (jako właściwość sesji) lub lokalnie (jako właściwość tabeli) z użyciem opcji `readChangeFeed`. Oba wspomniane podejścia konfiguracyjne zilustrowano na listingu 2.11.

Listing 2.11. Konfiguracja CDF w Delta Lake

```
spark_session_builder
  .config('spark.databricks.delta.properties.defaults.enableChangeDataFeed', 'true')
```


Kup książkę Poleć książkę

```
spark_session.sql('''
  CREATE TABLE events (
    visit_id STRING, event_time TIMESTAMP, user_id STRING, page STRING
  )
  TBLPROPERTIES (delta.enableChangeDataFeed = true)''')
```

Dodatkowo dzięki właściwości `enableChangeDataFeed` można skonfigurować limity przepustowości za pomocą parametrów `maxFilesPerTrigger` lub `maxBytesPerTrigger`. Tabele obsługują również funkcję „podróży w czasie", co pozwala rozpocząć odczyt od określonej wersji. Na listingu 2.12 jest przedstawiona najprostsza konfiguracja komponentu odczytującego, który przy każdym uruchomieniu przetwarza cztery pliki, począwszy od pierwszej wersji tabeli.

Listing 2.12. Wykorzystanie opcji CDF w Delta Lake

```
events = (spark_session.readStream.format('delta')
  .option('maxFilesPerTrigger', 4).option('readChangeFeed', 'true')
  .option('startingVersion', 0).table('events'))
query = events.writeStream.format('console').start()
```

Poza odmienną konfiguracją komponentu odczytującego, różnica występuje również w zwracanym zbiorze danych. Na listingu 2.13 pokazano, że tabela CDF zawiera dodatkowe kolumny w porównaniu z tabelą klasyczną.

Listing 2.13. Tabela wynikowa CDF

```
+-------------+-------------------+------------+---------------+--------------------+
|     visit_id|         event_time|_change_type|_commit_version|   _commit_timestamp|
+-------------+-------------------+------------+---------------+--------------------+
| 1400800256_0|2023-11-24 01:44:00|      insert|              6|2023-12-03 13:28:...|
| 1400800256_1|2023-11-24 01:36:00|      insert|              6|2023-12-03 13:28:...|
| 1400800256_2|2023-11-24 01:44:00|      insert|              6|2023-12-03 13:28:...|
| 1400800256_3|2023-11-24 01:37:00|      insert|              6|2023-12-03 13:28:...|
+-------------+-------------------+------------+---------------+--------------------+
```

Dodatkowe kolumny na listingu 2.13 rozpoczynają się od znaku podkreślenia i informują, odpowiednio, o tym, w jaki sposób wiersz się zmienił, w której wersji oraz kiedy to nastąpiło. Ten przykład pochodzi z tabeli tylko do dopisywania i może nie być szczególnie interesujący. Jeśli jednak zastosujesz operacje natychmiastowe, takie jak `UPDATE`, strumień zmian będzie zawierał wiersze zarówno sprzed aktualizacji, jak i po. Można je zidentyfikować za pomocą typów `update_preimage` i `update_postimage`.

Powielanie danych

Kolejną rodziną wzorców pobierania danych jest replikacja danych, której głównym celem jest kopiowanie danych w niezmienionej formie z jednego miejsca do drugiego. W rzeczywistości często jednak zachodzi potrzeba modyfikacji danych wejściowych, na przykład ze względu na wymogi prawne.

Kup książkę Poleć książkę

Wczytywanie danych a replikacja

Te dwa pojęcia na pierwszy rzut oka mogą się wydawać podobne, lecz istnieje między nimi subtelna różnica. Replikacja polega na przenoszeniu danych między tymi samymi typami magazynów z idealnym zachowaniem wszystkich metadanych, takich jak klucze główne w bazie danych czy pozycje zdarzeń w brokerze strumieniowym. Wczytywanie jest bardziej uniwersalne i nie wymaga jednorodnego środowiska.

Wzorzec: Replikator Bezpośredni

Podobnie jak w przypadku wczytywania danych, obszar replikacji danych ma tryb bezpośredniego przekazywania, który możemy wykorzystać domyślnie, jeśli zaakceptujemy jego konsekwencje.

Problem

Twój proces wdrażania obejmuje trzy odrębne środowiska: deweloperskie, testowe i produkcyjne. Wiele zadań korzysta ze zbioru danych referencyjnych zawierających parametry urządzeń, który codziennie jest wczytywany w środowisku produkcyjnym z zewnętrznego interfejsu API. Aby ułatwić rozwój i wykrywanie błędów, chcesz używać tego samego zbioru danych również w pozostałych środowiskach.

Proces wczytywania zbioru danych referencyjnych wykorzystuje zewnętrzny interfejs API i nie jest idempotentny. Oznacza to, że w ciągu dnia może zwracać dla tego samego wywołania API różne wyniki. Z tego względu nie możesz po prostu skopiować i ponownie uruchomić potoku wczytywania w środowiskach deweloperskim i testowym. Potrzebujesz dokładnie tych samych danych, które zostały użyte w produkcji.

Rozwiązanie

Dostawca danych, który nie jest idempotentny, to doskonały powód do zastosowania wzorca Replikator Bezpośredni — szczególnie wtedy, gdy dodatkowo trzeba zachować spójność w różnych środowiskach. Wzorzec ten można wdrożyć zarówno na poziomie obliczeniowym, jak i infrastrukturalnym.

Implementacja na poziomie obliczeniowym opiera się na zadaniu typu EL (ang. *Extract-Load*), które składa się wyłącznie z dwóch faz: odczytu i zapisu. W idealnym przypadku zadanie EL powinno kopiować pliki lub wiersze z danych wejściowych w niezmienionej postaci (czyli bez jakiejkolwiek transformacji danych). W przeciwnym razie mogłoby to prowadzić do problemów z jakością danych, takich jak konwersje typów (na przykład z łańcuchów na daty) czy zaokrąglanie liczb zmiennoprzecinkowych.

Część infrastrukturalna opiera się na dokumencie **polityki replikacji**, w którym konfiguruje się lokalizacje wejściowe i wyjściowe, co umożliwia dostawcy magazynu danych replikację rekordów w imieniu użytkownika.

Kup książkę Poleć książkę

Następstwa

Najważniejsza lekcja płynąca z zastosowania tego wzorca to konieczność zachowania prostoty implementacji. Nawet jednak najprostsze możliwe rozwiązanie może wymagać zmierzenia się z pewnymi wyzwaniami.

Zachowanie prostoty. Pamiętaj, że potrzebujesz danych dokładnie w takiej postaci, w jakiej zostały dostarczone. Aby zminimalizować ryzyko ingerencji w replikowany zbiór danych, należy polegać na najprostszym możliwym zadaniu replikacji — najlepiej na poleceniu kopiowania dostępnym w bazie danych.

Jeśli jednak takie polecenie nie jest dostępne i trzeba skorzystać z biblioteki do przetwarzania danych (na przykład w formatach tekstowych, takich jak JSON), należy unikać korzystania z API wejścia i wyjścia dla JSON. Zamiast tego lepiej użyć prostszego interfejsu operującego na surowym tekście, który wczytuje i kopiuje wiersze bez jakiejkolwiek interpretacji.

Dodatkowo, jeśli zależy Ci na zachowaniu innych właściwości — na przykład tej samej liczby plików czy identycznych nazw plików — powinieneś unikać używania rozproszonego rozwiązania do przetwarzania danych, jeśli nie pozwala ono na dostosowanie takich parametrów.

Bezpieczeństwo i odizolowanie. Komunikacja między środowiskami jest zawsze skomplikowana i może być podatna na błędy, jeśli proces replikacji zawiera usterki. W takim przypadku istnieje ryzyko negatywnego wpływu na środowisko docelowe — nawet do tego stopnia, że stanie się ono niestabilne. Z pewnością nie chcesz dopuścić do takiej sytuacji w środowisku produkcyjnym, dlatego replikację należy wdrożyć z wykorzystaniem opcji wypychania (ang. *push*) zamiast pobierania (ang. *pull*). Oznacza to, że środowisko będące właścicielem zbioru danych będzie kopiować go do pozostałych środowisk, kontrolując w ten sposób cały proces, jego częstotliwość i przepustowość.

Mimo że strategia wypychania znacząco zmniejsza ryzyko niestabilności, nadal mogą wystąpić pewne problemy. Można sobie wyobrazić sytuację, w której uruchamiasz w swojej subskrypcji chmurowej zadanie zajmujące ostatni dostępny adres IP w podsieci przetwarzania danych. Pozostałe zadania nie zostaną uruchomione, dopóki replikator nie zakończy działania.

Dane osobowe. Jeśli replikowany zbiór danych zawiera dane osobowe lub jakiekolwiek inne informacje, które nie mogą być przesyłane ze środowiska produkcyjnego, należy zastosować wzorzec Replikator Transformujący, omówiony w dalszej części rozdziału. Wzorzec ten wprowadza dodatkowy etap transformacji, którego celem jest usunięcie wszystkich niepożądanych atrybutów.

Opóźnienie. Implementacje oparte na infrastrukturze często wiążą się z dodatkowymi opóźnieniami, dlatego zawsze należy sprawdzić umowę o gwarantowanym poziomie usług (SLA) u dostawcy chmurowego, aby ocenić, czy dane rozwiązanie spełnia wymagania. Chociaż w tym przypadku mowa jest o doświadczeniu w środowisku deweloperskim, warto rozważyć zastosowanie tej strategii również w innych obszarach — szczególnie tych bardziej wrażliwych na czas.

Metadane. Nie należy pomijać kwestii metadanych, ponieważ ich brak może sprawić, że replikowany zbiór danych będzie bezużyteczny. Przykładowo nie wystarczy samo skopiowanie plików Apache Parquet z tabeli Delta Lake. Podobnie też w przypadku narzędzia Apache Kafka

Kup książkę Poleć książkę

należy zadbać nie tylko o klucze i wartości, lecz także o nagłówki oraz kolejność zdarzeń w ramach partycji.

Przykłady

Ten wzorzec można zaimplementować na dwa sposoby. Pierwszy z nich opiera się na kodzie — może to być zarówno biblioteka do rozproszonego przetwarzania danych, jak i prosty skrypt do kopiowania danych uruchamiany w warstwie magazynowania, który już przeanalizowaliśmy podczas omawiania wzorca Wczytywanie Pełne. Rozwiązanie oparte na kodzie jest bardziej podatne na błędy, ale pozwala wykorzystać warstwę wejścia i wyjścia oferowaną przez daną bibliotekę (patrz listing 2.14).

Listing 2.14. Replikacja danych JSON z użyciem platformy Apache Spark

```
input_dataset = spark_session.read.text(f'{base_dir}/input/date=2023-11-01')
input_dataset.write.mode('overwrite').text(f'{base_dir}/output-raw/date=2023-11-01')
```

Kod wykorzystuje bibliotekę Apache Spark do synchronizacji częściowo ustrukturyzowanych plików JSON. Zastosowano tu najprostszą możliwą metodę API do kopiowania wierszy w formacie JSON bez ingerencji w same dane. Należy jednak zauważyć, że fragment kodu z przykładu nie zachowuje struktury plików (oznacza to, że liczby plików w źródle i miejscu docelowym mogą się różnić, nawet jeśli zawierają te same dane).

Kod z listingu 2.14 wydaje się prosty. Niestety, w przypadku innych magazynów danych implementacja nie zawsze będzie równie bezproblemowa. Przyjrzyjmy się teraz tej samej operacji ekstrakcji i wczytywania w kontekście kanału danych z narzędzia Apache Kafka, który wymaga dodatkowej gwarancji zachowania kolejności w obrębie partycji. Kod z listingu 2.15 byłby prostym zadaniem ekstrakcji i wczytywania, gdyby nie znajdująca się pośrodku funkcja `write_sorted_events`. Funkcja ta ma kluczowe znaczenie, ponieważ zapewnia, że replikowane rekordy będą zawierać metadane (`.option('includeHeaders'...)`) i zachowają taką samą kolejność jak dane wejściowe (`sortWithinPartitions('offset', ascending=True)`).

Listing 2.15. Replikator Bezpośredni z zachowaniem kolejności

```
events_to_replicate = (input_data_stream
 .selectExpr('key', 'value', 'partition', 'headers', 'offset'))

def write_sorted_events(events: DataFrame, batch_number: int):
  (events.sortWithinPartitions('offset', ascending=True).drop('offset').write
    .format('kafka').option('kafka.bootstrap.servers', 'localhost:9094')
    .option('topic', 'events-replicated').option('includeHeaders', 'true').save())

write_data_stream = (events_to_replicate.writeStream
  .option('checkpointLocation', f'{get_base_dir()}/checkpoint-kafka-replicator')
  .foreachBatch(write_sorted_events))
```

Oprócz implementacji opartych na kodzie istnieją również implementacje infrastrukturalne. W replikacji kanału danych Apache Kafka można skorzystać z narzędzia MirrorMaker[3],

[3] Szczegółowe omówienie narzędzia MirrorMaker wykracza poza zakres tej książki. Jeśli jesteś zainteresowany dodatkowymi informacjami na ten temat, obszerną dokumentację znajdziesz na stronie *https://oreil.ly/x0BlQ*.

Kup książkę Poleć książkę

a w przypadku plików — z mechanizmu replikacji dostarczanego przez dostawcę usług w chmurze. Na listingu 2.16 przedstawiono przykład replikacji folderu S3 za pomocą narzędzia Terraform.

Listing 2.16. Replikacja folderów AWS S3

```
resource "aws_s3_bucket_replication_configuration" "replication" {
  role = aws_iam_role.replication.arn
  bucket = aws_s3_bucket.devices_production.id

  rule {
    id = "devices"
    status = "Enabled"
    destination {
      bucket  = aws_s3_bucket.devices_staging.arn
      storage_class = "STANDARD"
    }
  }
}
```

Wzorzec: Replikator Transformujący

Nawet jeśli przykład z wykorzystaniem narzędzia Apache Kafka wygląda na skomplikowany, w niektórych scenariuszach replikacji może być potrzebny jeszcze większy nakład pracy programistycznej. Tak jest w przypadku zastosowania wzorca Replikator Transformujący.

Problem

Przed wdrożeniem nowej wersji zadania przetwarzającego dane chcesz przeprowadzić testy w realnym środowisku, aby uniknąć wystąpienia nieoczekiwanych błędów. Nie możesz użyć generatora danych syntetycznych, ponieważ dostawca często dostarcza dane niskiej jakości i żadnym narzędziem nie da się tego wiarygodnie zasymulować. Musisz zatem zreplikować dane z produkcji do środowiska testowego. Niestety, zreplikowany zbiór danych zawiera dane osobowe, które nie mogą być dostępne nigdzie poza środowiskiem produkcyjnym. W związku z tym nie możesz użyć prostego wzorca Replikator Bezpośredni.

Rozwiązanie

Jednym z największych problemów w testowaniu systemów danych są… same dane. Jeśli dostawca danych nie jest w stanie zagwarantować spójności schematów i wartości, trzeba wykorzystać dane produkcyjne. Niestety, dane produkcyjne bardzo często zawierają wrażliwe atrybuty, których nie wolno przenosić do innych środowisk, gdzie potencjalnie więcej osób mogłoby uzyskać do nich dostęp ze względu na mniej restrykcyjne zasady.

W takim przypadku należy zastosować wzorzec Replikator Transformujący, który — oprócz tradycyjnych etapów odczytu i zapisu występujących we wzorcu Replikator Bezpośredni — zawiera dodatkową warstwę transformacji.

Kup książkę Poleć książkę

Transformacja to ogólny termin na określenie procesu, który w zależności od wykorzystanego stosu technologicznego można zrealizować w następujący sposób:

- jako niestandardową funkcję odwzorowującą, jeśli używasz bibliotek Apache Spark lub Apache Flink,
- jako zapytanie SQL typu `SELECT`, jeśli logika przetwarzania może być łatwo wyrażona i wykonana w języku programowania baz danych.

Transformacja polega na zastąpieniu atrybutów, których nie należy replikować (na przykład z użyciem wzorca Anonimizator), lub po prostu na ich usunięciu, jeśli nie są potrzebne w dalszym przetwarzaniu.

Następstwa

Ponieważ będziesz tworzyć niestandardową logikę, ryzyko uszkodzenia zbioru danych jest większe niż w przypadku replikatora bezpośredniego. I to niejedyna wada obecnego wzorca!

Ryzyko transformacji dla formatów plików tekstowych. Przyjrzyjmy się pozornie nieszkodliwej transformacji wykonywanej na pliku tekstowym, na przykład w formacie JSON lub CSV. Zdefiniowano schemat dla replikowanego zbioru danych, ale przeoczono, że format daty i czasu różni się od standardu używanego przez Twoją bibliotekę do przetwarzania danych. W rezultacie replikowany zbiór danych nie zawiera wszystkich kolumn ze znacznikami czasu, co powoduje niepowodzenie wykonania zadania w środowisku testowym. Chociaż problem można szybko naprawić, wprowadza on niepotrzebne opóźnienia w procesie wdrażania.

Również w tym przypadku warto więc zastosować zasadę „utrzymuj prostotę". W naszym przykładzie zamiast definiować kolumny ze znacznikami czasu w ich oryginalnej postaci, można je po prostu skonfigurować jako łańcuchy i nie martwić się o żadne niejawne transformacje.

Desynchronizacja. Należy zwrócić szczególną uwagę, aby zadania replikacji były zgodne z tym wzorcem, co pozwala uniknąć problemów związanych z prywatnością. Dane nieustannie się zmieniają i nie ma żadnej gwarancji, że pola, które są obecnie zdefiniowane jako prywatne, nadal takie będą w przyszłości. Być może pojawią się nowe pola, a niektóre atrybuty, które teraz nie są uznawane za dane osobowe, zostaną w przyszłości tak właśnie zaklasyfikowane.

Aby uniknąć tego typu problemów, warto — jeśli to możliwe — korzystać z narzędzia do zarządzania danymi, takiego jak katalog danych lub kontrakt danych, w którym oznaczono pola wrażliwe. Dzięki takiemu narzędziu można zautomatyzować logikę przekształceń. W przeciwnym razie konieczne będzie samodzielne zaimplementowanie odpowiednich reguł.

Przykłady

Jak wspomniano, istnieją dwie możliwe metody implementacji. Pierwsza z nich to podejście polegające na **redukcji danych**, które usuwa niepotrzebne pola. Stosuje się je względnie prosto. Niektóre bazy danych i warstwy obliczeniowe, takie jak biblioteki Databricks i BigQuery, obsługują operator `EXCEPT`. Działanie tego operatora pokazano na listingu 2.17 — wybierane są wszystkie wiersze z wyjątkiem pól `ip`, `latitude` i `longitude`.

Kup książkę Poleć książkę

Listing 2.17. Redukcja zbioru danych za pomocą operatora EXCEPT

```
SELECT * EXCEPT (ip, latitude, longitude)
```

Aby usunąć niepotrzebne kolumny, możesz też wykorzystać swoje narzędzie do przetwarzania danych. Na listingu 2.18 przedstawiono, jak użyć funkcji `drop` z interfejsu programistycznego PySpark do usunięcia ze zbioru danych kolumn `ip`, `latitude` i `longitude`.

Listing 2.18. Redukcja zbioru danych z użyciem funkcji drop

```
input_delta_dataset = spark_session.read.format('delta').load(users_table_path)
users_no_pii = input_delta_dataset.drop('ip', 'latitude', 'longitude')
```

Alternatywnym sposobem przekształcania zbioru danych jest kontrolowanie dostępu do niego. Na przykład dostawca danych może udostępnić użytkownikowi tylko podzbiór dozwolonych kolumn, tak jak pokazano na listingu 2.19.

Listing 2.19. Dostęp do kolumn z tabeli visits dla użytkownika user_a

```
GRANT SELECT (visit_id, event_time, user_id) ON TABLE visits TO user_a
```

Z listingu 2.19 można się dowiedzieć, jak nadać uprawnienia do podzbioru pól w usłudze AWS Redshift. Użytkownik `user_a` będzie miał dostęp wyłącznie do trzech kolumn wymienionych po operacji `SELECT`. Choć takie rozwiązanie jest bardziej rozbudowane niż podejście oparte na operatorze `EXCEPT`, zapewnia dodatkową warstwę ochrony przy dostępie do atrybutów prywatnych. Więcej na temat tego podejścia dowiesz się przy okazji analizy wzorca Dostęp Szczegółowy.

Usunięcie wierszy nie zawsze jest jednak możliwe, zwłaszcza gdy są one istotne dla testowanego zadania przetwarzania danych. W takich przypadkach definiuje się **transformację opartą na kolumnach**, która modyfikuje pola wrażliwe (patrz listing 2.20).

Listing 2.20. Transformacja oparta na kolumnach

```
devices_trunc_full_name = (input_delta_dataset
  .withColumn('full_name',
    functions.expr('SUBSTRING(full_name, 2, LENGTH(full_name))'))
)
```

Transformacje oparte na kolumnach sprawdzają się doskonale w przypadku operacji związanych z określonymi kolumnami. Jeśli jednak trzeba wykonać operacje na poziomie wiersza lub reguła modyfikacji jest złożona, może być potrzebna **funkcja odwzorowująca** (patrz listing 2.21). Kod wykorzystuje interfejs API języka Scala dla biblioteki Apache Spark, ponieważ jest bardziej zwięzły niż jego odpowiednik w Pythonie. Najpierw wywołuje się funkcję `as`, aby przekształcić dane wejściowe do określonego typu. Następnie stosuje się logikę przekształcenia, która ostatecznie zwraca zmodyfikowany wiersz na podstawie atrybutu `transformed`.

Listing 2.21. Funkcja odwzorowująca w silnie typowanym interfejsie API języka Scala dla biblioteki Apache Spark

```
case class Device(`type`: String, full_name: String, version: String) {
  lazy val transformed = {
    if (version.startsWith("1.")) {
      this.copy(full_name = full_name.substring(1), version = "invalid")
    } else {
      this
```


Kup książkę Poleć książkę

```
      }
    }
  }
  inputDataset.as[Device].map(device => device.transformed)
```

Kompaktowanie danych

Nawet idealny zbiór danych może z czasem stać się wąskim gardłem — zwłaszcza wtedy, gdy rośnie wskutek dopływu nowych danych. W wyniku tego operacje związane z metadanymi, takie jak wyświetlanie listy plików, mogą w pewnym momencie trwać dłużej niż same transformacje przetwarzania danych.

Wzorzec: Kompaktor

Najprostszym sposobem rozwiązania problemu rosnącego zbioru danych jest zmniejszenie rozmiaru plików źródłowych w magazynie danych. W tym celu stosuje się wzorzec Kompaktor.

Problem

Twoja procedura przetwarzania danych w czasie rzeczywistym synchronizuje zdarzenia z brokera strumieniowego do magazynu obiektowego. Głównym celem jest udostępnienie danych do zadań wsadowych w czasie nie dłuższym niż 10 minut. Ponieważ jest to prosta strategia wypychania danych, cały proces działa pozornie bezproblemowo. Jednak po trzech miesiącach wszystkie zadania wsadowe zaczynają cierpieć z powodu **narzutu metadanych** — zbiór danych składa się z nadmiernej liczby małych plików. W rezultacie 70% czasu wykonywania zadań jest poświęcane na wyświetlanie listy plików do przetworzenia, a jedynie 30% na faktyczne przetwarzanie. Ma to istotny wpływ na opóźnienie i koszty, zwłaszcza gdy korzysta się z usług rozliczanych w modelu pre-paid.

Rozwiązanie

W inżynierii danych problem zbyt wielu małych plików jest dobrze znany[4]. Istnieje on od czasów platformy Hadoop i wciąż występuje w nowoczesnych architekturach opartych na magazynach obiektowych, nawet tych o praktycznie nieograniczonej pojemności. Przechowywanie dużej liczby małych plików powoduje wydłużenie operacji wyświetlania listy oraz zwiększa koszt operacji wejścia i wyjścia związanych z otwieraniem i zamykaniem plików. Naturalnym rozwiązaniem tego problemu jest zmniejszenie liczby plików.

Na tym właśnie polega działanie wzorca Kompaktor. Rozwiązuje on problem przez łączenie wielu małych plików w większe, co zmniejsza ogólny narzut operacji wejścia i wyjścia podczas odczytu. Implementacja zależy od użytej technologii. Otwarty format plików tabelarycznych zazwyczaj pozwala na użycie specjalnego polecenia kompaktowania, które w tle uruchamia

[4] Szczegółowe omówienie problemu zbyt wielu małych plików znajduje się w rozdziałach 4. i 5. książki *The Cloud Data Lake* autorstwa Rukmani Gopalan (O'Reilly, 2023).

Kup książkę Poleć książkę

transakcyjne, rozproszone zadanie przetwarzania danych w celu połączenia mniejszych plików w większe w ramach nowego zatwierdzenia. Na przykład biblioteka Apache Iceberg realizuje to poprzez operację **przepisywania pliku z danymi** (ang. *rewrite data file*), a Delta Lake używa polecenia `OPTIMIZE`.

W odróżnieniu od innych rozwiązań, kompaktowanie w Apache Hudi — trzecim z otwartych formatów plików tabelarycznych — przebiega w inny sposób. Tabela Hudi może być skonfigurowana w trybie **łączenia w czasie czytania** (ang. *merge-on-read*, w skrócie *MoR*), w którym dane są zapisywane w formacie kolumnowym, a wszelkie późniejsze zmiany — w formacie wierszowym. Ponieważ takie podejście sprzyja szybkiemu zapisywaniu danych, operacja kompaktowania w Hudi łączy zmiany z formatu wierszowego z bazową wersją kolumnową, aby zoptymalizować odczyt. Podejście to różni się od stosowanego w bibliotekach Delta Lake i Apache Iceberg, które operują wyłącznie na jednorodnym formacie kolumnowym.

Wzorzec Kompaktor można także stosować w magazynach danych innych niż te związane z architekturą lakehouse. Przykładem jest narzędzie Kafka — system dzienników oparty na kluczach, działający w trybie tylko do dopisywania. W tej implementacji opartej na konfiguracji wystarczy określić częstotliwość kompaktowania. Całym procesem zarządza później magazyn danych, zgodnie z ustaloną wcześniej częstotliwością. W przypadku systemów opartych na kluczach kompaktowanie polega na optymalizacji przestrzeni przez zachowywanie wyłącznie najnowszego wpisu dla danego klucza rekordu. Taki proces również poprawia efektywność przechowywania, jednak (w przeciwieństwie do kompaktowania w formatach plików tabelarycznych) operacja nadpisuje istniejące dane.

Następstwa

Mimo że wzorzec Kompaktor wydaje się nieszkodliwy i jest natywnie obsługiwany przez wiele magazynów danych, jego wdrożenie może wymagać znacznego wysiłku projektowego.

Kompromis między kosztem a wydajnością. Zadanie kompaktowania to w istocie zwykłe zadanie przetwarzania danych, które przy dużych rozmiarach tabel może być bardzo obciążające obliczeniowo. Z tego powodu warto wykonywać je rzadko — na przykład raz dziennie, najlepiej poza godzinami pracy i z pominięciem głównej procedury tworzenia zbioru danych.

Z drugiej strony, zbyt rzadkie uruchamianie może być problematyczne w przypadku zadań pracujących na danych jeszcze niepoddanych kompaktowaniu — nie będą one w stanie skorzystać z tej techniki optymalizacji. Trzeba więc wybrać odpowiednią strategię i zaakceptować to, że może nie być idealna ani pod względem kosztów, ani wydajności.

Niestety, nie istnieje jedno uniwersalne rozwiązanie. Czasem wystarczy uruchamiać kompaktowanie raz dziennie, jeśli zbiór danych jest na tyle mały, że proces nie wpływa negatywnie na odbiorców. Innym razem może to być nie do przyjęcia. Bardziej opłacalne będzie wówczas włączenie kompaktowania bezpośrednio do procesu pozyskiwania danych, ponieważ negatywny wpływ rozwiązania na odbiorców danych może być większy niż na przepustowość.

Spójność. Należy pamiętać, że kompaktowanie polega na ponownym zapisie już istniejących danych. W związku z tym odbiorcy mogą mieć trudności z odróżnieniem danych, z których powinni korzystać, od tych, które są obecnie poddawane kompaktowaniu. Z tego powodu

Kup książkę Poleć książkę

wdrożenie kompaktowania w przypadku nowoczesnych, otwartych formatów plików tabelarycznych z właściwościami ACID (takich jak Delta Lake i biblioteka Apache Iceberg) jest znacznie prostsze i bezpieczniejsze niż w surowych formatach plików (takich jak JSON i CSV).

Czyszczenie. Zadanie kompaktowania może pozostawiać pliki źródłowe. W wyniku tego niewielkie pliki wciąż będą obecne i nadal mogą wpływać na operacje na metadanych. Z tego powodu samo kompaktowanie nie wystarczy — należy uzupełnić je o zadanie czyszczące, które odzyska miejsce zajmowane przez pliki już skompaktowane.

Aby poradzić sobie z tym skutkiem ubocznym, należy zajętą, ale nieużywaną przestrzeń odzyskać z użyciem poleceń takich jak `VACUUM`, które są dostępne w nowoczesnych systemach przechowywania danych, takich jak narzędzie Delta Lake, biblioteka Apache Iceberg, system bazodanowy PostgreSQL oraz usługa Redshift. Trzeba jednak rozważnie dobrać strategię czyszczenia, ponieważ może się okazać, że nie będzie można odtworzyć zbioru danych w wersji opartej na tych usuniętych, wcześniej skompaktowanych plikach.

Przykład

Zacznijmy od sprawdzenia, w jaki sposób ten wzorzec stosuje się w repozytoriach danych typu data lake i lakehouse wykorzystujących otwarte formaty plików tabelarycznych, takie jak Delta Lake. Format ten oferuje natywną funkcję kompaktowania, dostępną przez programistyczny interfejs API lub jako polecenie SQL. W obu przypadkach należy szukać słowa kluczowego `optimize` (patrz listing 2.22). Kod inicjalizuje obiekt `DeltaTable` wykorzystujący ścieżkę i wywołuje operację kompaktowania danych. Ponieważ reorganizowane są jedynie pliki dostępne w momencie uruchomienia zadania, operacja ta jest bezpieczna i nie powoduje konfliktów w procesach odczytu i zapisu.

Listing 2.22. Zadanie kompaktowania z użyciem narzędzia Delta Lake

```
devices_table = DeltaTable.forPath(spark_session, table_dir)
devices_table.optimize().executeCompaction()
```

Kompaktowanie może jednak wymagać dodatkowego kroku w postaci polecenia `VACUUM`, które usunie wszystkie nieistotne (bo już skompaktowane) pliki. Na listingu 2.23 ponownie wykorzystano abstrakcję `DeltaTable`, ale tym razem wywołano funkcję `vacuum()`. Proces czyszczenia obejmuje wyłącznie pliki starsze niż skonfigurowany próg retencji. W przeciwnym razie mogłoby to doprowadzić do uszkodzenia stanu tabeli, ponieważ funkcja `vacuum` mogłaby usunąć pliki, które są w trakcie zapisu i nie zostały jeszcze zatwierdzone.

Listing 2.23. Użycie polecenia VACUUM w Delta Lake

```
devices_table = DeltaTable.forPath(spark_session, table_dir)
devices_table.vacuum()
```

Kompaktowanie jest dostępne również w innych magazynach danych, ale w porównaniu z przykładem dotyczącym narzędzia Delta Lake może w pewnym stopniu wprowadzać w błąd. Przyjrzyjmy się transakcyjnemu systemowi bazodanowemu PostgreSQL. W tym przypadku kompaktowanie polega wyłącznie na użyciu polecenia `VACUUM`, które odzyskuje przestrzeń zajmowaną przez martwe krotki, czyli wiersze skasowane, ale jeszcze fizycznie nieusunięte z magazynu danych. To polecenie można uruchomić jawnie przez jego wywołanie w konsoli.

Kup książkę Poleć książkę

Wzorzec Kompaktor występuje również w narzędziu Kafka. Podczas tworzenia strumienia danych można ustawić konfigurację kompaktowania dziennika, co jest szczególnie przydatne, gdy zapisuje się rekordy oparte na kluczu, a każde nowe dopisanie aktualizuje stan. Po włączeniu kompaktowania narzędzie Kafka uruchamia proces czyszczenia, który usuwa wszystkie wersje danego klucza z wyjątkiem najnowszej. Konfiguracja obsługuje właściwości takie jak `log.cleanup.policy` (strategia kompaktowania) oraz `log.cleaner.min.compaction.lag.ms` i `log.cleaner.max.compaction.lag.ms` (częstotliwość kompaktowania). W przeciwieństwie do wcześniejszego przykładu z użyciem narzędzia Delta Lake, kompaktowanie w środowisku Kafka jest niedeterministyczne — nie można oczekiwać, że będzie uruchamiane według stałego harmonogramu, a więc nie ma gwarancji, że każdy klucz zawsze umożliwi dostęp do unikatowego rekordu.

Gotowość danych

Analizując w tym rozdziale różne semantyki pozyskiwania danych, nie omówiliśmy dotąd wszystkich problemów, które mogą się pojawić w ramach tego z pozoru prostego zadania. Kolejne trudne pytanie, które z pewnością sobie zadasz, brzmi: „Kiedy powinienem rozpocząć proces pozyskiwania danych?”.

Wzorzec: Znacznik Gotowości

Wzorzec Znacznik Gotowości pomaga uruchomić proces pozyskiwania danych w najbardziej odpowiednim momencie. Jego celem jest zapewnienie, że zostanie pozyskany kompletny zbiór danych. Przyjrzyjmy się, jak ten cel zostaje osiągnięty.

Problem

Co godzinę uruchamiasz zadanie wsadowe, które przygotowuje dane w warstwie Silver w architekturze Medallion. Zbiór danych zawiera dane ze wszystkimi już naprawionymi znanymi problemami dotyczącymi jakości, a ponadto jest wzbogacony o dodatkowy kontekst, pochodzący z bazy danych użytkownika oraz z zewnętrznej usługi API. Pewne zespoły opierają się więc na tych danych przy tworzeniu modeli ML i pulpitów BI. Pojawia się jednak poważny problem: zespoły te często skarżą się na niekompletne zbiory danych i proszą o wdrożenie mechanizmu, który (bezpośrednio lub pośrednio) poinformuje, kiedy mogą rozpocząć korzystanie z nich.

Rozwiązanie

Problem ten jest szczególnie widoczny w przypadku logicznie powiązanych, ale fizycznie odizolowanych procedur utrzymywanych przez różne zespoły. Ze względu na tę izolację nie da się uruchomić kolejnych procedur bezpośrednio z poziomu Twojego zadania. Można jednak oznaczyć zbiór danych jako gotowy do przetwarzania przez zastosowanie wzorca Znacznik Gotowości. Jego implementacja zależy od formatu pliku oraz organizacji przechowywania danych.

Pierwsza implementacja wykorzystuje zdarzenie jako sygnał, że zbiór danych jest kompletny. Ze względu na popularność magazynów obiektowych i rozproszonych systemów plików

Kup książkę Poleć książkę

w nowoczesnych architekturach danych, podejście to można łatwo wdrożyć przez utworzenie *pliku flagi* po pomyślnym wygenerowaniu danych. Funkcjonalność ta może być natywnie dostępna w Twojej warstwie przetwarzania danych — na przykład w bibliotece Apache Spark (która w przypadku surowych formatów plików zapisuje plik *_SUCCESS*) lub jako nowy wpis w dzienniku zatwierdzeń w narzędziu Delta Lake. Jeśli nie możesz wykorzystać warstwy przetwarzania danych, plik sygnalizacyjny możesz utworzyć z poziomu warstwy orkiestracji jako osobne zadanie uruchamiane po pomyślnym przetworzeniu danych. Przykład takiej implementacji jest przedstawiony w podpunkcie „Przykłady".

Inna implementacja dotyczy źródeł danych podzielonych na partycje. Jeśli generujesz dane dla tabel lub lokalizacji opartych na czasie, wzorzec Znacznik Gotowości może mieć charakter konwencjonalny. Brzmi niejasno? Aby łatwiej Ci było zrozumieć tę konwencję, posłużmy się przykładem. Twoje zadanie uruchamia się co godzinę i zapisuje dane do partycji odpowiadającej konkretnej godzinie. W rezultacie uruchomienie o 10:00 zapisuje dane do partycji 10, uruchomienie o 11:00 — do partycji 11 itd. Jeśli odbiorca danych chce przetwarzać partycję 10, musi po prostu poczekać, aż Twoje zadanie rozpocznie obsługę partycji 11.

Następstwa

Wzorzec Znacznik Gotowości opiera się na wykorzystaniu metody „pull", w której to odbiorcy danych kontrolują proces ich pobierania. Ponieważ implementacje są niejawne, należy zwrócić uwagę na kilka kwestii.

Brak wymuszenia. Nie ma łatwego sposobu na to, aby wymusić przestrzeganie konwencjonalnej gotowości opartej na pliku flagi lub wykrywaniu kolejnej partycji. W każdej z tych sytuacji odbiorca może rozpocząć pobieranie zbioru danych w momencie, gdy jest on jeszcze generowany.

Ze względu na tę niejawność niezwykle istotna jest komunikacja z odbiorcami danych oraz uzgodnienie warunków, które mogą uruchomić przetwarzanie po ich stronie. Należy również jasno wyjaśnić ryzyko związane z niestosowaniem się do konwencji dotyczących gotowości.

Niezawodność w przypadku spóźnionych danych. Jeśli partycje są oparte na czasie zdarzenia, implementacja będzie podatna na problemy związane z danymi opóźnionymi. Aby to lepiej zrozumieć, wyobraź sobie, że zamknąłeś partycje odpowiadające godzinom ósmej, dziewiątej i dziesiątej. Oznacza to, że Twoi odbiorcy danych przetworzyli już partycje z tych godzin. Niestety, właśnie wykryłeś dane opóźnione dla godziny dziewiątej. Ponieważ partycje są oparte na czasie zdarzenia, postanawiasz zintegrować te dane z partycją odpowiadającą godzinie dziewiątej. Najprawdopodobniej Twoi odbiorcy nie będą jednak w stanie zrobić tego samego, ponieważ mogą uznać tę partycję za zamkniętą.

Należy więc traktować partycje jako niezmienne elementy, które po zamknięciu nigdy się nie zmienią, albo jasno określić i przekazać odbiorcom warunki, w jakich dopuszcza się ich modyfikację. Oprócz udostępniania odbiorcom aktualizacji partycji należy także informować ich o nowo dostępnych danych do ewentualnego przetworzenia. Tym problemem zajmiemy się w podrozdziale „Dane opóźnione" w rozdziale 3.

Kup książkę Poleć książkę

Przykłady

W przypadku plików surowych biblioteka Apache Spark domyślnie tworzy plik flagi o nazwie *_SUCCESS*. Za każdym razem, gdy generujesz pliki w formacie Apache Parquet, Apache Avro lub innym, zadanie najpierw zapisuje pliki z danymi, a dopiero po zakończeniu tej operacji tworzy plik flagi. Zaimplementowano to na listingu 2.24, na którym zadanie konwertuje pliki JSON do formatu Apache Parquet, a w tle tworzy plik *_SUCCESS*.

Listing 2.24. Kod PySpark generujący plik _SUCCESS

```
dataset = (spark_session.read.schema('...').json(f'{base_dir}/input'))
dataset.write.mode('overwrite').format('parquet').save('devices-parquet')
```

Aby zaimplementować wzorzec Znacznik Gotowości, jako odbiorca danych polegasz na utworzonym pliku *_SUCCESS*. Jeśli korzystasz z narzędzia Apache Airflow, możesz określić ten plik jako warunek w komponencie `FileSensor` (patrz listing 2.25).

Listing 2.25. Komponent FileSensor oczekujący na plik _SUCCESS

```
FileSensor(filepath=f'{input_data_file_path}/_SUCCESS, mode='reschedule'
#...
```

Na listingu 2.25 zdefiniowano ścieżkę do pliku, a także przypisano istotnej właściwości `mode` wartość `reschedule`. Dzięki temu zadanie sensora nie będzie bez przerwy zajmować slotu wykonawczego. Zamiast tego będzie się okresowo uruchamiać i sprawdzać, czy skonfigurowany plik istnieje. To ważny aspekt, jeśli chcesz uniknąć sytuacji, w której warstwa orkiestracji jest zajęta wyłącznie oczekiwaniem na dane, podczas gdy inne zadania mogą być gotowe do wykonania.

Swoją drogą, narzędzie Apache Airflow ułatwia również tworzenie pliku będącego znacznikiem gotowości, jeśli nie jest on natywnie generowany przez zadanie przetwarzania danych. Kod z listingu 2.26 generuje plik o nazwie *COMPLETED* w ostatnim zadaniu, o nazwie `created_readiness_file`.

Listing 2.26. Tworzenie pliku znacznika gotowości jako część procesu orkiestracji danych

```
@task
def delete_dataset():
  shutil.rmtree(dataset_dir, ignore_errors=True)

@task
def generate_dataset():
# kod przetwarzający pominięty dla zachowania zwięzłości, ale dostępny w repozytorium na GitHubie

@task
def create_readiness_file():
  with open(f'{dataset_dir}/COMPLETED', 'w') as marker_file:
    marker_file.write('')

delete_dataset() >> generate_dataset() >> create_readiness_file()
```

Ten fragment kodu wykorzystuje dekorator `@task`, który umożliwia proste deklarowanie funkcji przetwarzających dane w narzędziu Apache Airflow. Najważniejszą sprawą, o której należy pamiętać, jest to, że znacznik gotowości powinien być zawsze generowany jako finalny krok w procedurze, czyli po wykonaniu ostatniej transformacji zbioru danych.

Kup książkę Poleć książkę

Sterowane zdarzeniami

W idealnych sytuacjach pozyskiwania danych nowe zbiory danych są dostępne zgodnie z regularnym harmonogramem. Można wtedy uruchomić procedurę i przed rozpoczęciem przetwarzania zastosować wzorzec Znacznik Gotowości. W mniej idealnych przypadkach trudno przewidzieć częstotliwość napływu danych. W takiej sytuacji należy zmienić podejście — z pozyskiwania statycznego na pozyskiwanie zdarzeniowe.

Wzorzec: Wyzwalacz Zewnętrzny

Wzorzec Znacznik Gotowości, omówiony w poprzednim podrozdziale, opiera się na **semantyce pobierania**, w której to odbiorca danych odpowiada za sprawdzanie, czy pojawiły się nowe dane do przetworzenia. Zdarzeniowy charakter zbioru danych sprzyja jednak **semantyce wypychania**, w której to producent danych odpowiada za powiadomienie odbiorców o dostępności danych.

Problem

Załóżmy, że zespół infrastruktury technicznej w Twojej organizacji wprowadza nowe funkcje co najwyżej raz w tygodniu — między poniedziałkiem a czwartkiem. Każda taka publikacja wzbogaca referencyjne zbiory danych, w których przechowujesz wszystkie funkcje dostępne w danym momencie na stronie internetowej. Do tej pory zadanie odświeżania tego zbioru danych było zaplanowane jako codzienne. Powodowało to przeładowywanie danych nawet wtedy, gdy nie było żadnych zmian, co prowadziło do niepotrzebnego zużycia zasobów obliczeniowych.

W celu ograniczenia kosztów chcesz zmienić mechanizm harmonogramowania i uruchamiać procedurę tylko wtedy, gdy pojawi się coś nowego do przetworzenia. Ponieważ zespół wysyła zdarzenie powiadamiające do centralnej szyny komunikatów za każdym razem, gdy publikuje nową funkcję, rozważasz wdrożenie podejścia zorientowanego na zdarzenia.

Rozwiązanie

Nieprzewidywalne generowanie danych jest często wynikiem ich zdarzeniowego charakteru. Problem ten można rozwiązać za pomocą zadania uruchamianego cyklicznie, które kopiuje cały zbiór danych lub bardzo często sprawdza, czy pojawiły się nowe dane do przetworzenia. Takie rozwiązanie prowadzi jednak do marnotrawienia zasobów obliczeniowych i niepotrzebnie obciąża system. Lepszym podejściem jest zastosowanie zdarzeniowego wzorca Wyzwalacz Zewnętrzny.

Nie tylko wyzwalacz

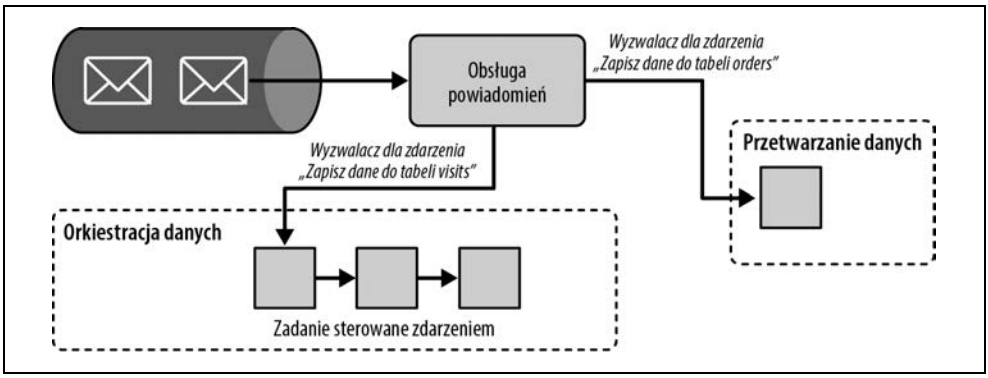
Jeśli nie istnieje żadne zadanie, które można wywołać, proces pozyskiwania danych realizuje się bezpośrednio w komponencie obsługującym powiadomienie. To podejście określamy jako **pozyskiwanie danych sterowane zdarzeniem**.

Kup książkę Poleć książkę

Wzorzec składa się z trzech głównych kroków:

1. Subskrybowanie kanału powiadomień. Pierwszy krok polega na ustanowieniu połączenia między światem zewnętrznym (zdarzeniowymi producentami danych) a Twoimi procedurami. Od tej pory za każdym razem, gdy wydarzy się coś nowego, będziesz mieć możliwość odpowiedniej reakcji.
2. Reagowanie na powiadomienia. Jest to obsługa zdarzeń, której zadaniem jest analiza zdarzenia i decyzja, czy powinno ono skutkować uruchomieniem procedury w warstwie orkiestracji danych czy też zadania w warstwie przetwarzania danych. W zależności od zastosowanej technologii szyny komunikatów, obsługa może subskrybować określone zdarzenia — na przykład utworzenie danych w pewnej tabeli. Jeśli taka funkcja filtrowania nie jest dostępna, trzeba ją będzie zaimplementować samodzielnie.
3. Wyzwolenie procedury pozyskiwania danych w warstwie orkiestracji danych lub przetwarzania danych. W tym kroku obsługa uruchamia proces pozyskiwania danych — albo przez wyzwolenie przepływu zadań do orkiestracji, albo bezpośrednio przez uruchomienie zadania. Dla uproszczenia — jedno zdarzenie powinno wyzwalać jedną procedurę pozyskiwania danych. Możliwe jest jednak uruchamianie wielu procedur — na przykład wtedy, gdy ten sam zbiór danych jest źródłem danych wejściowych dla różnych zadań.

Te trzy główne działania są przedstawione na rysunku 2.4, na którym strzałka wyzwalacza reprezentuje reakcję na subskrybowany kanał powiadomień.



Rysunek 2.4. Zewnętrzny wyzwalacz dla warstwy orkiestracji danych i warstwy przetwarzania danych

Następstwa

Chociaż model sterowany zdarzeniami wydaje się atrakcyjny ze względu na zmniejszenie marnotrawstwa zasobów, jego stosowanie wiąże się również z pewnymi konsekwencjami dla stosu danych.

Wypychanie a pobieranie. Komponent Wyzwalacz Zewnętrzny może realizować semantykę pobierania lub wypychania. Różnica między nimi jest kluczowa dla zrozumienia wpływu wzorca na Twój system. Wyzwalacz oparty na pobieraniu wciąż sprawdza, czy pojawiły się nowe zdarzenia do przetworzenia, natomiast wyzwalacz oparty na wypychaniu pozostaje nieaktywny, dopóki producent nie powiadomi go o nowych danych do przetworzenia.

Kup książkę Poleć książkę

Wyzwalacz oparty na pobieraniu to po prostu długo działające zadanie, czyli proces, który jest stale uruchomiony i w krótkich, regularnych odstępach sprawdza, czy są dostępne nowe dane. To implementacja poprawna technicznie, ale nieoptymalna — zadanie może przez większość czasu nie odbierać żadnych komunikatów z kanału powiadomień.

Zamiast tego wzorca lepiej użyć wyzwalacza opartego na wypychaniu, w którym źródło danych informuje punkt końcowy o nowych komunikatach obecnych na szynie komunikatów. Każde powiadomienie uruchamia nową instancję odbiorcy danych, która kończy działanie po obsłużeniu zdarzenia.

Kontekst wykonania. Istnieje ryzyko, że wyzwalacz zewnętrzny stanie się *jedynie* mechanizmem typu „ping", który wywołuje punkt końcowy orkiestratora danych. Choć takie uproszczone podejście może kusić, należy pamiętać, że wyzwalana procedura pozyskiwania danych będzie musiała być utrzymywana jak każda inna. Jeśli więc ograniczysz się do jej uruchamiania, możesz nie dysponować wystarczającym kontekstem, by zrozumieć, dlaczego została wywołana i co ma zostać przetworzone.

Z tego względu ważne jest, aby wzbogacić wywołanie wyzwalające o odpowiednie metadane — takie jak wersja zadania wyzwalającego, struktura powiadomienia, czas przetwarzania oraz czas zdarzenia. Informacje te przydadzą się w codziennym monitorowaniu, zwłaszcza gdy konieczne będzie ustalenie przyczyn ewentualnych błędów.

Zarządzanie błędami. Zdarzenia są tutaj elementami kluczowymi — bez nich nie da się wyzwolić żadnego działania. Wyzwalacz powinno się więc projektować z myślą o awariach, tak aby zdarzenia były zachowane niezależnie od tego, co się wydarzy. W tym celu zazwyczaj stosuje się wzorce opisane w następnym rozdziale, takie jak wzorzec Obsługa Błędnych Komunikatów.

Przykłady

Wzorzec ten jest łatwy do zaimplementowania w chmurze. AWS, Azure i GCP oferują usługi funkcji bezserwerowych, które umożliwiają sterowanie zdarzeniami. Idealnie nadają się więc do roli wyzwalaczy. Ponadto większość narzędzi do orkiestracji danych udostępnia interfejs API, za pomocą którego można uruchamiać procedury. Przyjrzyjmy się teraz, jak połączyć funkcję AWS Lambda z narzędziem Apache Airflow. Najpierw spójrzmy na definicję procedury pokazaną na listingu 2.27.

Listing 2.27. Definicja procesu DAG wyzwalanego zewnętrznie

```
with DAG('devices-loader', max_active_runs=5, schedule_interval=None,
default_args={'depends_on_past': False,}) as dag:
# Procedura jedynie kopiuje plik z miejsca wskazanego przez wyzwalacz
# Pomijam treść dla zachowania zwięzłości, całość jest dostępna w repozytorium na GitHubie
```

W odróżnieniu od poprzednich procedur DAG, ta przedstawiona na listingu 2.27 wykorzystuje parametr `schedule_interval` ustawiony na wartość `None`. Oznacza to, że na etapie planowania harmonogram Apache Airflow całkowicie ją pominie, a jedynym sposobem uruchomienia procedury będzie jawne wywołanie przez mechanizm wyzwalający. Odpowiedzialność za to spoczywa na funkcji Lambda, która zostanie uruchomiona za każdym razem, gdy w monitorowanym folderze *S3* pojawi się nowy obiekt (patrz listing 2.28).

Kup książkę Poleć książkę

Listing 2.28. Funkcja AWS Lambda służąca do uruchamiania procesu DAG

```
def lambda_handler(event, ctx):
  payload = {
    'event': json.dumps(event),
    'trigger': {
      'function_name': ctx.function_name, 'function_version': ctx.function_version,
      'lambda_request_id': ctx.aws_request_id
    },
    'file_to_load': (urllib.parse.
    unquote_plus(event['Records'][0]['s3']['object']['key'],encoding='utf-8')),
    'dag_run_id': f'External-{ctx.aws_request_id}'
  }

trigger_response = requests
  .post('http://localhost:8080/api/v1/dags/devices-loader/dagRuns',
  data=json.dumps({'conf': payload}), auth=('dedp', 'dedp'), headers=headers)

if trigger_response.status_code != 200:
  raise Exception(f"""Couldn't trigger the `devices-loader` DAG.
    {trigger_response} for {payload}""")
else:
    return True
```

Jak widać, funkcja jest stosunkowo prosta. Możesz się zastanawiać, gdzie w tym wszystkim podział się element odporności na błędy. AWS Lambda zapewnia go na poziomie infrastruktury, wykorzystując mechanizm **docelowych miejsc dla nieudanych zdarzeń** (ang. *failed-event destinations*), do których trafiają wszystkie rekordy z nieudanych wywołań funkcji. Dodatkowo można skonfigurować rozmiar partii dla źródeł danych strumieniowych lub poziom współbieżności.

Jawność fragmentów kodu

Fragmenty kodu — o ile nie zaznaczono inaczej — napisano z myślą o czytelności. To jeden z powodów, dla których na listingu 2.28 znajdują się zakodowane dane uwierzytelniające. Umieszczanie danych uwierzytelniających bezpośrednio w kodzie jest złą praktyką, ponieważ łatwo mogą one zostać ujawnione. Aby temu zapobiec, można zastosować jeden ze wzorców projektowych dotyczących bezpieczeństwa danych, opisanych w rozdziale 7.

Podsumowanie

Gdy rozpoczynałeś lekturę tego rozdziału, prawdopodobnie postrzegałeś pozyskiwanie danych jako etap niezbędny, lecz niezbyt wymagający technicznie. Mam nadzieję, że ten rozdział udowodnił Ci, jak mylne jest to przekonanie.

Dowiedziałeś się, że nawet z tak, wydawałoby się, prostą operacją jak przenoszenie danych z jednego miejsca do drugiego wiąże się szereg wyzwań. Jeśli nie zastosujesz wzorca Znacznik Gotowości, jako konsument możesz wczytać niekompletne dane, a jako dostawca — narazić się na złą opinię wśród użytkowników. Z kolei bez zastosowania wzorca Kompaktor nawet praktycznie nieograniczony lakehouse może bardzo szybko stać się wąskim gardłem wydajności, i to wyłącznie z powodu liczby wywołań API.

Kup książkę Poleć książkę

Choć temat pozyskiwania danych został przedstawiony już na początkowych stronach tej książki, pamiętaj, że ten krok nie jest zarezerwowany wyłącznie dla punktów wejściowych systemu ani dla prostych procedur EL. Większość wzorców przeanalizowanych w tym rozdziale doskonale się nadaje do zastosowania w kroku ekstrakcji w procedurach ETL oraz ELT — a zatem również w zadaniach o charakterze biznesowym. To kolejny powód, by nie lekceważyć ich znaczenia.

Na koniec mam dla Ciebie dwie wiadomości: dobrą i złą. Dobra to taka, że masz teraz do dyspozycji listę gotowych szablonów, które możesz zastosować do pozyskiwania danych — zarówno w kontekstach czysto technicznych, jak i biznesowych. Zła wiadomość brzmi: to dopiero początek. Po pozyskaniu danych trzeba je przetworzyć — a i na tym etapie może się pojawić wiele problemów. Mam jednak nadzieję, że następny rozdział dostarczy Ci dalszych wskazówek, które pomogą budować jeszcze bardziej wydajne systemy inżynierii danych!

Kup książkę Poleć książkę


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Bartosz Konieczny stworzył kompleksowy przewodnik po projektowaniu odpornych potoków danych, oparty na sprawdzonych i wielokrotnie używanych wzorcach projektowych.

— **Adi Polak**, dyrektor, Confluent

Każda nowoczesna organizacja opiera się na danych. Mimo to inżynierowie w wielu firmach niemal każdego dnia mierzą się z licznymi problemami, które można rozwiązać za pomocą sprawdzonych już metod. Zastosowanie gotowego wzorca projektowego umożliwia reagowanie na typowe wyzwania biznesowe w sposób bezpieczny i zoptymalizowany.

Z tej książki dowiesz się, jak dostarczać dane o realnej wartości, koncentrując się na kluczowych aspektach inżynierii danych: pozyskiwaniu danych, ich jakości czy idempotentności. Poznasz cały proces budowania niezawodnych i kompleksowych rozwiązań z zakresu inżynierii danych na bazie wzorców projektowych, przygotowanych do realizacji konkretnych celów biznesowych. Opis każdego wzorca zawiera prezentację problemu z perspektywy użytkownika, rozwiązanie, a także omówienie następstw, co pozwala osadzić ten problem w kontekście realnych sytuacji. Pokazano też, jak w praktyce zaimplementować opisane rozwiązania za pomocą narzędzi *open source* i usług chmury publicznej.

W książce:

- praca z danymi w wielu formatach, bazami i magazynami danych
- istotne problemy dotyczące poszczególnych elementów systemów danych
- praktyczne zastosowania wzorców
- identyfikacja i rozwiązywanie problemów istniejących komponentów danych
- niezależne od technologii rozwiązania dla nowych i istniejących projektów

Bartosz Konieczny to niezależny inżynier danych, tworzy oprogramowanie od 2010 roku. Zajmował wysokie stanowiska techniczne, na których zdobywał szerokie doświadczenie w pracy nad zagadnieniami z zakresu inżynierii danych — zarówno w przetwarzaniu wsadowym, jak i strumieniowym.

Ta książka to fundamentalne dzieło, które wyznacza kierunek rozwoju wzorców projektowych w inżynierii danych!

Scott Haines, współautor książki *Delta Lake: The Definitive Guide*

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-3179-4
9 788328 931794
Cena: 89,00 zł