Hurtownia danych dla analizy ofert bankowych

Paweł Bubak 132197,
Damian Horna 132240, horna.damian@gmail.com
Dominik Kuszmiruk 132266,
Rafał Morawski 131341, rafal.morawski@protonmail.com
Kamil Pluciński 132307

15 czerwca 2020

Spis treści

1 Opis problemu

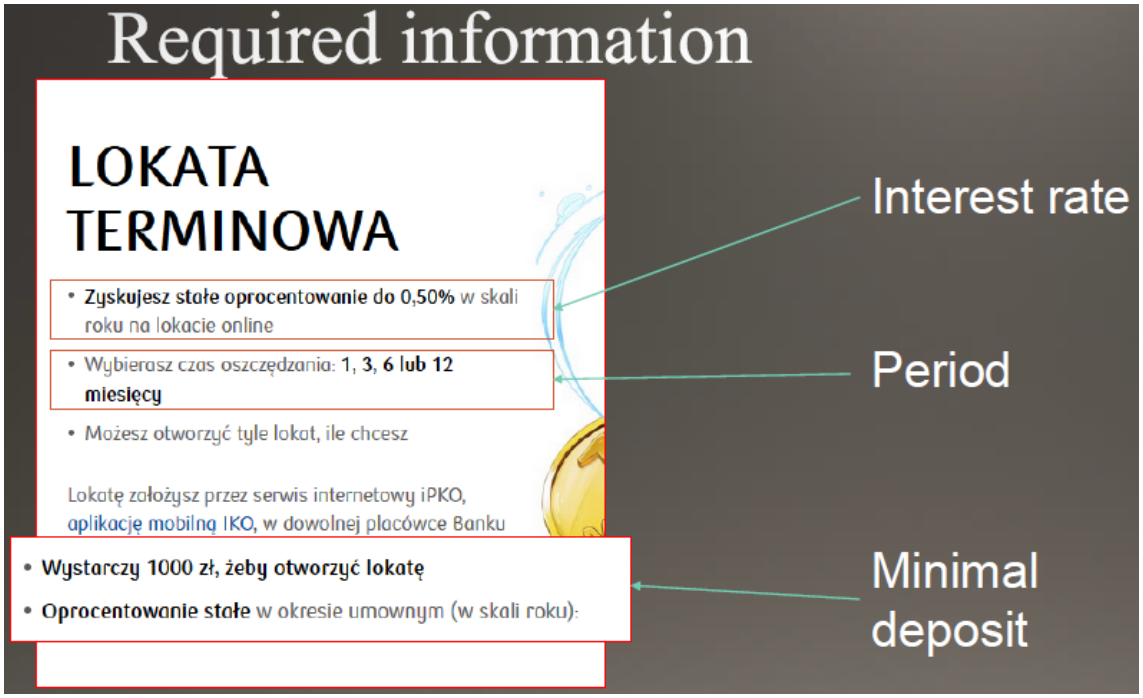
Celem projektu jest zaprojektowanie i zbudowanie architektury systemu umożliwiającego automatyczne pobieranie, składowanie i przetwarzanie danych na temat ofert podmiotów konkurencyjnych. Początkowymi założeniami projektu było uzyskanie:

- automatycznego pozyskiwania danych o ofertach konkurujących podmiotów,
- wyznaczania miary atrakcyjności i pozycjonowania oferty w stosunku do konkurencji,
- znalezienie odpowiedniej bazy danych gromadzącej oferty,
- zagwarantowanie odświeżania ofert co 24h,
- porównywania ofert, analizy zmian w ofertach w zadanym okresie czasu oraz wizualizacji otrzymanych wyników.

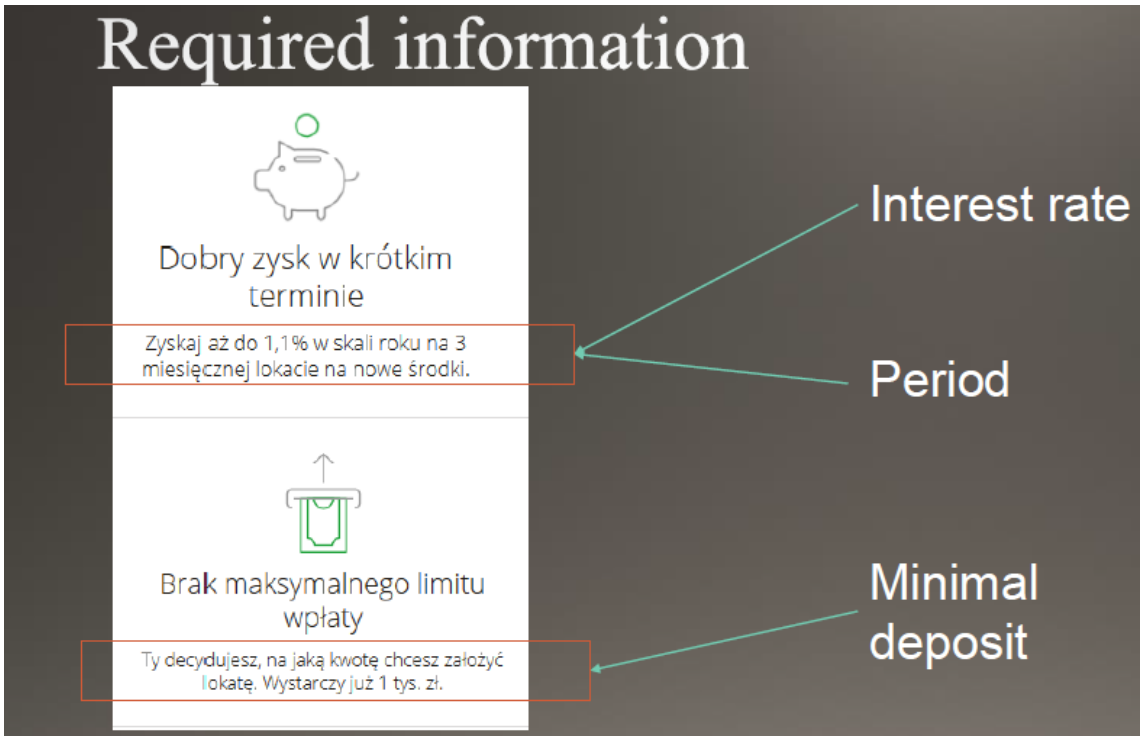
Poza zaprojektowaniem i zbudowaniem architektury z uwzględnieniem spisanych powyżej celów końcowych okazało się, iż projekt wymaga skupienia się na znalezieniu różnych komponentów umożliwiających alternatywne ścieżki zbudowania rozwiązania, ich opisanie i porównanie. Projekt został oparty o rozwiązania chmurowe, w tym celu wybrano platformę Google Cloud Platform (GCP) oraz Amazon Web Services (AWS). Rozwiązanie budowane jest w oparciu o narzędzia dostępne na tychże platformach i między tymi platformami dokonywane jest porównanie. Zgodnie z ustaleniami z PKO BP, projekt opiera się na gromadzeniu i analizie danych dotyczących lokat na nowe środki w wybranych bankach.

2 Scrappowanie

Pozyskiwane dane bardzo często znajdują się na stronach internetowych danych podmiotów. Każda ze stron może się od siebie różnić, a zawarte informacje mogą być przedstawione w inny sposób. Na rysunkach 1 oraz 2 zaprezentowano wycinki stron z banków PKO oraz mBank.



Rysunek 1: Bank PKO - lokata terminowa[1]



Rysunek 2: mBank - lokata na nowe środki[2]

Głównym problemem przy ekstrakcji danych ze stron, jest znaczna rozbieżność między ich dokumentami HTML. Ponieważ taki dokument może składać się z wielu różnych zagnieżdżonych elementów , <ol>, <ul>, <table>, <script> itd. Powoduje to, że każdą z tych stron należy analizować oddzielnie. Dodatkowo, dane w nich zawarte to głównie dane liczbowe i mogą być zaprezentowane w różny sposób. Duże liczby jak np. 10000, mogą być przedstawione jako 10 000, albo 10.000. Liczby ułamkowe mogą być zapisane z przecinkiem (0,50) albo z kropką (0.50). Mogą być też opisane słownie - 1000 jako tysiąc, bądź tys. Tekst może również zawierać przyrostki, przedrostki i znaki interpunkcyjne.

Najłatwiejszy sposób na uzyskanie informacji ze strony to zbadanie dokumentu HTML. Po pobraniu kodu HTML i jego sparsowaniu, należy znaleźć element zawierający dane, a następnie je wyciągnąć. Te dane poddawane są ewentualnej konwersji, na przykład na typ liczbowy. Plusem takiej metody jest prostota, a co za tym idzie - łatwość napisania programu do scrappowania. Dodatkowo, gdyby struktury kodu HTML poszczególnych stron nie różniłyby się znacząco od siebie to przy dodatkowej pracy można by było uzyskać jeden wspólny kod wydobywający dane. Każda witryna jednak znacząco się różni, przez co wymaga to indywidualnego stworzenie kodu do scrappowania dla każdej ze stron. Problemem staje się również zmiana kodu takiej strony, co pociąga za sobą re-implementacje utworzonego programu oraz ponownego przejrzenia kodu HTML w poszukiwania elementów zawierających dane do ekstrakcji.

W kolejnych podpunktach zostanie zaprezentowana inna metoda pozwalająca na chociaż częściowe wyeliminowanie powyższych problemów. Całość zostanie zaimplementowana w języku `Python`

2.1 Ogólna koncepcja rozwiązania

W projekcie została zaimplementowana inna metoda. Polega na utworzeniu wzorców. Zadziałają one jako filtry, którymi będzie można przeszukiwać tekst zawarty na stronie i odzyskiwać z nich informacje. Dla przykładu, niech ten wzorzec będzie pod postacią pewnego klucza (KEY) oraz wartości (VALUES). Należy przyjąć, że znany jest dany klucz, natomiast wartości przyjmują niewiadome, ale mają taki sam opis (np. mogą cechować się tym, że są liczbami). Gdy posiadamy tekst, to można go podzielić na tokeny. Przykładowo, przyjmując tekst zawierający informacje o oprocentowaniu z rysunku 1, po jego procesie tokenizacji otrzymamy listę tokenów:

$$[\text{"}Zyskujesz\text{"}, \text{"}stale\text{"}, \text{"}oprocentowanie\text{"}, \text{"}do\text{"}, \text{"}0,50\text{"}, \text{"}\%\text{"}, \text{"}w\text{"}, \text{"}skali\text{"}, \text{"}roku\text{"}]$$

Z niej, pozostaną tylko te tokeny, które są zgodne ze wzorcem. Przyjmując, że poszukiwanym kluczem będzie "oprocentowanie", a wartością jest pewna wartość liczbowa, to można odfiltrować listę tokenów, otrzymując:

$$[\text{"}oprocentowanie\text{"}, \text{"}0,50\text{"}]$$

Łącząc te dwa tokeny w parę KLUCZ:WARTOŚĆ, można odzyskać informację, że oprocentowanie wynosi 0,50%. Taka metoda po rozwinięciu, będzie mogła pozwalać na przeszukiwanie informacji zawartych w tekście i nie będzie ograniczona względem źródła pochodzenia tego tekstu.
Cały proces pozyskiwania danych, można podzielić na kilka etapów:

1. Pozyskiwanie tekstu - etap, który pozwoli na pozyskanie danych do przetworzenia, w tym przypadku ze strony internetowej. Jest to ważny proces, ponieważ błąd w pozyskaniu danych może powodować ich utratę. W przypadku innych nośników danych, można przystąpić analogicznie w tworzeniu metod do wyciągania danych.

2. Przetworzenie tekstu - na ten etap składa się tokenizacja pozyskanego tekstu na tokeny, łączenie i rozbijanie ich, filtrowanie zbędnych tokenów i nadawanie etykiet pozostałym. Przy tokenizacji wykorzystana zostanie zewnętrzna biblioteka. Proces łączenia i rozbicia tokenów, będzie głównie dotyczył liczb, w którym posłużymy się funkcją utworzoną specjalnie do tego celu. Z kolei do filtrowania i etykietowania posłuży jedna funkcja mapująca tokeny do obiektów specjalnej klasy opisującej dany token za pomocą etykiety.

3. Tworzenie wzorców - etap, w którym zostaną utworzone filtry klucza i wartości. Razem będą tworzyć strukturę, za pomocą której będziemy przetwarzać listę tokenów. Wynikiem tej operacji - w przypadku znalezienia informacji zgodnych z tym wzorcem - informacje w postaci obiektów KLUCZ:WARTOŚĆ.

4. Sformatowanie otrzymanych danych - Z poprzedniego etapu, dla listy wielu wzorców otrzymamy listę wielu struktur KLUCZ:WARTOŚĆ. Etap ten, ma na celu odzyskanie informacji z tej listy podając wartość klucza oraz sformatowanie danych, na przykład rzutowanie na typ liczbowy, znalezienie maksimum w przypadku listy wielu wartości itp.

2.2 Pozyskiwanie tekstu

Zanim będzie można wykonać odpowiednie działania na tekście, należy go uprzednio wyciągnąć ze strony wraz z zachowaniem informacji. Do tego celu wykorzystano bibliotekę *BeautifulSoup*. Pozwala ona na sprasowanie pobranego kodu HTML, dzięki czemu, możliwe będzie poruszanie się i modyfikacja powstałego, sprasowanego drzewa HTML.

Na wstępie należy przyjąć parę założeń i warunków, co pomoże w uspójnieniu tworzonego programu:

- Nie jest znana wewnętrzna struktura kodu HTML stron internetowych;
- Nie jest znany, jaki element strony zawiera potrzebne informacje;
- Znane są natomiast elementy, składające się, na taki dokument, m.in. <html>, <body>, <head>, <script>, , <ul>, <ol>, <il>, <table>, itd.;
- Pewne elementy, takie, jak <footer> albo <head> zawierają albo informacje o właścicielu danej strony, albo o samej stronie. Te elementy, oraz elementy zawierające skrypty <script> czy elementy <form> powinny zostać usunięte lub pominięte w analizie;
- Można przyjąć, że element typu , służą do podziału strony na sekcje;

Biblioteka *BeatifulSoup* pozwala pobrać cały tekst ze strony za pomocą własności `text`. Pobierany jest cały tekst, włącznie z zawartością takich elementów jak <footer> i <script>. Można się ich pozbyć, iterując po wskazanych elementach oraz usuwając je.

```
for e in html.find_all footer['footer', 'script']
    e.extract()

html_text = e.text
```

Tutaj występuje następny problem. Jak będzie wyglądał wyciągnięty tekst? Dla porównania, wykorzystano prosty kod HTML w listingach 1 oraz 2. Różnicę stanowią dodatkowe dwa znaki nowej linii w listingu 2.

Listing 1

```
html_text = BeatifulSoup(
"<body>"
    "<div>"
        "<p>1</p>"
        "<div>"
            "<p>2</p>"
        "</div>"
    "</div>"
    "<div>"
        "3"
    "</div>"
"<body>")
```

Listing 2

```
html_text = BeatifulSoup(
"<body>"
    "<div>"
        "<p>1</p>"
        "<div>"
            "\n"
            "<p>2\n</p>"
        "</div>"
    "</div>"
    "<div>"
        "3"
    "</div>"
"<body>")
```

Po wyciągnięciu z nich tekstu, otrzyma się dwa różne rezultaty. Z pierwszego otrzyma się połączony ciąg znaków ”123”. Z drugiego otrzymamy 3 osobne linie. Lepszym wynikiem byłoby uzyskanie wszystkich ciągów znaków rozdzielonych spacją - ”1 2 3”. Do tego celu utworzono specjalną funkcję `collapse()`, której kod pokazano na listingu 3. Jej głównym zadaniem jest skanowanie drzewa HTML i poszukiwanie wszystkich elementów typu `elem_name`. Gdy zostanie znaleziony, to w następnej kolejności szukane jest dziecko tego elementu o tym samym typie. Gdy nie napotka żadnego dziecka rozpoczyna łączenie przetworzonego tekstu wchodząc w górę drzewa. Pod koniec, zostaje zwrócony zmodyfikowany dokument HTML, zawierający tylko jeden element, typu `elem_name` zawierający

Listing 3

```
def collapse(html_fragment, elem_name: str, start_depth: int = 0,
min_depth: int = 0, make_copy=True, agg_func=None):
    if make_copy:
        html_fragment = copy.copy(html_fragment)

    stack = deque()
    elem = html_fragment.find(elem_name)
    while elem:
        stack.append((elem, start_depth))
        elem = elem.find_next_sibling(elem_name)

    while stack:
        parent = stack[-1]
        child = parent[0].find(elem_name)
        depth = parent[1] + 1

        if child:
            while child:
                stack.append((child, depth))
                child = child.find_next_sibling(elem_name)
        else:
            parent[0].name = '__COLLAPSED_' + parent[0].name.upper() + "__"
            parent[0]['__COLLAPSED_DEPTH__'] = str(parent[1])

            if int(parent[0]['__COLLAPSED_DEPTH__']) >= min_depth:
                parent[0].string = agg_func(parent[0],
                        parent[0]['__COLLAPSED_DEPTH__']) if agg_func \
                    else " " + " ".join((parent[0].get_text() + " ").split()) + " "

            stack.pop()
    return html_fragment
```

długi ciąg znaków, którę łączą wszystkie węzły tekstowe. Po zastosowaniu tej funkcji podając element grupujący jako ”div” z kodu, z listingów 1 oraz 2 otrzyma się ten sam wynik - ”1 2 3”.

2.3 Przetworzenie tekstu

W następnej kolejności można wykonać przetworzenie tekstu. Na wstępie, należy tekst podzielić na tokeny. Z pomocą przychodzi biblioteka *nltk* do przetwarzania języka naturalnego. Nie ma wsparcia dla polskiej wersji, ale zawiera ona funkcję `word_tokenize`, która potrafi sprawnie podzielić tekst na tokeny. Listing 4 zawiera przykładowy kod HTML, który zostanie w ramach tej sekcji przetwarzany. Stosując metodę z sekcji 2.2 do wyciągnięcia tekstu, otrzyma się dane zawierające się w listingu 5.

Listing 4

```
<html>
    <head><title>Przykładowa strona</title></head>
    <body>
        <div>
            <div>
                <div>
                    <ul>
                        <li>XXXX Oprocentowanie</li>
                        <li>wynosi</li>
                        <li>(0,50%) </li>
                        <li> XXX </li>
                    </ul>
                    <p>XXX oprocentowanie stałe XXX</p>
                </div>
            </div>
        </div>
        <div>
            <div>
                <p>XXXX</p>
                <p>Depozyt wynosi</p>
                <p>10 000zł XXX<p>
                <p>XXXX</p>
            </div>
        </div>
        <div>
            <p>XXXX czas oszczędzania: 1, 3, 6 lub 12 miesięcy XXXX</p>
        </div>
    <footer>
        <p>Informacje niepotrzebne<p>
        <ul>
            <li>Adres</li>
            <li>Telefon</li>
        </ul>
    </footer>
    </body>
</html>
```

Listing 5

```
"""Przykładowa strona XXXX Oprocentowanie wynosi (0,50%) XXX XXX oprocentowanie stałe
XXXX Depozyt wynosi 10 000zł XXX XXXX XXXX czas oszczędzania: 1, 3, 6 lub 12 miesięcy
XXXX Informacje niepotrzebne Adres Telefon"""
```

Po zastosowaniu funkcji `word_tokenize` z pakietu *nltk* otrzyma się listę tokenów:

Listing 6

```
["Przykładowa", "strona", "XXXX", "Oprocentowanie", "wynosi", "(", "0,50", "%", ")",
"XXX", "XXX", "oprocentowanie", "stałe","XXXX", "Depozyt", "wynosi", "10", "000zł",
"XXX", "XXXX", "XXXX", "czas", "oszczędzania", ":", "1", ",", "3", ",", "6", "lub",
"12", "miesięcy", "XXXX", "Informacje", "niepotrzebne", "Adres", "Telefon"]
```

Można zauważyć, że w tym przypadku liczba "10000" została źle zapisana. Otrzymano dwa tokeny "10" oraz "000zl". Dlatego w następnej kolejności należy przetworzyć otrzymaną listę w poszukiwaniu tych tokenów, które należałoby ponownie złożyć ze sobą. Zaimplementowana funkcja `mergeNumberTokens()` służy do takich przypadków. Jej głównym zadaniem jest złożenie liczb do jednego formatu i pozbycie się wszystkich zawartych sufiksów. Dla przykładu niech pewien tekst zawiera liczbę opisaną w postaci ciągu znaków "1 200 001,055". Po tokenizacji, otrzyma się listę tokenów: ["1", "200", "001,055"]. Funkcja sprawdza na początku, czy dany token może być przedstawiony jako liczba całkowita. Jeżeli tak, to trafiają do bufora. W tym przypadku do bufora trafiają tokeny "1" oraz "200". Jeżeli nie, to dany token sprawdzany jest wyrażeniem regex "(+([.?,?]?+)?)", czyli sprawdza czy token jest liczbą zapisaną z przecinkiem lub kropką. Jeżeli tak, to dodaje do bufora ten token. Na koniec łączy wszystkie tokeny zawarte w buforze w jeden. Ponieważ token "001,055" pasuje do opisanego wcześniej wyrażenia regex, to również trafia do bufora. Zawarte w buforze elementy, czyli ["1", "200", "001,055"], zostają złączone zwracając ciąg "1200001,055". Funkcja ta nie zamienia przecinka na kropkę, ponieważ taki zapis liczby może być wymagany przez użytkownika.

Po zastosowaniu funkcji `mergeNumberTokens()` otrzyma się:

Listing 7

```
tokens = ["Przykładowa", "strona", "XXXX", "Oprocentowanie", "wynosi", "(", "0,50",
"%", ")", "XXX", "XXX", "oprocentowanie", "stałe","XXXX", "Depozyt", "wynosi", "10000",
"zł", "XXX", "XXXX", "XXXX", "czas", "oszczędzania", ":", "1", ",", "3", ",", "6",
"lub", "12", "miesięcy", "XXXX", "Informacje", "niepotrzebne", "Adres", "Telefon"]
```

Mając taką listę należałoby odfiltrować tylko te słowa, które mają znaczenie w poszukiwaniu danych. Klasa *Keyword*, pozwala na stworzenie obiektu, który zawiera dany token oraz opisujące je etykiety. Na listingu 8 wykorzystano tą klasę do opisania tokenów, są to m.in "Depozyt" z etykietą "DEPOSIT"; "Oprocentowanie" z etykietą "PERCENTAGE"; "%" jako "SUFFIX" oraz "PERCENTAGE"; "zł", jako "SUFFIX", "CURRENCY"; "miesięcy", jako "DATE" oraz wszystkie liczby. Sufiksy oraz liczby można opisać również za pomocą wyrażenia regex.

Listing 8

```
Keys = Keyword.Keywords()
Keys.addKey(Keyword.Keyword("Depozyt", labels={"DEPOSIT"})). \
    addKey(Keyword.Keyword("Oprocentowanie", labels={"PERCENTAGE"})). \
    addKey(Keyword.Keyword("%", labels={"SUFFIX", "PROCENTAGE"})). \
    addKey(Keyword.Keyword(labels={"SUFFIX", "CURRENCY"},
                           regex=r"(zł[\W]?)")). \
    addKey(Keyword.Keyword(labels={"NUMBER"},
                           regex=Keyword.NUM_PATTERN)). \
    addKey(Keyword.Keyword("miesięcy", labels={"DATE"}))
```

Funkcja `lineToKeys()` pozwala zmapować token do określonego obiektu klasy Keyword. Można wypisać taką listę do konsoli, która przedstawia każdy z obiektów Keyword jako element typu *dict*, którego kluczem jest zmapowany token, a wartościami są etykiety:

Listing 9

```
line2key = Keyword.lineToKeys(tokens, Keys)
print(line2key)
"""
[{'Oprocentowanie': {'PERCENTAGE'}}, {'19': {'NUMBER'}},
{'%': {'PERCENTAGE', 'SUFFIX'}}, {'Oprocentowanie': {'PERCENTAGE'}},
{'Depozyt': {'DEPOSIT'}}, {'10000': {'NUMBER'}},
{'zł': {'CURRENCY', 'SUFFIX'}}, {'1': {'NUMBER'}},
{'3': {'NUMBER'}}, {'6': {'NUMBER'}},
{'12': {'NUMBER'}}, {'miesięcy': {'DATE'}}]
"""
```

2.4 Tworzenie wzorców

Na sam koniec należy stworzyć pewien wzorzec, który będzie przeszukiwał daną listę. Podstawowym elementem, do utworzenia wzorca jest klasa `Slot`.

Listing 10

```
class Slot:
    def __init__(self, whitelistkey: Set[str] = None,
                 blacklistkey: Set[str] = None,
                 whitelistlabel: Set[str] = None,
                 blacklistlabel: Set[str] = None,
                 max: int = 1):

s11 = Pattern.Slot(whitelistkey={"oprocentowanie"})
s12 = Pattern.Slot(whitelistlabel={"NUMBER"}, max=2)
s13 = Pattern.Slot(whitelistlabel={"SUFFIX"})
```

Pozwala ona na określenie filtru na podstawie zawartości klucza i etykiet, za pomocą list białych - "whitelistkey", "whitelistlabel" - oraz czarnych - "blacklistkey" i "blacklistlabel". Dodatkowo pozwala ustawić maksymalną powtarzalność danego obiektu *Keyword*. Na przykładzie slotu "s12" z listingu 10, będzie pobierała tylko te klucze, których etykieta

zawiera wartość "NUMBER" oraz może pobrać maksymalnie dwa takie klucze z rzędu - zgodnie z parametrem "max". Takie obiekty, będą służyć do inicjalizacji klasy `Pattern`, jak pokazano na listingu 11. Potrzebuje ona głównego slotu "slot" oraz możliwych lewych i/lub prawych slotów:

Listing 11

```
class Pattern:
    def __init__(self, id, slot: Slot, right_slots: list = None,
                 left_slots: list = None):

patternA = Pattern.Pattern(id='INTEREST', slot=s11, right_slots=[s12, s13])

s21 = Pattern.Slot(whitelistkey={"depozyt"})
patternB = Pattern.Pattern(id='INTEREST', slot=s21, right_slots=[s12, s13])
pattern_list = [patternA, patternB]
```

W powyższym przykładzie został utworzony wzorzec do szukania tokenu zawierającego wartość "oprocentowanie". Po jego prawej stronie kolejno mają znajdować się tokeny z etykietą "NUMBER" oraz "SUFFIX". Zdefiniowane wcześniej sloty można wykorzystać wielokrotnie. Dlatego zdefiniowano slot główny przyjmujący token "depozyt" i wykorzystano sloty 's12' oraz 's13' do utworzeniu wzorca wyszukującego wartość minimalnego depozytu.

Utworzoną listę wzorców 'pattern_slot' z listingu 11 można wykorzystać w funkcji *processLineWithPatterns()*. Pobiera ona listę obiektów typu Keyword oraz listę wzorców rozpoczynając przeszukiwanie danych dla każdego wzorca. Algorytm na samym początku przeszukuje listę Keywords w celu znalezienia pasującego elementu do głównego slotu z danego wzorca. Jeżeli został znaleziony, następuje porównanie tokenów po prawej stronie ze slotami prawymi i analogicznie ze stroną lewą. Domyślnie funkcja zwraca listę obiektów *dict*, składających się z klucza otrzymanego po filtracji ze slotu głównego oraz z wartości stanowiącej listę wszystkich slotów zawierających klucze z lewej i z prawej strony slotu głównego.

2.5 Sformatowanie otrzymanych danych

Zwracane dane mają format, który jest trudny w analizie. Aby je sformatować zostały utworzone klasy `SchemaItem` oraz `SchemaContainer`. Obiekt klasy `SchemaItem` zawiera informacje czy poszukiwane dane mogą być puste (NULL). Dodatkowo może zawierać funkcję filtrującą oraz funkcję grupującą. Funkcja filtrująca pozwoli na otrzymanie odpowiednich danych i ich przetworzenie np. liczbę przedstawioną jako ciąg znaków, może zamienić na typ float. Może się okazać również, że otrzymanych informacji o tym samym kluczu jest wiele. Funkcja grupująca pozwala na określenie, który element jest decydujący.

Listing 12

```
class SchemaItem:
    def __init__(self, key: str, filter_, nullable: bool = False,
                 group_choice: Callable = None):
        self.key = key
        self.nullable = nullable
        self.filter = filter_
        self.group_choice: Callable = group_choice
```

Klasa *SchemaContainer* wymaga podania jej listy otrzymanej z funkcji *processLineWithPatterns()*. Po przekazaniu do metody listy obiektów `SchemaItem` otrzyma się ostatecznie obiekt typu *dict* z kluczem o wartości podanej w obiekcie `SchemaItem` i o wartości otrzymanej z funkcji filtrującej i/lub grupującej. Poniższy przykład ilustruje użycie:

Listing 13

```
schema = Pattern.SchemaContainer(pattern_result)
d = schema.createFromSchema([Pattern.SchemaItem("oprocentowanie",
                 filter_=lambda k, s: Keyword.toNumber(s[0][0]))])
print(d) # 'oprocentowanie': 0,50
```

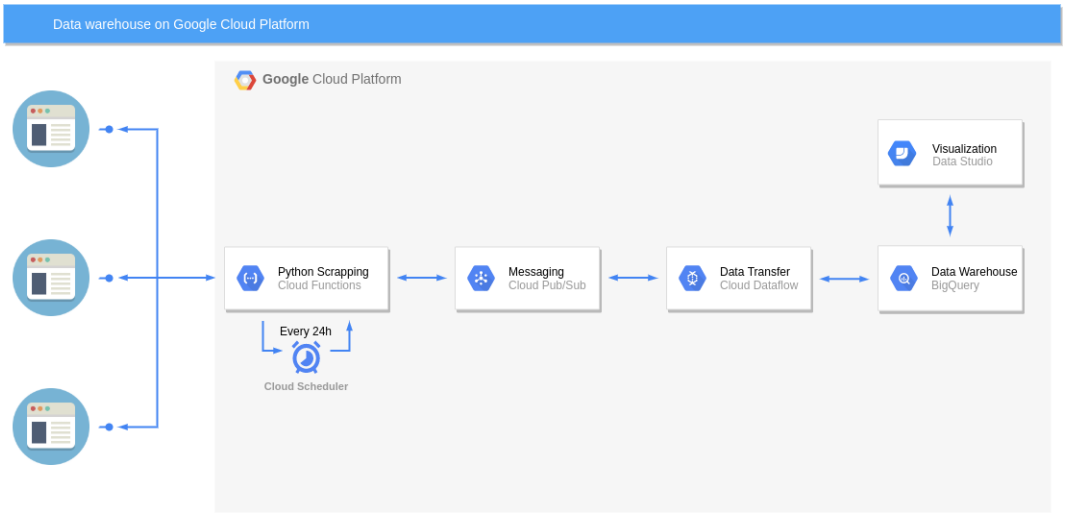
W tym przypadku poszukiwany jest element o wartości klucza "oprocentowanie". Można również wykorzystać etykietę klucza, jednak należy mieć pewność, że jest ona unikalna wśród innych obiektów typu KEYWORD utworzonych w punkcie 2.2. Podając pewną funkcję do argumentu *filter_* można sformatować otrzymane dane. W tym przypadku do funkcji zostanie przekazany obiekt KEYWORD(parametr 'k') oraz listę slotów zawierające wszystkie dane(parametr s). Ponieważ we wzorze z listingów 10 oraz 11, mamy dwa sloty - jeden zawiera liczbę, drugi sufiks - to wartość z pierwszego slotu, przekształcamy na typ numeryczny. Należy pamiętać, że parametr 's', jest listą, stąd wynika podwójny dostęp za pomocą kwadratowych nawiasów. Ten etap kończy część scrappującą i odzyskującą informacje.

3 Hurtownia danych na GCP

Google Cloud Platform to zbiór usług chmurowych oferowanych przez Google, które funkcjonują na infrastrukturze wykorzystywanej przez takie produkty jak Gmail, YouTube lub Google Search. Oprócz zbioru narzędzi, służących do zarządzania, GCP udostępnia również takie usługi, które pozwalają na przechowywanie danych, analizę, uczenie maszynowe czy też samo przetwarzanie na życzenie.

3.1 Architektura systemu

Poniższy diagram przedstawia architekturę systemu opartego o GCP (rysunek nr 3):



Rysunek 3: Architektura systemu zbudowanego na GCP

3.2 Opis poszczególnych komponentów systemu

Niewątpliwie architektura ta cechuje się prostotą, a przygotowanie takiego systemu zajmuje maksymalnie kilka godzin. Postaramy się teraz opisać poszczególne komponenty systemu.

3.2.1 Cloud Scheduler

Wszystko zaczyna się od ofert bankowych, konkretnie skupiliśmy się na lokatach na nowe środki. Co 24 godziny narzędzie o nazwie Cloud Scheduler, uruchamia Google Cloud Function (GCF), której zadaniem jest zebranie ofert lokat ze stron wybranych banków. Opis Cloud Schedulera znaleźć możemy w dokumentacji[3].

> Cloud Scheduler is a fully managed enterprise-grade cron job scheduler. It allows you to schedule virtually any job, including batch, big data jobs, cloud infrastructure operations, and more. You can automate everything, including retries in case of failure to reduce manual toil and intervention. Cloud Scheduler even acts as a single pane of glass, allowing you to manage all your automation tasks from one place.

Komponent ten ponadto cechuje się - jak zapewnia Google - niezawodnością, ze względu na to iż jest on rozproszony. Unikamy więc problemu tzw. ”single point of failure”.

3.2.2 Google Cloud Functions

Jeżeli chodzi z kolei o GCF, to opis tego komponentu możemy znaleźć w dokumentacji[4]. Komponent ten umożliwia uruchamianie przygotowanych fragmentów kodu. Cechuje się on tym, iż jest tzw. "serverless" - płacimy tylko za zasoby i moc obliczeniową, którą faktycznie wykorzystujemy. Jeżeli funkcja wykonuje się raz na dobę przez kilkanaście sekund - płacimy tylko za to wykonanie. Jesteśmy zwolnieni z kosztownego utrzymywania całych serwerów. Komponent ten automatycznie skaluje się, tzn. zwiększa zasoby w zależności od zapotrzebowania.

W ramach GCF wykonywane są skrypty Python z użyciem biblioteki BeautifulSoup, który to zajmuje się scrapowaniem stron odpowiednich banków. Ideą jest, aby programista nie musiał zaglądać do kodu HTML stron. Zamiast tego, na podstawie wyglądu wizualnego, może określić jakie informacje są potrzebne, a następnie za pomocą odpowiednich funkcji wyciągnąć je ze stron.

3.2.3 Pub/Sub

Oferty takie po pobraniu ich ze stron banków publikowane są w ramach tematu Pub/Sub o nazwie "deposit-offer". Pub/Sub to kolejna usługa oferowana w ramach GCP, jest to (wg dokumentacji[5]):

> fully-managed real-time messaging service that allows you to send and receive messages between independent applications.

3.2.4 Google Cloud Dataflow

Następnie, przy pomocy narzędzia o nazwie Google Cloud Dataflow[6], oferty pobierane są z tematu Pub/Sub i transferowane do Google BigQuery.

> Dataflow is a fully managed streaming analytics service that minimizes latency, processing time, and cost through autoscaling and batch processing. With its serverless approach to resource provisioning and management, you have access to virtually limitless capacity to solve your biggest data processing challenges, while paying only for what you use.

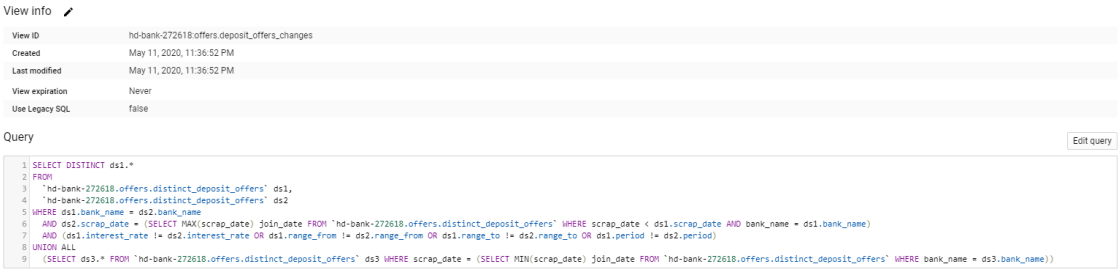
Do jego głównych zalet bez wątpienia należy możliwość horyzontalnego skalowania, dzięki czemu jest on w stanie przetwarzać bardzo duże wolumeny danych w sposób efektywny. Ponadto zapewnia jednorodny model przetwarzania wsadowego i strumieniowego, niezawodność i przetwarzanie masywnych danych "dokładnie raz".

3.2.5 Google BigQuery

Prosto z Dataflow dane trafiają do Google BigQuery[7]. Jest to hurtownia danych w chmurze Google. BigQuery to kolejne wysoce skalowalne rozwiązanie z rodziny "serverless". Zapewnia ono możliwość przetwarzania nawet petabajtów danych, przy dużej szybkości realizacji poleceń. Tutaj również płatności odbywają się według zużycia przestrzeni dyskowej, bez martwienia się o trwałość i awarie, ponieważ dane są replikowane. Komunikacja z BigQuery odbywa się za pomocą poleceń SQL, ponadto można wykonywać zapytania używając REST API.

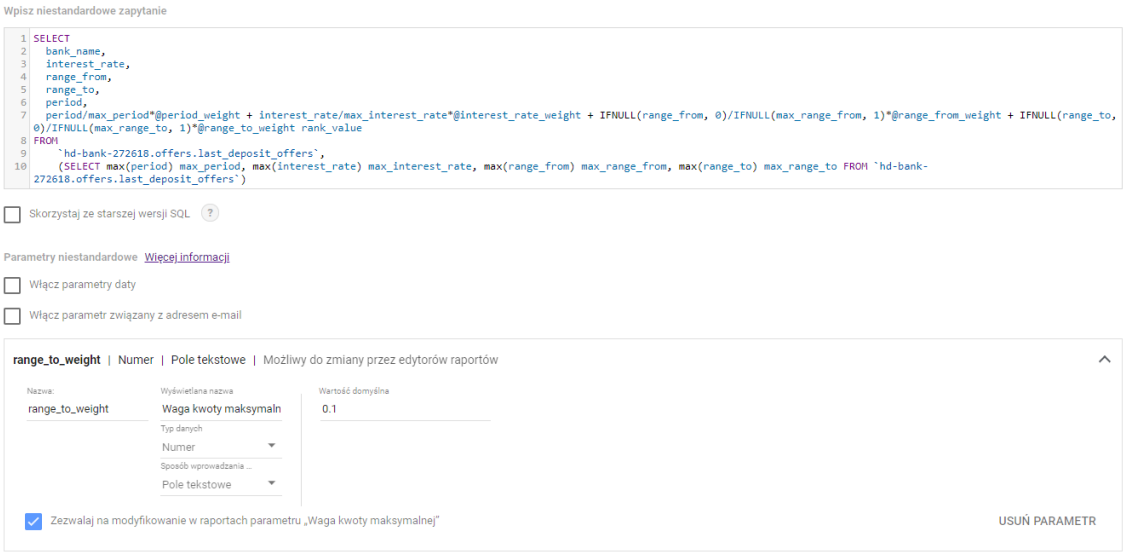
3.2.6 Google Data Studio

Jest to usługa służąca do wizualizacji danych, która pozwala na tworzenie złożonych raportów. Jedną z zalet tej platformy jest możliwość wykorzystywania wielu źródeł danych jednocześnie, a duża liczba udostępnionych konektorów nie ogranicza nikogo do skorzystania wyłącznie z usług Google. Warto również zauważyć, że Google Data Studio[8] jest udostępniane za darmo. W naszym rozwiązaniu rolę źródeł danych wyświetlanych na raporcie pełnią m.in. perspektywy stworzone w usłudze Google BigQuery. Perspektywy te przygotowują wstępnie dane wyświetlane następnie na raporcie. Data Studio pozwala również rozszerzyć wybrane źródło danych o dodatkowe (w pełni konfigurowalne) dynamicznie wyliczane kolumny.



Rysunek 4: Przykładowa perspektywa przygotowująca dane w BigQuery

Wybór źródła danych nie ogranicza się wyłącznie do wyboru tabeli/perspektywy w wybranej usłudze z której pobierane są dane. Data Studio umożliwia zdefiniowanie niestandardowego zapytania, które dodatkowo może być sparametryzowane. Wartości zdefiniowanych parametrów można następnie wprowadzić w widoku edycji w wybranych komponentach wyświetlających dane.



Rysunek 5: Niestandardowe zapytanie pobierające dane z BigQuery

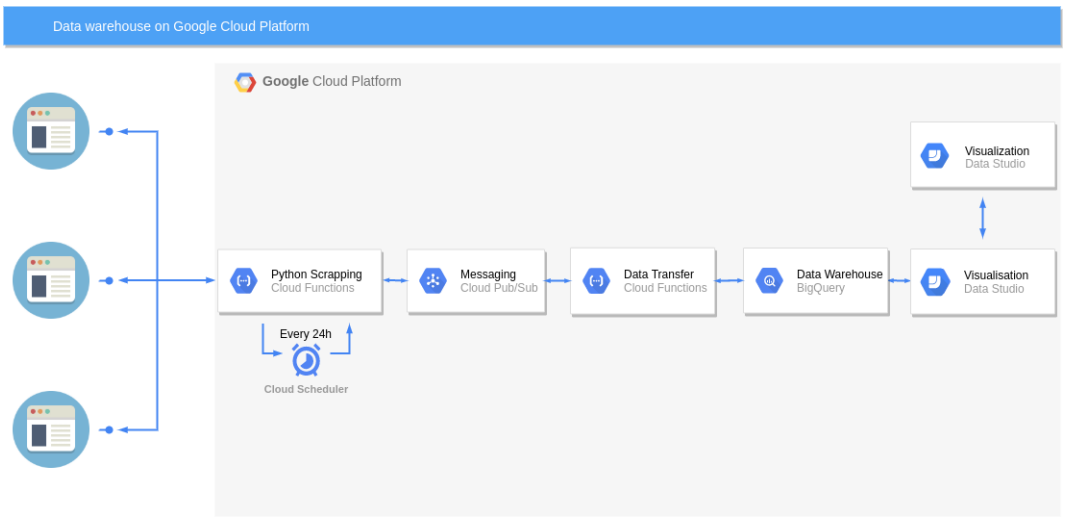
Stworzony raport jest tak właściwie aplikacją webową wprowadzającą możliwość interakcji. Osoba korzystająca z gotowego raportu poprzez wcześniej zdefiniowane i udostępnione filtry może manipulować zakresem wyświetlanych danych czy też ograniczyć się tylko do wybranych banków.

3.3 Wady rozwiązania

Do wad rozwiązania należą stosunkowo duże koszty. Większość z nich jest jednak generowana przez narzędzie Google Cloud Dataflow ($10 dziennie). Koszty pozostałych komponentów są na poziomie kilku centów miesięcznie. W związku z tym naszym następnym zadaniem będzie próba zastąpienia tego komponentu czymś bardziej efektywnym kosztowo.

3.4 Redukcja kosztów

Wadą zaprojektowanej w oparciu o Google Cloud Platform architektury był koszt generowany przez narzędzie Google Cloud Dataflow użyte do przesyłania danych pomiędzy tematem Pub/Sub a BigQuery. W celu zredukowania kosztów usunięto wspomniane narzędzie i zastąpiono je komponentem GCF opisanym w punkcie 3.2.2. Skrypt umieszczony w GCF napisany jest w języku Python, natomiast konfiguracja w tym przypadku działa trochę inaczej niż w przypadku scrappowania stron. Kod funkcji nie jest uruchamiany przy pomocy Cloud Schedulera, tylko ustawiony jako wyzwalacz (z ang. trigger) na temacie Pub/Sub. Skonfigurowanie tego w ten sposób powoduje, że funkcja uruchamia się gdy tylko w temacie Pub/Sub pojawią się dane. Jak już wcześniej zostało wspomniane GCF generuje niewielkie koszty co oznacza iż ta wada została wyeliminowana. Poprawiona architektura systemu znajduje się poniżej.



Rysunek 6: Architektura systemu zbudowanego na GCP (po redukcji kosztów)

3.5 Lokalizacja danych

W momencie aktywowania różnych komponentów mamy możliwość wyboru w jakim regionie powinny być składowane dane. Oprócz wyboru podczas tworzenia/ aktywowania komponentów można skonfigurować wszystko tak aby członkowie projektu nie mieli możliwości korzystania z niechcianych regionów. Mamy możliwość wyboru wielu różnych regionów, które podzielone są na cztery różne strefy: Ameryka, Europa, Azja Pacyfik oraz multi-region. Aktualnie w Europie mamy do wyboru 6 różnych miejsc składowania danych a najbliższą Polsce jest Frankfurt (europe-west3). W 2019 roku Google zapowiedział stworzenie nowego regionu w Warszawie, jednak na chwilę obecną nie zostało to jeszcze zrealizowane.

3.6 Wskazówki konfiguracyjne

Podczas tworzenia rozwiązania bardzo przydatne okazały się następujące artykuły:

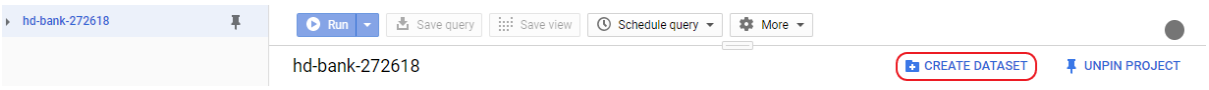
- Running a scraping platform at Google Cloud for as little as US$0.05month [9]
- Web scraping with Google Cloud Functions, Pub/Sub, DataFlow & BigQuery [10]

Zawarta w nich wiedza bez problemu pozwoli na zbudowanie rozwiązania, o którym mowa w projekcie. Sama platforma GCP jest też dość intuicyjna i łatwa w obsłudze, ponadto Googla zapewnia obszerną dokumentację.

Ponadto razem z dokumentacją załączamy kod wykorzystywany w ramach Cloud Functions (opisanych w punkcie 3.2.2) na platformie GCP. W następnych kilku podpunktach zostanie w skrócie pokazana konfiguracja wszystkich użytych komponentów w projekcie.

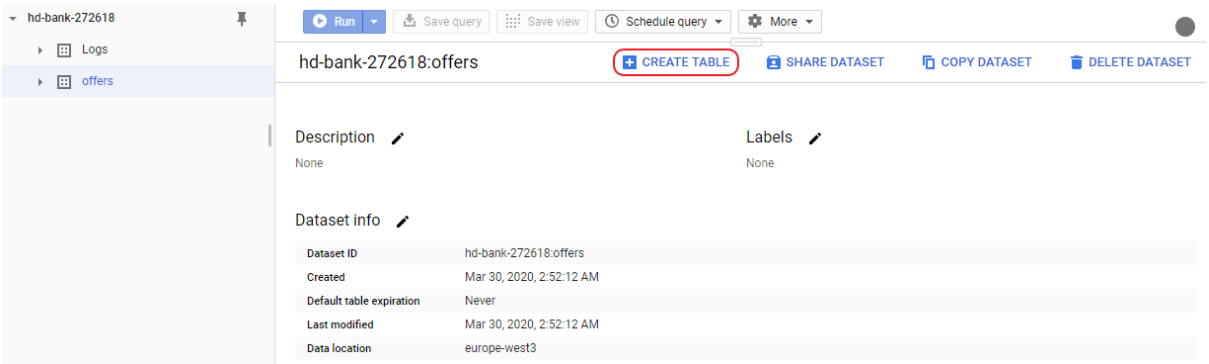
BigQuery

W usłudze BigQuery tworzymy dwa zbiory danych (rys. 7). Pierwszy z nich będzie zawierał tabele i perspektywy dotyczące ofert bankowych, a drugi tabelę zawierającą logi przetwarzania. Podczas tworzenia zboru danych istnieje możliwość wskazania miejsca przechowywania danych. W naszym przypadku jest to *europe-west3* (Frankfurt).



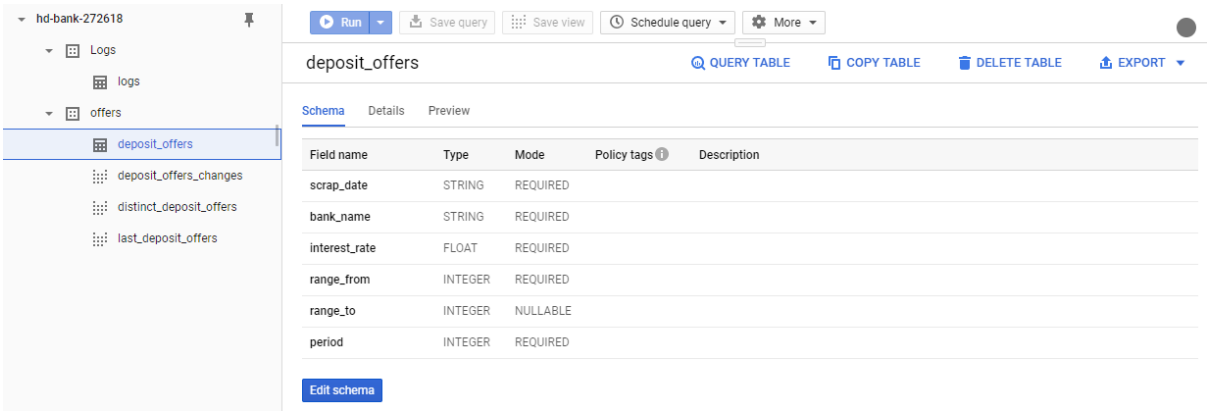
Rysunek 7: Tworzenie zbioru danych

Następnym krokiem jest stworzenie tabel w każdym z wcześniej utworzonych zbiorów danych (rys. 8). Podczas tworzenia tabeli możemy wskazać czy tabela ma powstać na podstawie wybranego pliku z danymi. W naszym przypadku tabela tworzona jest od zera. Tworzenie schematu opiera się na podaniu nazwy parametru, jego typu (np. string, float). Dodatkowo istnieje możliwość ustawienia sposobu partycjonowania danych. Perspektywy w usłudze BigQuery można stworzyć zapisując zapytanie odnoszące się do interesujących nas danych za pomocą widocznego na samej górze rysunku (rys. 8) przycisku *Save view*.



Rysunek 8: Tworzenie tabel

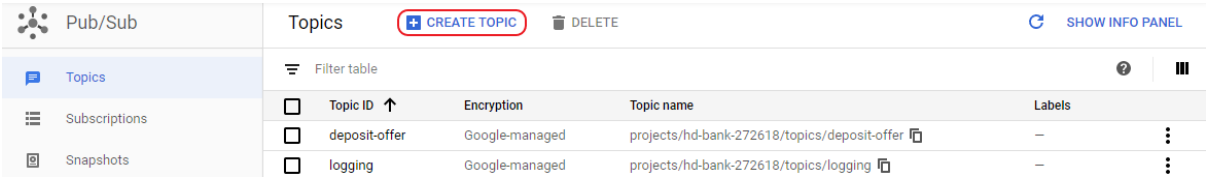
Ostateczna hierarchia wykorzystywanych tabel i perspektyw w projekcie zaprezentowana została na rysunku nr 9.



Rysunek 9: Wykorzystane tabele

Pub/Sub

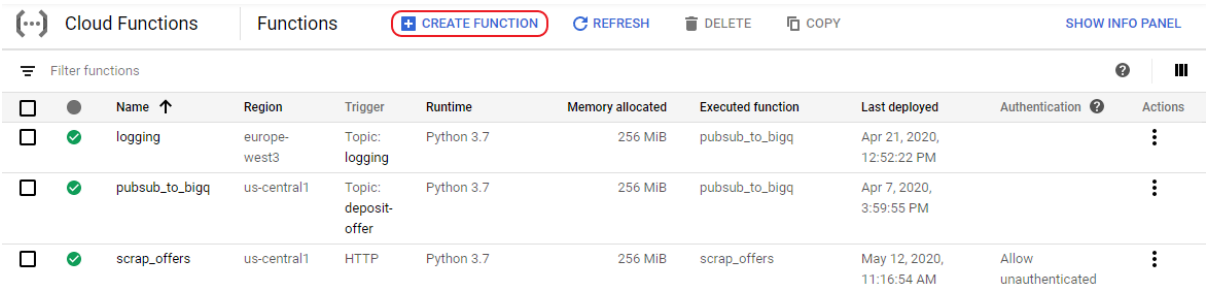
Kolejnym krokiem jest utworzenie tematu w usłudze Pub/Sub. Samo tworzenie tematu ogranicza się do podania nazwy tematu (rys. 10).



Rysunek 10: Tworzenie tematu Pub/Sub

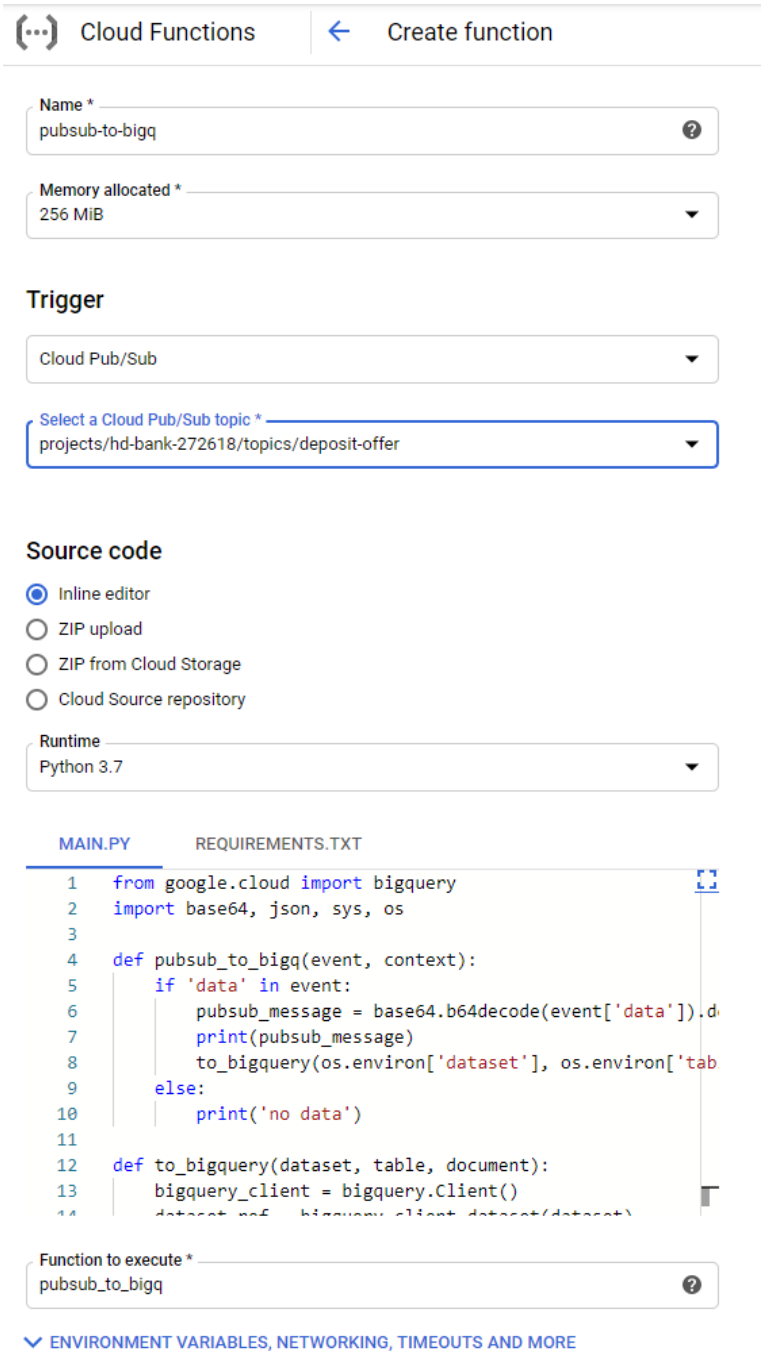
Cloud Functions

W usłudze Cloud Functions tworzymy trzy funkcje (rys. 11). Jedna z nich odpowiedzialna będzie za pobieranie danych ze stron internetowych wybranych banków i następnie za przekazywanie zebranych danych do usługi Pub/Sub. Kolejne dwie funkcje odpowiedzialne będą za przesyłanie danych (dane, logi aplikacji) ze wskazanego tematu usługi Pub/Sub do BigQuery.



Rysunek 11: Tworzenie funkcji lambda

Przykładowa konfiguracja funkcji nasłuchującej na temacie *deposit-offer* usługi Pub/Sub i zapisująca dane w usłudze BigQuery została pokazana na rysunku nr 12.



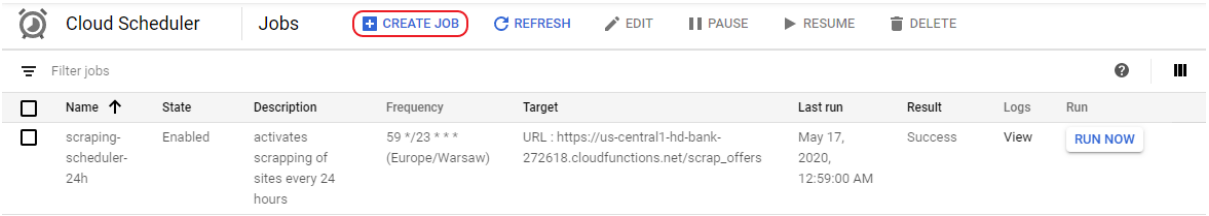
Rysunek 12: Specyfikacja funkcji lambda

Podczas tworzenia bardziej złożonych funkcji (np. składająca się z wielu plików źródłowych) przydatnym może okazać się CLI udostępnione przez Google. Przykładowa komenda tworząca funkcję wywoływaną zapytaniem typu GET przez protokół HTTP w usłudze Google Cloud Functions:

```
gcloud functions deploy scrap_offers --runtime python37 --trigger-http --project hd-bank-272618
```

Cloud Scheduler

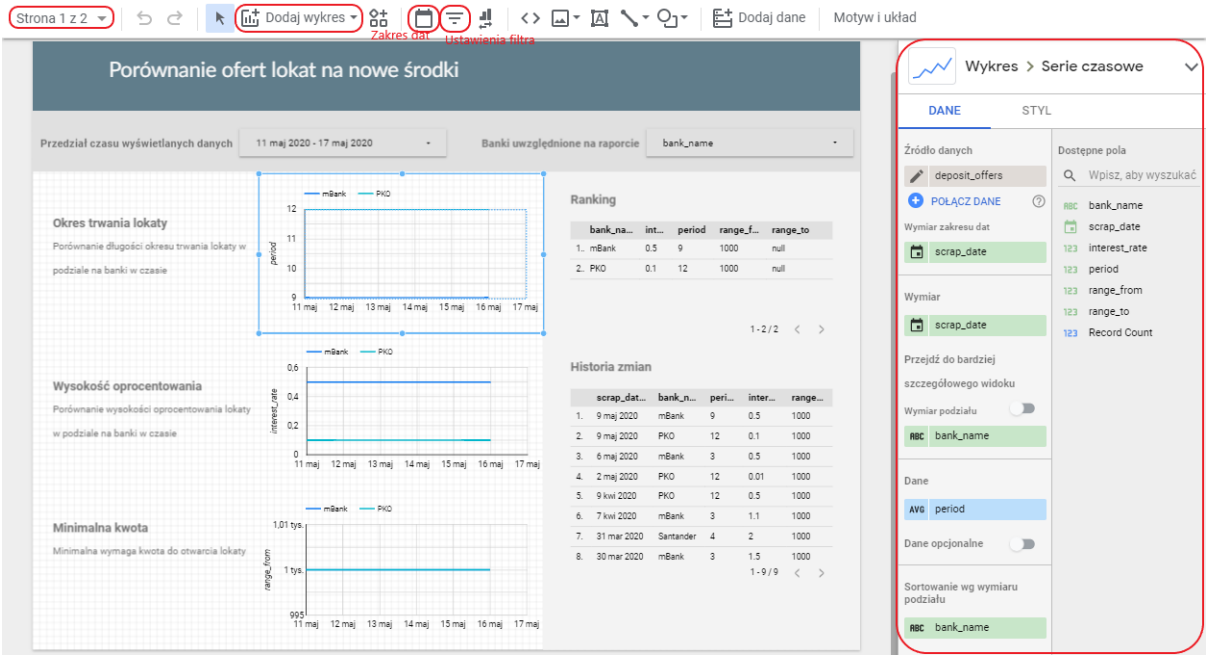
W usłudze Cloud Scheduler tworzymy zadanie (rys. 13), które będzie uruchamiało zdefiniowaną w poprzednim kroku funkcję pobierającą dane o lokatach ze stron internetowych. Podczas tworzenia zadania należy podać co jaki czas funkcja będzie uruchamiana (format unix-cron), strefę czasową oraz sposób wywołania funkcji. W naszym przypadku funkcja będzie uruchamiana za pomocą metody GET protokołu HTTP pod wskazanym w konfiguracji adresem.



Rysunek 13: Tworzenie zadania

Data Studio

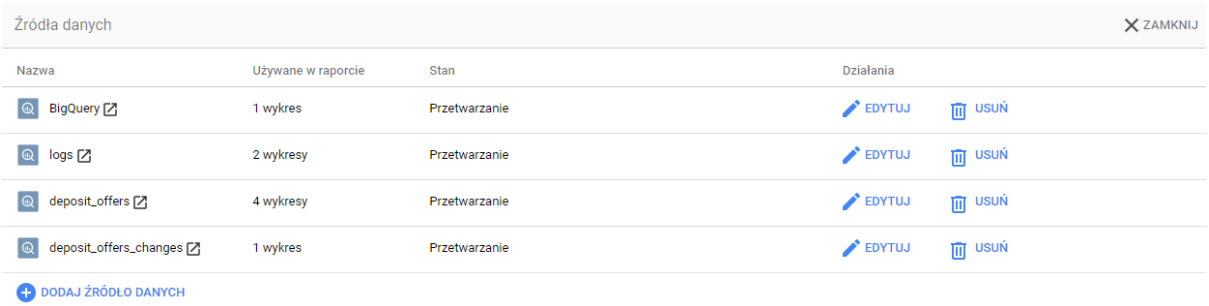
Raport tworzony w usłudze Data Studio może być wielostronicowym dokumentem, który posiada różnego typu wykresy oparte na wielu źródłach danych. Usługa ta pozwala udostępnić użytkownikom końcowym komponenty pozwalające filtrować dane. Modyfikacja wykresów/tabel odbywa się przy wykorzystaniu panelu właściwości wyświetlanego po prawej stronie. W oknie tym można m.in. wskazać źródło danych, dodać własne filtrowanie, ustawić wartości parametrów sparametryzowanego niestandardowego zapytania (rys. 14).



Rysunek 14: Konfiguracja Data Studio

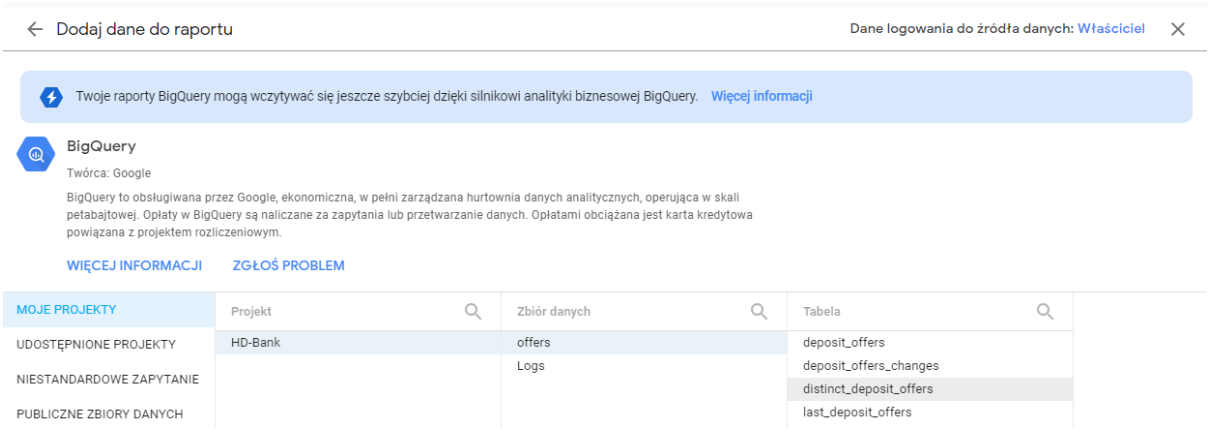
Data Studio pozwala zarządzać dodanymi źródłami danych (rys. 15). Jedną z możliwości jest dodawanie kolejnych źródeł, a drugą ich modyfikacja. Wchodząc w edycję

wybranego źródła możemy zmienić typy kolumn czy też dodawać własne, dynamicznie wyliczane.



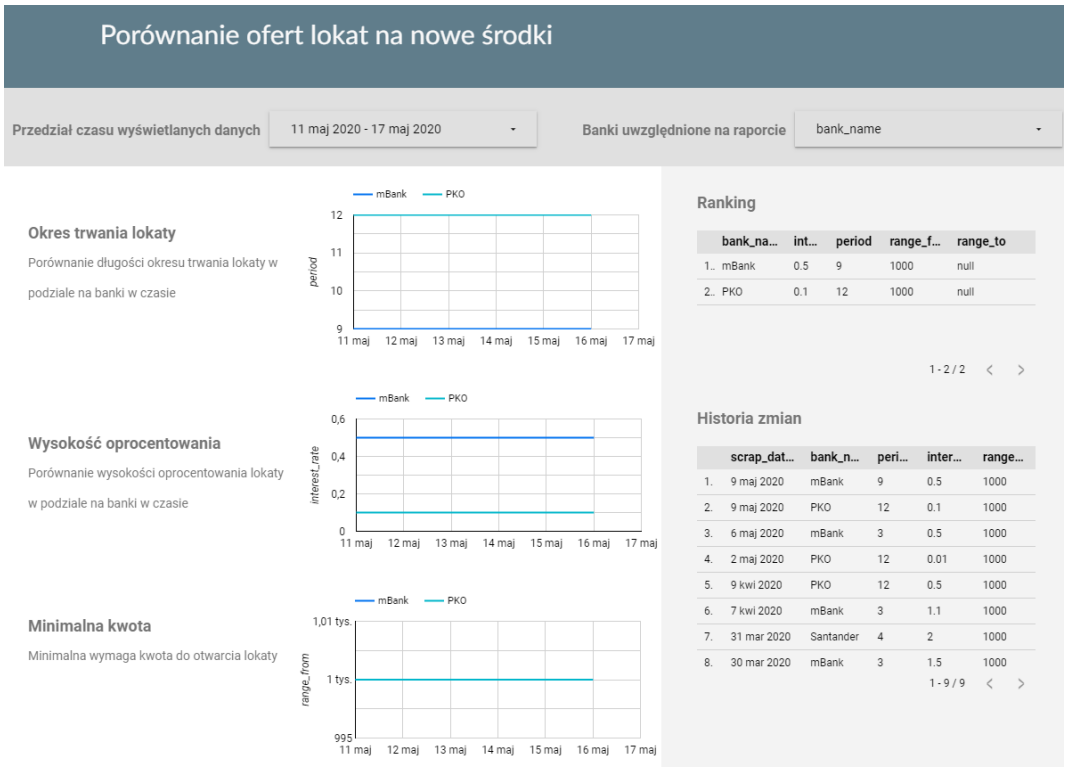
Rysunek 15: Źródła danych

W naszym przypadku wykorzystywane są cztery źródła danych. Dwa z nich wykorzystują wcześniej przygotowane perspektywy w usłudze Google BigQuery (wykresy ofert lokat oraz historia). Kolejnym źródłem danych jest sparametryzowane niestandardowe zapytanie oparte na perspektywie (ranking). Ostatnie źródło danych opiera się na tabeli i ma dodane dynamicznie wyliczane kolumny (logi aplikacji). Widok dodawania źródła danych został zaprezentowany na rysunku nr 16.

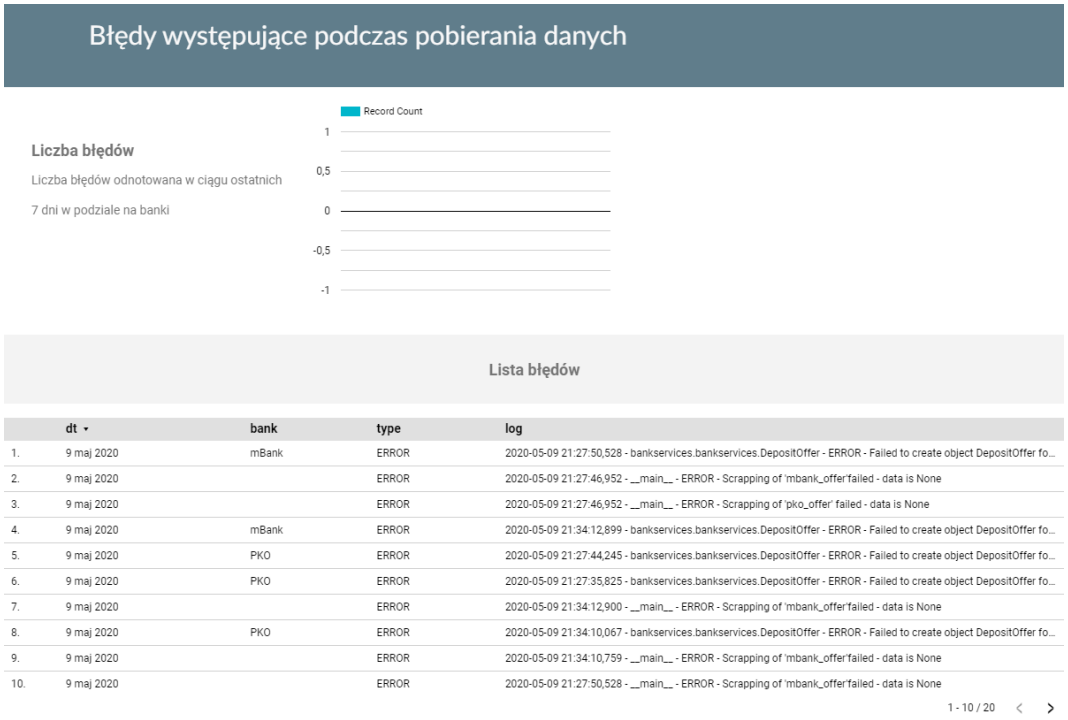


Rysunek 16: Dodawanie źródła danych

Efekt końcowy



Rysunek 17: Porównanie ofert

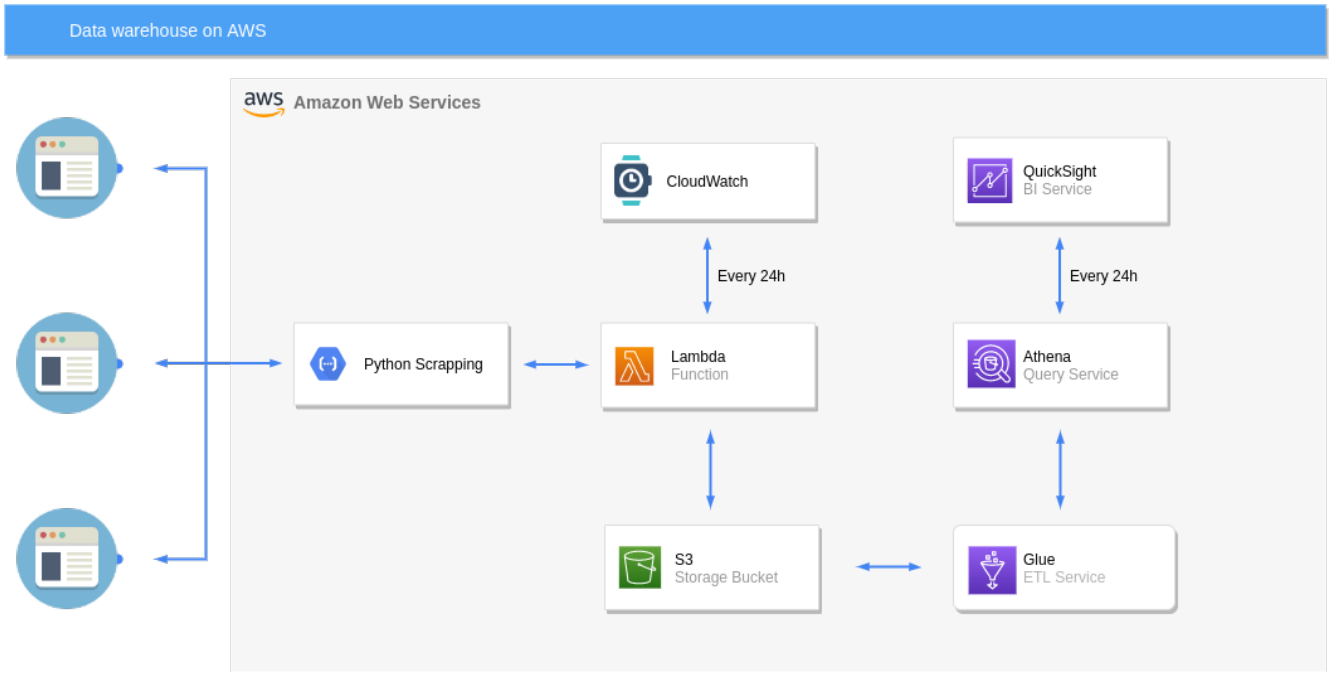


Rysunek 18: Raport o błędach

4 Hurtownia danych na AWS

Amazon Web Services to największa oraz najpopularniejsza platforma usług w chmurze, która oferuje zarówno moc obliczeniową jak i hosting baz danych. Usługi oferowane są na żądanie i rozliczane w modelu Pay-As-You-Go (korzystaj tyle, ile przedpłaciłeś).

4.1 Architektura systemu



Rysunek 19: Architektura systemu zbudowanego na AWS

Cała architektura jest oparta na Lambda Function. Jest to funkcja, która uruchamia skrypt pythonowy opisany w sekcji 2, który zapisuje dane w bazie danych (S3). Proces scrapowania jest uruchamiany o północy za pomocą CloudWatcha.

4.2 Opisy poszczególnych komponentów systemu

4.2.1 Cloud Watch

Cloud Watch jest serwisem stworzonym do monitorowania innych zasobów. Pozwala między innymi na sprawdzanie czy inny serwis odpowiada. My używamy go do wywoływania naszej funkcji lambda o określonej porze każdego dnia. Precyzyjne ustawienie czasu, bądź interwałów czasu może odbyć się poleceniem cron, bądź specjalną składnią zaproponowaną przez Amazon.

4.2.2 Lambda Function

Funkcja lambda jest sercem naszej architektury. Jest także odpowiednikiem GCF dla GCP. Pozwala ona na bardzo elastyczne wywołania kodu bez konieczności przejmowania się architekturą bądź administracją maszyn. Integracja z innymi serwisami AWSa jest prosta. W tej architekturze zapewnione jest również skalowanie, a my płacimy tylko i

wyłącznie za czas przetwarzania (serverless).
Funkcja lambda może składać się z warstw. Oddzielają one kod od jego zależności, a warstwy mogą być współdzielone między różnymi funkcjami. My stworzyliśmy dwie warstwy - jedną dla zależności z przetwarzaniem języka, a drugą dla pozostałych.
Funkcja Lambda uruchamia wskazaną przez nas funkcję ze skryptu napisanego w pythonie. Do wyciągania informacji ze stron internetowych używamy tych samych funkcji co w przypadku GCP, natomiast obsługa błędów odbywa się inaczej. Przetwarzamy każdą stronę oddzielnie, jeśli jej pobranie się powiodło zapisujemy o tym informacje w buckecie. W przeciwnym wypadku tworzymy obiekt przechowujący informacje o błędzie.

4.2.3 S3

Bucket jest folderem na pliki umieszczonym w chmurze, w którym można umieszczać dane dowolnych typów. S3 jest serwisem do zarządzania bucketami jak również uprawnieniami do nich.My przechowujemy informacje z uruchomień tzn. logi z oprogramowania, błędy jakie zostały zgłoszone przez skrypt oraz informacje o depozytach. S3 używamy również do przechowywania kodu źródłowego naszych zależności jak i całej funkcji lambda. S3 jest odpowiednikiem Cloud Storage w GCP..

4.2.4 Glue

Glue jest serwisem odpowiedzialnym za ETL. Możemy w nim definiować, modyfikować nasze dane oraz je przetwarzać. My używamy go, aby wprost określić jaką strukturę przyjmuje tabela, albo bardziej precyzyjnie jak tabela musi wyglądać aby mogła przyjmować przechowywane przez nas dane. W przypadku GCP dzieję się to w czasie tworzenia tabeli w BigQuery. Tworzymy trzy schematy - ofert, błędów oraz logów. AWS Glue pozwala na automatyczne scrapowanie danych w naszym buckecie i inteligentne tworzenie schematów oraz ich aktualizowanie co ustalony czas. Z naszych doświadczeń wynika, że dużo łatwiej operuje się na plikach CSV niż na JSONach. Serwis Glue dużo lepiej radził sobie z rozpoznawaniem ich struktury.

4.2.5 Athena

Athena jest serwisem pozwalającym na wyciąganie danych z bucketów S3 za pomocą języka poleceń SQL. Athena używa struktur zdefiniowanych w AWS Glue i transformuje dane do postaci tabeli. Możemy tutaj definiować takie same perspektywy jak we wspomnianym 3.2.6 - kod jest taki sam.

4.2.6 QuickSight

QuickSight jest usługą dostarczającą tworzenie interaktywnych dashboardów dla naszych danych. Tworzymy w nim Datasety, które mogą pochodzić z wielu źródeł. W naszym przypadku są to trzy datasety stworzone za pomocą zapytań w Athena. Działa to w ten sposób, że QuickSight wywołuje query w Athenie, która wyjmuje wskazane dane z S3 i przedstawia je w postaci ustalonej w AWS Glue. Następnie tak zgromadzone dane są zapisywane w SPICE. SPICE (QuickSight's Super-fast, Parallel, In-memory Calculation Engine) jest specjalnym silnikiem obliczeń, który również przechowuje nasze dane. Dzięki wykorzystaniu SPICE wszelkie operacje dokonywane na interaktywnych wykresach wykonują się bez opóźnień, nawet w przypadku dużych wolumenów danych.

4.3 Wady rozwiązania

W rozwiązaniu na GCP implementacja używa bibliotekę NLTK do przetwarzania języka, która ułatwia tokenizację. Niestety nie udało się jej użyć. Różni się ona od innych bibliotek tym, że wymaga pobranych wcześniej korpusów języka. Na lokalnym komputerze korpus zostaje pobrany i zapisany w katalogu użytkownika. My pobraliśmy ten sam korpus i dodaliśmy go wraz z innym bibliotekami do warstwy zależności lambda. Następnie w kodzie podaliśmy poprawną (według dokumentacji) ścieżkę do miejsca, do którego te dane są przenoszone podczas uruchomienia funkcji lambda. Mimo to program dalej nie znajduje wskazanych plików. Jednym z możliwych obejść problemu wydaje się składowanie tych korpusów w buckecie. Natomiast nie jest to rozwiązanie zgodne z dobrymi praktykami. Nie chcemy trzymać w naszej bazie danych korpusów do jednej wybranej biblioteki, skoro mamy do tego cały oddzielny serwis. Drugą kwestią, która mogłaby zostać poprawiona jest połączenie logów i błędów w jedną strukturę tak jak to jest na GCP. Rozbieżność wynika z przyrostowej implementacji.

4.4 Koszty AWSa

Amazon udostępnia miesięczne limity, w których przetwarzanie jest całkowicie darmowe. Dla serwisu S3 limitowana jest liczba zapytań każdego typu do 2000 oraz nakładane są limity na poszczególne zapytania np. GET - 20000. Dla funkcji lambda mamy miesięcznie do wykorzystania 400 000 sekund przetwarzania oraz milion zapytań. Względem naszych potrzeb jest to bardzo dużo. Przez tydzień użytkowania najwięcej wykorzystaliśmy zapytań w serwisie S3 - 3.3%. Według predykcji dokonanych przez serwis Amazonu miesięcznie powinniśmy wykorzystywać około 10% darmowego transferu.
W trakcie pracy nad projektem nie obyło się bez pomyłek. Mianowicie zapisywaliśmy każdą linie logów oddzielnie do bucketa zamiast całą tablicę. W ten sposób szybko wykorzystaliśmy darmowe limity requestów. Przez darmowe progi nie jesteśmy w stanie realnie obliczyć ile kosztuje użytkowanie tej architektury - natomiast wydaje nam się, że podobnie jak w przypadku GCP są to niskie ceny.
Problemem może okazać się serwis do tworzenia wizualizacji danych - QuickSight. Niestety poza wcześniejszymi limitami między innymi zapytań, jest on dodatkowo płatny 18$ za każdego twórce - jedynie pierwsze dwa miesiące licencji są udostępnione za darmo.

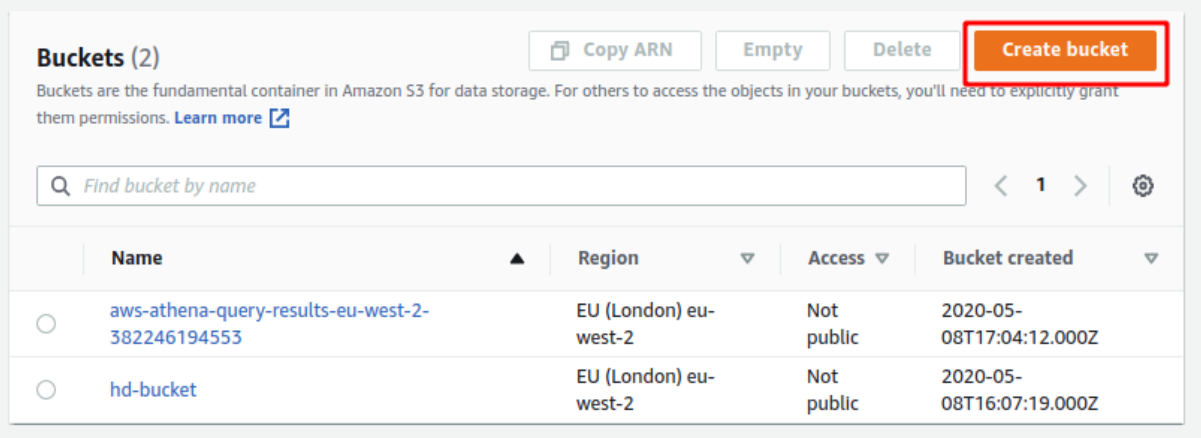
4.5 Lokalizacja danych

Każdy komponent, który tworzymy w AWSie ma swoją lokalizację - nawet jeśli nie przechowuje danych. Istotne jest to, że wszystkie komponenty muszą być w tej samej lokalizacji aby widziały się nawzajem. Dane są składowane w buckecie oraz w pamięci SPICE, których lokalizację znamy. Nic nie stoi na przeszkodzie aby dane do wykresów były każdorazowo dynamicznie wyciągane za pomocą query z S3. Niestety, zwiększyłoby to liczbę zapytań a tym samym koszty. Pamięć w SPICE mimo, że jest ograniczona to używamy zaledwie kilku megabajtów z dostępnego gigabajta.

4.6 Wskazówki

Podążaliśmy za przystępnie napisanym poradnikiem Make Data Acquisition Easy with AWS & Lambda (Python) in 12 Steps[11]. Przejdziemy przez wszystkie komponenty i w skrócie opiszemy co należy w nich skonfigurować.

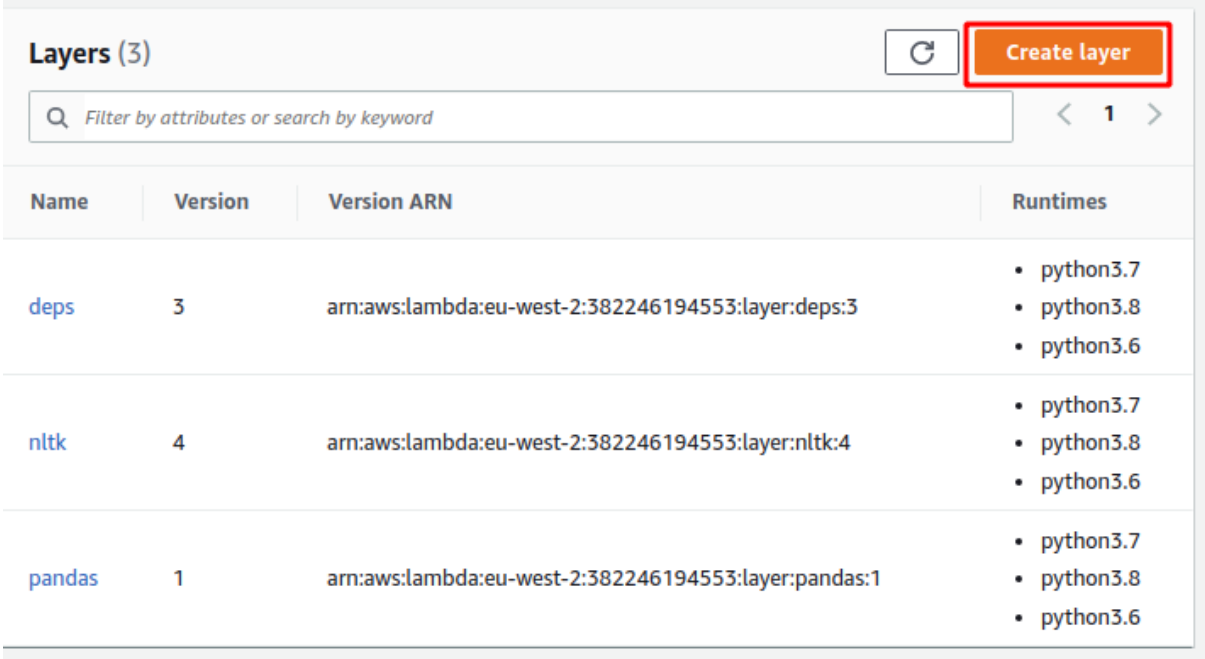
S3



Rysunek 20: Widok serwisu S3

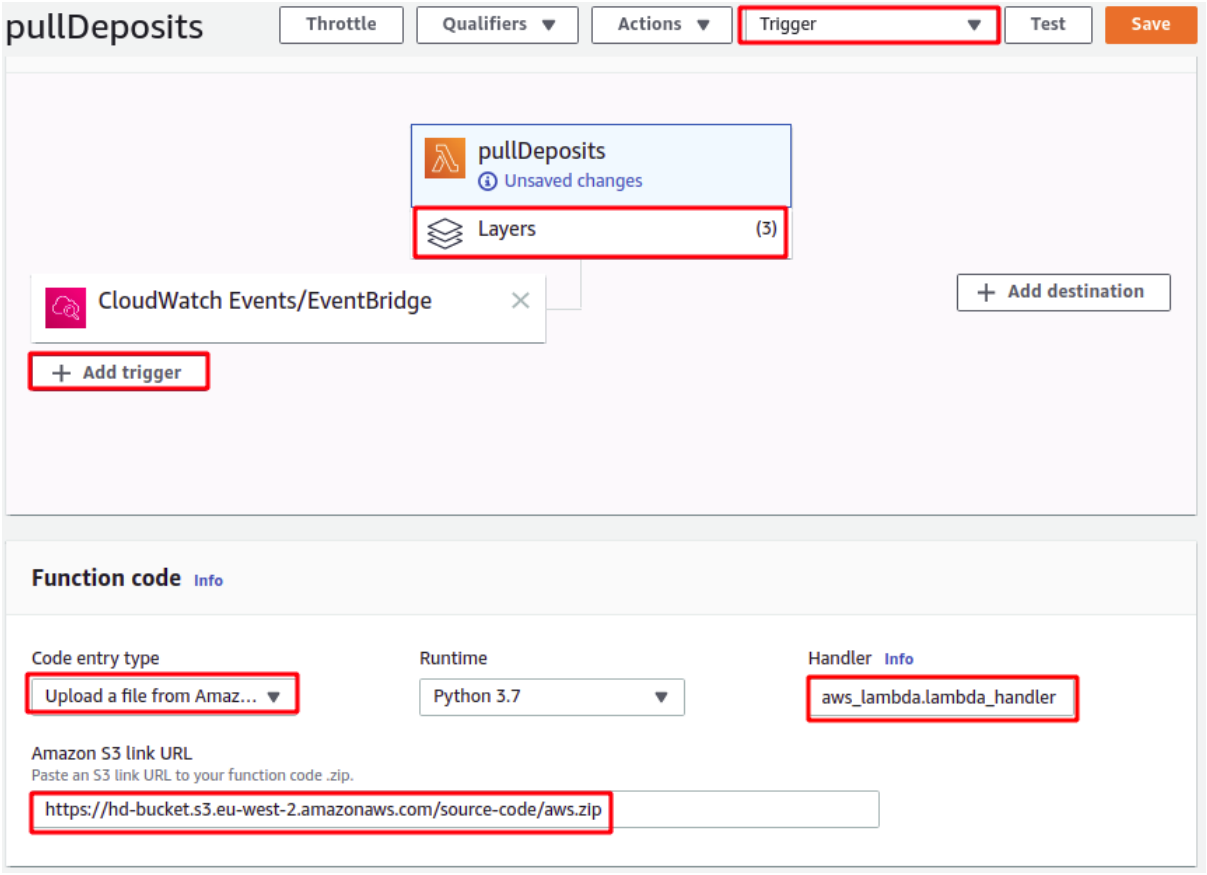
W serwisie S3 przedstawionym na rysunku 20 tworzymy bucket. Ważna jest jego lokalizacja - my wybraliśmy eu-west-2 (Londyn). Do bucketów przenosimy paczki .zip z zależnościami oraz kodem źródłowym. Możemy je wygenerować za pomocą poleceń podanych w README.md w kodzie źródłowym.

Lambda



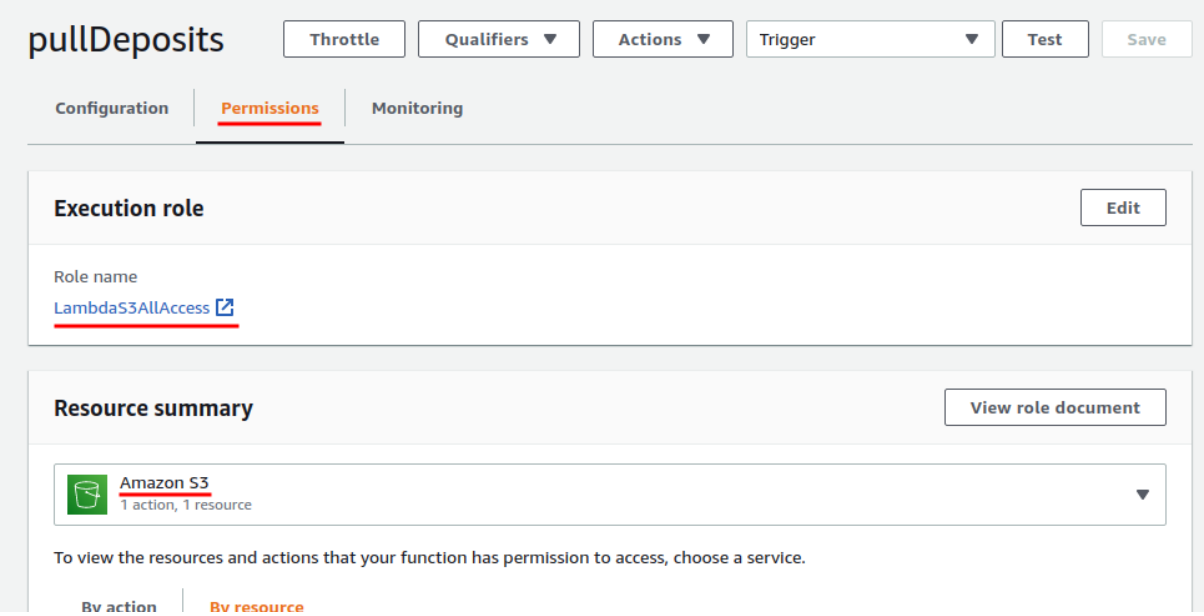
Rysunek 21: Dodane zależności w serwisie AWS Lambda

Następnie tworzymy warstwy zależności (rysunek 21), podając ścieżki do wgranych do S3 zipów.



Rysunek 22: Widok serwisu AWS Lambda

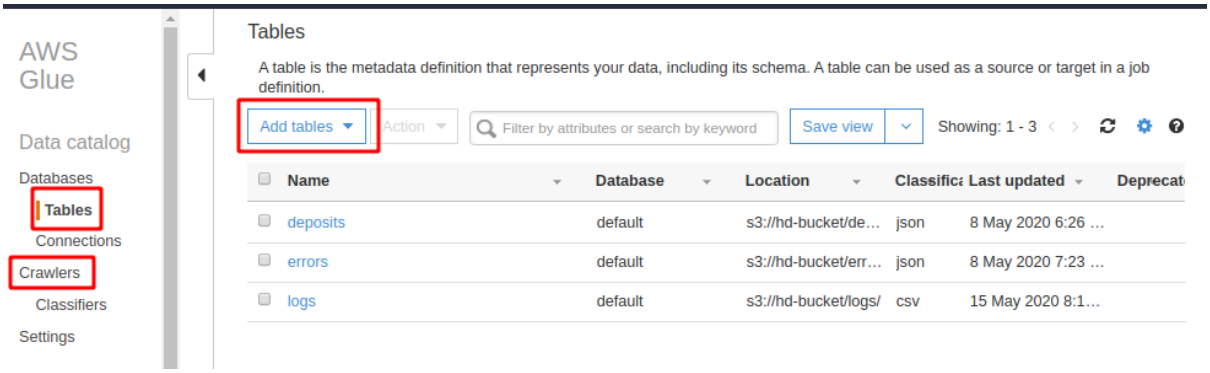
Przechodzimy do widoku funkcji lambda, tutaj mamy kilka rzeczy do ustawienia. Zaczynamy od stworzenia testu (na rysnku 22 pole Trigger) - nie musimy ustawiać żadnych parametrów. Wystarczy nam wywołanie naszej funkcji. Ustawiamy również automatyczne wywołanie funkcji (pole Add trigger). Konfigurjemy je przykładowo poleceniem CRON(0 0 * * ? *) - uruchom codziennie o północy. W pierwszym panelu przechodzimy do Layers i dodajemy stworzone wcześniej warstwy zależności. W polu Function code wybieramy wgranie kodu przez zip w buckecie S3 (możemy również pisać kod w edytorze online, bądź lokalnie i używać AWS CLI do synchronizacji danych oraz deployowania oprogramowania). W polu Handler podajemy nazwa_pliku.nazwa_funkcji. W Basic settings należy zwiększyć timeout do np. 20 sekund, nasz kod wykonuje się dłużej, niż domyślne 3 sekundy.



Rysunek 23: Widok uprawnień serwisu AWS Lambda

Funkcja lambda (rysunek 23) musi mieć uprawnienia do bucketa S3, konfigurujemy je przez pole permission bądź serwis AWS IAM. Na końcu zapisujemy i możemy przetestować naszą funkcję. Dane powinny pojawić się w S3.

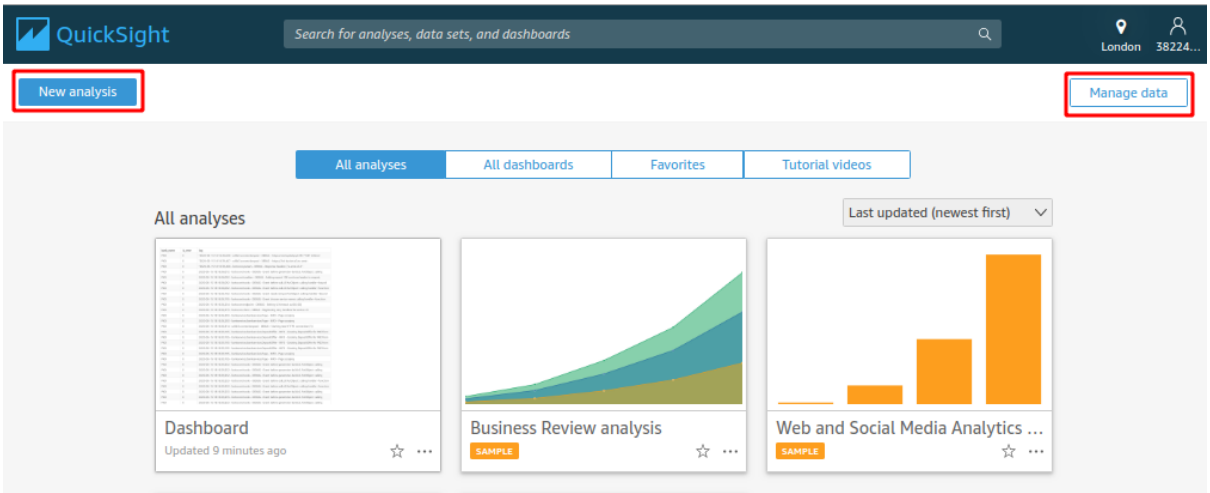
Glue



Rysunek 24: Tabele dodane w serwisie AWS Glue

W serwisie AWS Glue przedstawionym na rysunku 24tworzymy schematy naszych struktur, bądź używamy crawlera, który automatycznie się ich domyśli. W naszym przypadku o ile z CSV sobie radził świetnie to z jsonami już nie dał rady.

QuickSight



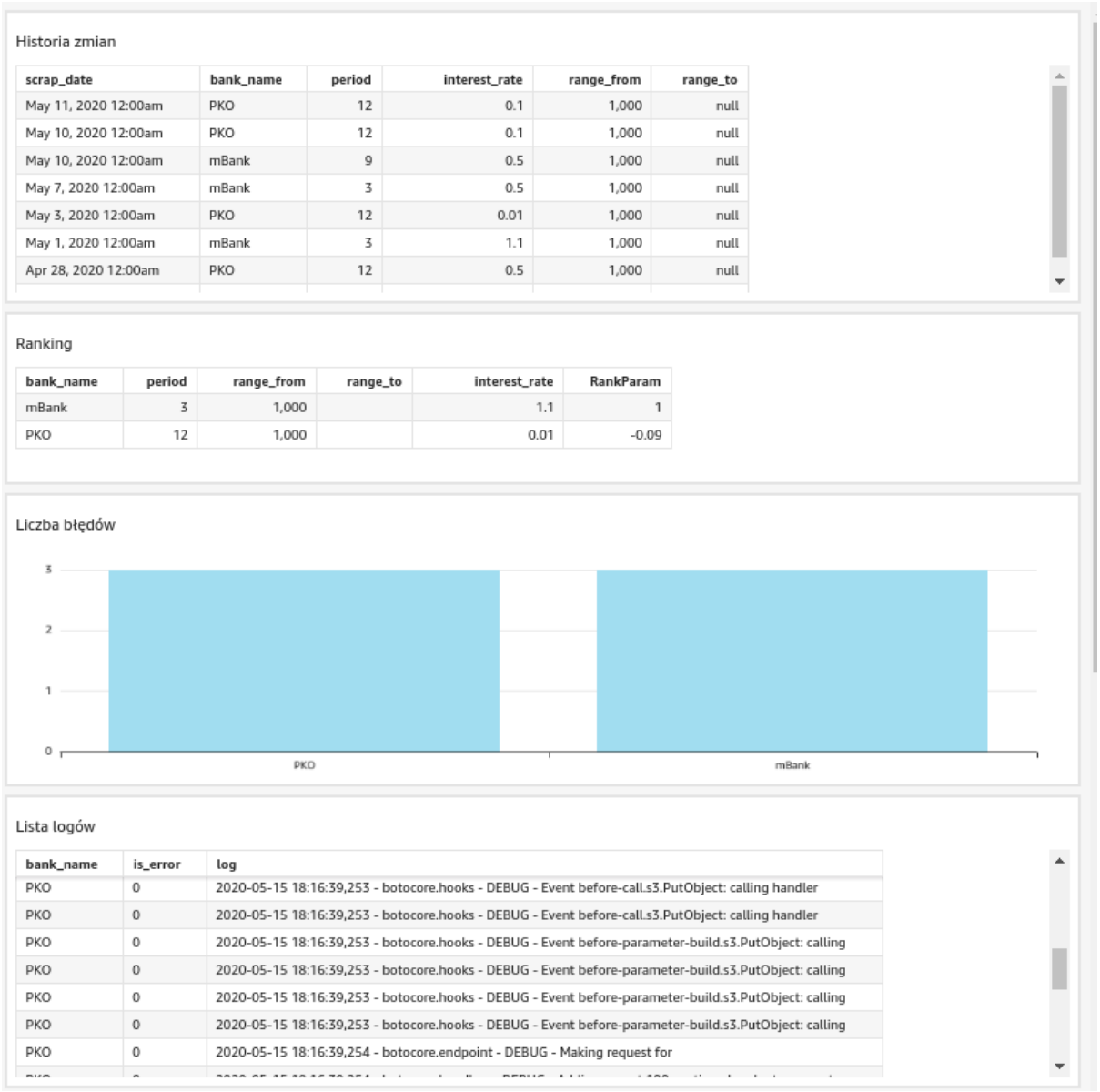
Rysunek 25: Wygląd serwisu AWS QuickSight

W serwisie QuickSight (rysunek 25) pod Manage data tworzymy dataset gdzie jako źródło podajemy Athena i wpisujemy tam zapytanie SQL, bądź wcześniej przygotowane query. Samo zapytanie SQL możemy przetestować w serwisie Athena, a potem je zapisać. Jest to opcjonalne. Podczas dodawania źródeł danych możemy dokładnie określić format dat, stworzyć pola wyliczane itp. Następnie przechodzimy do New analysis, gdzie tworzymy wykresy z naszych datasetów.

Efekt końcowy

Wszystkie wykresy (rysunek 26 oraz 27) przedstawiają te same informacje co wykresy z GCP.

Rysunek 26: Wykres utworzony na platformie AWS



Rysunek 27: Wykres utworzony na platformie AWS

5 Porównanie rozwiązań

Poniższy punkt służy opisaniu naszych doświadczeń, które nabyliśmy tworząc opisane wyżej rozwiązania. Zaczniemy od porównania od strony technicznej, przejdziemy przez materiały, które są dostępne i dzięki którym możemy nauczyć się tworzyć rozwiązania na chmurze GCP i AWS, a na końcu podzielimy się własną opinią w zakresie doboru technologii do tego projektu.

5.1 Porównanie od strony technicznej

Porównanie obu platform w oparciu o wykorzystywane przez nas komponenty można zobaczyć w tabeli 1. We wspomnianej tabeli, porównujemy obie platformy (GCP i AWS) pod względem najważniejszych funkcjonalności które były niezbędne do realizacji tego projektu. Są to: szeregowanie zadań (umożliwiające cykliczne uruchamianie scrapowania ofert co 24h), uruchamianie kodu, składowanie danych oraz wizualizacja. W odpowiednich komórkach tabeli wspomniane są komponenty, które umożliwiają realizację odpowiedniej funkcjonalności na danej platformie, możliwości jakie owe narzędzia oferują, ich ewentualne wady i zalety oraz koszty z jakimi trzeba się liczyć przy ich wykorzystaniu.

Tabela 1: Porównanie platform GCP i AWS

Funkcjonalność	**GCP**	**AWS**
Szeregowanie zadań	Cloud scheduler - Google oferuje do 3 darmowych zadań, które można wywołać o planowanym czasie. Każde kolejne zadanie kosztuje 0.10$ miesięcznie.	CloudWatch - platforma Amazonu oferuje dowolną ilość zadań o ile liczba requestów przez nie wykonanych nie przekroczy miliona miesięcznie. My wykorzystujemy 1 każdego dnia. Czas wykonania zadania ustawiamy za pomocą wyrażeń CRON bądź składni zaproponowanej przez Amazon.
Uruchamianie kodu	Cloud functions - Zaimplementowaliśmy serwer flask, który wystawia endpoint wywoływany przez Cloud Schedulera. Dane trafiają do tematu w PubSub, który przenosi je do BigQuery, gdzie magazynowe są w postaci tabeli. Kod programu wgrywany za pomocą CLI. Darmowe 2 miliony wywołań za pomocą protokołu HTTP, darmowe 5GB transferu danych z zewnętrznej sieci, darmowe 200 000 GHz-sekund i 400 000 GB-sekund. Dodatkowo do przesyłania danych pomiędzy wywoływanymi funkcjami wykorzystywana jest usługa Google Pub/Sub, która udostępnia 10GB darmowego transferu. Koszt kolejnych danych wynosi 40$ za TB.	CloudWatch uruchamia funkcję lambda (program przygotowany w języku Python), a następnie z poziomu kodu obsługiwany jest zapis pliku json/csv do S3. Kod funkcji wgrywany jest za pomocą pliku zip. Darmowo dostępne 400 000 sekund przetwarzania oraz 2000 odwołań do bucketa.
Składowanie danych	BigQuery - Skalowalna, bezserwerowa hurtownia danych która umożliwia szybką analizą dużych wolumenów danych. W celu wykonywania zapytań korzystamy z ANSI SQL. Jeżeli chodzi o wydajność, to rozwiązanie z GCP jest znacznie lepsze niż Athena na AWS, co potwierdzają liczne benchmarki. Jeżeli chodzi z kolei o koszt, to w przypadku obu rozwiązań płacimy za ilość danych przetworzonych podczas zapytania. W przypadku BigQuery pierwszy TB mamy za darmo. Później zarówno BigQuery jak i Athena kosztują 5$ za każdy przetworzony TB. Początkowe opłaty to 0.02 USD/GB miesięcznie, cena ta ulega obniżeniu do 0.01 przy długotrwałym korzystaniu.	Athena + Glue + S3. Koszty ponosimy za przechowywanie wyników query z Atheny w Buckecie S3 oraz 5$ za każdy przetworzony TB. Pamięć w buckecie S3 kosztuje 0.024$ za każdy GB misięcznie. Cena maleje 0.01 po przekroczeniu 50 oraz 450 TB. Pierwsze 5GB jest darmowe.
Wizualizacja	Główną zaletą Google Data Studio jest zdecydowanie możliwość łatwej integracji wielu źródeł danych. Mamy do dyspozycji wbudowane konektory nie tylko do BigQuery, ale także do GoogleSheets, Google Analytics, Google Adwords, Facebook, Twitter i wiele innych. Nawet osoby bez doświadczenia w programowaniu mogą wyciągnać wiele wartości z tego komponentu. Jeżeli chodzi o cenę, to Google Data Studio jest za darmo.	Integracja Quicksight z Amazonem okazała się bardziej kłopotliwa niż myśleliśmy na początku. Musieliśmy dodać dodatkowy krok transformujący dane z json/csv do tabeli, który w GCP wykonywał się automatycznie. Pod względem funkcjonalności komponent ten nie odstaje zbytnio od Google Data Studio. Wydaje się jednak, że GDS jest bardziej przyjazny dla osób bez doświadczenia w programowaniu oraz pozwala w łatwiejszy sposób tworzyć przejrzystsze dashboardy. Niestety QuickSight jest usługą dużo droższą 18$ za każdego użytkownika lub 5$ za każdego obserwatora miesięcznie. Dodatkowo pamięć wykorzystywana przy interaktywnych wizualizacjach jest ograniczona do 10GB dla każdego użytkownika. Pierwszy gigabajt jest darmowy, każdy kolejny kosztuje 0.38$. Możemy nie korzystać z tej pamięci, a używać w to miejsce query z serwisu Athena.

5.2 Łatwość używania, dokumentacja

Już na samym początku projektu podczas tworzenia swoistego MVP na Google Cloud Platform przekonaliśmy się, jak kosztowne może być eksperymentowanie na tejże platformie. Po pierwszej iteracji proste pobieranie danych ze stron banku i składowanie ich w BigQuery generowało koszty na poziomie 10$ dziennie. Platforma ta zatem w naszym odczuciu dużo gorzej sprawdzi się, kiedy chcemy prototypować w porównaniu do AWS. AWS udostępnia nam możliwość prototypowania i używania wielu serwisów (w ograniczonym wprawdzie stopniu zasobowym, ale jednak) za darmo, co jest jego dużą zaletą. Jeżeli chodzi o liczbę materiałów, które znajdują się w sieci niezbędnych do nauki GCP, to jest ich naprawdę sporo. Pomijając samą dokumentacje, która jest napisana dobrze i przejrzyście, możemy natknąć się na całe mnóstwo blogów i kodów źródłowych np. na GitHub. W przypadku AWS dokumentacja również jest bardzo dobrze napisana i zawiera wiele przykładów. Odpowiedzi na forum stackoverflow zazwyczaj kierują na odpowiedni artykuł wspomnianej dokumentacji.

5.3 Wnioski

Po zrealizowaniu opisanego powyżej projektu, można z całą stanowczością stwierdzić, że zarówno GCP jak i AWS mają swoje wady i zalety, jednak trudno jest jednoznacznie wskazać, która platforma jest lepsza. W naszych odczuciach obie platformy nie odstają od siebie zbyt mocno pod względem oferowanych funkcjonalności, a realizacja tego projektu pozwoliła nam częściowo odkryć ich tajniki. Mimo, iż pojawiały się drobne komplikacje podczas realizacji poszczególnych etapów, takie jak zbyt duże koszty na GCP o czym można przeczytać w punkcie 3.3 lub problem z dodaniem biblioteki w zależnościach na AWS opisany w punkcie 4.3, to ostatecznie udało się zrealizować projekt zarówno na jednej, jak i na drugiej platformie.
To, którą platformę byśmy wybrali, gdybyśmy musieli zdecydować się na jedną do realizacji projektu, zależy od wielu czynników. Jednym z nich może być lokalizacja w której chcemy - bądź musimy - składować nasze dane (komponenty chmury służące do składowania danych takie, jak S3 lub Cloud Storage mogą bowiem znajdować się fizycznie na terenie różnych krajów). Innym czynnikiem są też kwestie finansowe - wiemy już, że chwila nieuwagi na GCP może narazić nas na niemałe (jak na studenckie możliwości) koszty. Jednak zawsze warto jest się przyjrzeć komponentom poszczególnych platform, które opisaliśmy w odpowiednich paragrafach, zobaczyć ich mocne i słabe strony oraz określić czego bardziej potrzebujemy w naszym projekcie i wówczas zdecydować się na konkretną platformę.
Dodatkowo kluczowym parametrem wyboru wydaje się być fakt czy zamierzamy dane wizualizować. W tej kwestii usługa Googla nie dość, że całkowicie darmowa to pozwala tworzyć naszym zdaniem dużo milsze dla oka wykresy. W naszym przypadku część implementacyjna nie różniła się znacząco miedzy platformami. W obu przypadkach musieliśmy wykonać te same kroki tylko w innych serwisach, bądź w nieco inny sposób. Nakład pracy jednak pozostał ten sam, a wspomniana usługa do wizualizacji wydaje się być jedyną znaczną różnicą.

Bibliografia

[1] Strona banku PKO podlegająca scrappowaniu.
[on-line]
https://www.pkobp.pl/klienci-indywidualni/oszczednosci/lokaty/lokata-terminowa/,
dostęp: kwiecień 2020.

[2] Strona banku mBank podlegająca scrappowaniu.
[on-line]
https://www.mbank.pl/indywidualny/oszczednosci/lokaty/lokata-dla-nowych-srodkow/,
dostęp: kwiecień 2020.

[3] Dokumentacja Google Cloud Scheduler.
[on-line] https://cloud.google.com/scheduler,
dostęp: czerwiec 2020.

[4] Dokumentacja Google Cloud Functions.
[on-line] https://cloud.google.com/functions,
dostęp: czerwiec 2020.

[5] Dokumentacja Google Pub/Sub.
[on-line] https://cloud.google.com/pubsub/docs/,
dostęp: czerwiec 2020.

[6] Dokumentacja Google Cloud Dataflow.
[on-line] https://cloud.google.com/dataflow,
dostęp: czerwiec 2020.

[7] Dokumentacja Google BigQuery.
[on-line] https://cloud.google.com/bigquery,
dostęp: czerwiec 2020.

[8] Dokumentacja Google Data Studio.
[on-line] https://datastudio.google.com,
dostęp: czerwiec 2020.

[9] Running a scraping platform at Google Cloud for as little as US$ 0.05/month.
[on-line] https://levelup.gitconnected.com/
running-a-scraping-platform-at-google-cloud-for-as-little-as-us-0-05-month-6d9658982f04,
dostęp: czerwiec 2020.

[10] Web scraping with Google Cloud Functions, Pub/Sub, DataFlow & BigQuery.
[on-line] https://www.roelpeters.be/
web-scraping-with-google-cloud-functions-pub-sub-dataflow-bigquery/?fbclid=
IwAR2KygJ0EIDVdgTBr7O04Den-lkSU2˙0s-3Z-CjTco14yzSP6plwvK-nnSY,
dostęp: czerwiec 2020.

[11] Make Data Acquisition Easy with AWS & Lambda (Python) in 12 Steps.
[on-line] https://towardsdatascience.com/
make-data-acquisition-easy-with-aws-lambda-python-in-12-steps-33fe201d1bb4,
dostęp: czerwiec 2020.

[12] Dokumentacja Amazon CloudWatch.
[on-line] https://aws.amazon.com/cloudwatch/,
dostęp: czerwiec 2020.

[13] Dokumentacja AWS Lambda.
[on-line] https://aws.amazon.com/lambda/,
dostęp: czerwiec 2020.

[14] Dokumentacja Amazon S3.
[on-line] https://aws.amazon.com/s3/,
dostęp: czerwiec 2020.

[15] Dokumentacja AWS Glue.
[on-line] https://aws.amazon.com/glue/,
dostęp: czerwiec 2020.

[16] Dokumentacja Amazon Athena.
[on-line] https://aws.amazon.com/athena/,
dostęp: czerwiec 2020.

[17] Dokumentacja Amazon QuickSight.
[on-line] https://aws.amazon.com/quicksight/,
dostęp: czerwiec 2020.