Zaawansowane Systemy Baz Danych

Wykład 0

Model relacyjny – Wprowadzenie

Definicja: Model relacyjny to sposób organizacji danych w bazach danych, w którym dane są przechowywane w tabelach (relacjach). Każda tabela składa się z wierszy (rekordów) i kolumn (atrybutów).

Podstawowe pojęcia:

- **Relacja**: Tabela w bazie danych.
- **Atrybut**: Kolumna w tabeli, reprezentująca cechę danych.
- **Krotka**: Wiersz w tabeli, reprezentujący pojedynczy rekord.
- **Klucz główny**: Unikalny identyfikator dla każdego rekordu w tabeli.
- **Klucz obcy**: Atrybut w jednej tabeli, który odwołuje się do klucza głównego w innej tabeli, tworząc relację między tabelami.

Zalety modelu relacyjnego:

- **Elastyczność**: Łatwość modyfikacji struktury bazy danych bez wpływu na aplikacje korzystające z danych.
- **Spójność danych**: Mechanizmy zapewniające integralność danych, takie jak reguły integralności referencyjnej.
- **Niezależność danych**: Oddzielenie logiki aplikacji od fizycznego przechowywania danych.

Operacje na danych:

- **Selekcja**: Wybieranie określonych wierszy z tabeli na podstawie warunków.
- **Projekcja**: Wybieranie określonych kolumn z tabeli.
- **Złączenie**: Łączenie danych z dwóch lub więcej tabel na podstawie relacji między nimi.
- **Unia**: Łączenie wyników dwóch zapytań w jeden zestaw wyników.
- **Różnica**: Znajdowanie różnic między dwoma zestawami danych.

Normalizacja:

Proces organizacji struktury bazy danych w celu minimalizacji redundancji danych i zapewnienia ich integralności.

Język SQL:

Standardowy język używany do definiowania, manipulowania i kontrolowania danych w relacyjnych bazach danych.

Model relacyjny – Wikipedia

Model relacyjny – model organizacji danych bazujący na matematycznej teorii mnogości, w szczególności na pojęciu relacji. Na modelu relacyjnym oparta jest relacyjna baza danych (ang. Relational Database) – baza danych, w której dane są przedstawione w postaci relacyjnej.

- **Historię**: Wprowadzenie modelu relacyjnego przez Edgara F. Codda w 1970 roku.
- **Podstawowe pojęcia**: Relacje, atrybuty, krotki, klucze główne i obce.
- **Operacje relacyjne**: Selekcja, projekcja, złączenie, unia, różnica.
- **Normalizację**: Proces eliminacji redundancji danych poprzez podział tabel na mniejsze, bardziej zrozumiałe jednostki.
- **Zalety**: Spójność danych, elastyczność, niezależność danych.

Czym jest relacyjna baza danych – Oracle

- **Definicję**: Relacyjna baza danych to typ bazy danych, który przechowuje i zapewnia dostęp do powiązanych ze sobą elementów danych.
- **Przykład**: Dwie tabele: jedna z informacjami o klientach (nazwa, adres, dane kontaktowe), druga z zamówieniami (identyfikator klienta, produkt, ilość). Relacja między nimi jest tworzona za pomocą identyfikatora klienta.
- **Organizację**: Oddzielenie logicznej struktury danych (tabele, widoki, indeksy) od fizycznych struktur pamięci, co umożliwia zarządzanie przechowywaniem danych bez wpływu na dostęp do nich.
- **Model relacyjny**: Dane są przedstawiane w tabelach, co eliminuje potrzebę tworzenia skomplikowanych struktur danych dla każdej aplikacji.
- **Korzyści**: Spójność danych, łatwość zarządzania, elastyczność, skalowalność.

Relational model – Wikipedia (angielska)

- **Historię**: Model relacyjny został zaproponowany przez Edgara F. Codda w 1969 roku jako sposób zarządzania danymi przy użyciu struktury i języka zgodnego z logiką pierwszego rzędu.
- **Koncepcje**:
  - **Podstawowe pojęcia**: Relacje (tabele), atrybuty (kolumny), krotki (wiersze).
  - **Ograniczenia**: Klucze (główne i obce), ograniczenia integralności.
  - **Operacje relacyjne**: Selekcja, projekcja, złączenie, unia, różnica.
  - **Normalizacja**: Proces organizacji danych w celu minimal

Wykład 1 – Pandas: instalacja i pierwsze kroki

1. Instalacja i uruchomienie Pandas

- Pandas można zainstalować za pomocą:
  - `!pip install pandas` – instalacja za pomocą pip.
  - `!conda install pandas` – instalacja przy użyciu Conda.
- Sprawdzanie wersji:

```
import pandas as pd
print(pd.__version__)  # Wyświetla bieżącą wersję Pandas
```

2. Tworzenie obiektów DataFrame

- DataFrame można tworzyć m.in. za pomocą słowników:

```
x = [1, 2, 3, 4, 5, 6, 7]
y = [12., 22.1, 15.5, 35, 1, 1, 1]
df1 = pd.DataFrame({"kol1": x, "kol2": y})
```

- Tworzenie DataFrame z losowymi wartościami:

```
import numpy as np
z = np.random.randn(1000, 2)
df2 = pd.DataFrame(data=z, columns=["losowe1", "losowe2"])
```

3. Inspekcja danych

- **Podgląd danych**:

```
df2.head()       # Domyślnie wyświetla pierwsze 5 wierszy
df2.head(3)      # Wyświetla pierwsze 3 wiersze
df2.tail(4)      # Wyświetla ostatnie 4 wiersze
df2.sample()     # Losowy wiersz
df2.sample(5)    # 5 losowych wierszy
```

- **Wymiary tabeli**:

```
df2.shape  # Zwraca liczbę wierszy i kolumn
```

4. Modyfikacje danych

- **Zmiana nazw kolumn**:

```
df2.rename(columns={"losowe1": "kolumna1", "losowe2": "kolumna2"},
inplace=True)
```

- **Dodawanie kolumn**:

```
df2["nowa_kolumna"] = "tekst"
```

- **Usuwanie kolumn**:

```
df2.drop(columns=["kolumna1"], inplace=True)
```

5. Selekcja danych

- **Selekcja za pomocą indeksów**:

```
df2.iloc[3:6, :1]   # Wiersze 3-5, kolumna pierwsza
df2.loc[555, "kolumna1"]  # Wartość w wierszu 555, kolumna "kolumna1"
```

- **Filtracja danych**:

```
df2[df2.kolumna1 > 0.7]   # Wiersze, gdzie wartość w "kolumna1" > 0.7
```

```
df2[df2.kolumna2.isnull()]  # Wiersze z brakującymi wartościami w
"kolumna2"
```

6. Obsługa brakujących danych

- **Sprawdzanie braków danych**:

```
df1.isna()  # Maska logiczna braków danych
df1.isna().sum()  # Liczba brakujących wartości w każdej kolumnie
```

- **Uzupełnianie braków**:

```
df1.fillna("brak")  # Uzupełnia wartości brakujące
```

7. Analiza danych

- **Podstawowe statystyki**:

```
df1.mean()      # Średnia
df1.median()    # Mediana
df1.min(), df1.max()  # Min i max
df1.sum()       # Suma wartości
```

- **Unikalne wartości i ich liczba**:

```
df1["kol2"].unique()  # Unikalne wartości
df1["kol2"].value_counts()  # Liczność wartości
```

8. Wizualizacja danych

- **Histogramy**:

```
df2.hist(bins=50, figsize=(12, 6))  # Histogram z 50 koszykami
```

- **Wykresy słupkowe**:

```
df1.plot.bar()
```

Wykład 3 – Obliczenia i transformacje w modelu Tidy Data

1. Obliczenia – Podstawowe operacje

- Tworzenie DataFrame:

```
import pandas as pd

df = pd.DataFrame({
    "imię": ["Ania", "Jan", "Adam", "Ewa", "Ola", "Olaf"],
    "brutto": [55000., 120000, 60000, 130000, 55000, 50000],
    "płeć": ["K", "M", "M", "K", "K", "M"],
    "PJ": ["N", "T", "N", "N", "N", "T"]
```

```
})
```

- **Dodawanie kolumny „podatek”**:

```
df["podatek"] = df["brutto"] * 18 / 100
```

- **Dodawanie kolumny „netto”**:
  - Wykorzystanie odejmowania:

```
df["netto"] = df["brutto"] - df["podatek"]
```

  - Wykorzystanie `apply`:

```
df["netto"] = df["brutto"].apply(lambda x: x - x * 18 / 100)
```

- **Zaawansowane obliczanie podatku**:

```
def oblicz_podatek(kwota_brutto):
    prog = 85528
    if kwota_brutto < prog:
        podatek = kwota_brutto * 18 / 100
    else:
        podatek = prog * 18 / 100 + (kwota_brutto - prog) * 32 / 100
    return podatek

df["podatek"] = df["brutto"].apply(oblicz_podatek)
df["netto"] = df["brutto"] - df["podatek"]
```

2. Transformacje danych

- **Transpozycja**:

```
df.transpose()
```

- **Grupowanie danych (`groupby`)**:
  - Grupowanie po kolumnie „płeć”:

```
df.groupby("płeć")[["brutto", "podatek"]].mean()
```

  - Grupowanie wielopoziomowe (np. „płeć” i „PJ”):

```
df.groupby(["płeć", "PJ"])[["podatek", "netto"]].mean()
```

- **Topienie (`melt`)**:
  - Zmiana układu danych:

```
df.melt(id_vars=["imię"], value_vars=["podatek", "netto"])
```

  - Dodanie własnych nazw:

```
df.melt(
    id_vars=["imię"],
    value_vars=["podatek", "netto"],
    var_name="parametr",
```

```
    value_name="wartość"
)
```

- **Tabela przestawna (`pivot_table`)**:
  - Tworzenie tabel przestawnych:

```
df.pivot_table(index="imię", columns="PJ", values=["netto",
"podatek"])
```

  - Grupowanie wielopoziomowe:

```
df.pivot_table(index="płeć", columns=["PJ"], values=["netto"],
aggfunc=[min, max])
```

- **Krzyżowe zliczenia (`crosstab`)**:

```
pd.crosstab(df["płeć"], df["PJ"])
```

3. Łączenie i modyfikowanie danych

- **Łączenie danych (`concat`)**:
  - Łączenie dwóch DataFrame:

```
df2 = pd.DataFrame({"imię": ["Oskar", "Grzegorz"], "brutto":
[75000., 62000], "płeć": ["M", "M"]})
pd.concat((df, df2))
```

  - Resetowanie indeksu po połączeniu:

```
pd.concat((df, df2)).reset_index(drop=True)
```

- **Usuwanie duplikatów (`drop_duplicates`)**:

```
df.drop_duplicates(subset=["imię"])
```

- **Łączenie tabel (`merge`)**:
  - Proste łączenie po wspólnej kolumnie:

```
df4 = pd.DataFrame({"imię": ["Ania", "Jan", "Adam", "Ewa"],
"nazwisko": ["Nowak", "Nowacki", "Kwiatkowski", "Olejnik"]})
df.merge(df4, on="imię")
```

  - Łączenie lewostronne:

```
df.merge(df4, on="imię", how="left")
```

4. Podstawowe informacje o brakach danych

- **Obsługa wartości `NaN`**:
  - Tworzenie wartości `NaN`:

```
import numpy as np
np.nan
```

- Sprawdzanie braków danych:

```
df.isna().sum()
```

Wykład 4 – Połączenie Pandas z Oracle

1. Wymagania i instalacje

- **Biblioteki**:
  - `cx_Oracle` – do połączenia z bazą danych Oracle.
  - `sqlalchemy` – do pracy z SQL w Pythonie.
  - Instalacja:

```
pip install cx_Oracle
pip install sqlalchemy
```

- **Pakiet instalacyjny Oracle**:
  - Pobierz klienta Oracle: Oracle Instant Client.
  - Przykładowy wątek pomocniczy: StackOverflow.

2. Konfiguracja połączenia

- Inicjalizacja klienta Oracle:

```
import cx_Oracle

lib_dir = r"D:\work\Oracle\instantclient_21_3"
cx_Oracle.init_oracle_client(lib_dir=lib_dir)
```

- Dane do połączenia:

```
oracle_pass = str(open('pass.txt', "r").readlines()[0])
con = cx_Oracle.connect('bruszczak/' + oracle_pass +
'@217.173.198.135:1521/tpdb')
```

- Test połączenia:

```
print(con.version)  # Wersja serwera
cur = con.cursor()
cur.execute("select * from dual")
for a in cur:
    print(a)
```

3. Pobieranie danych z Oracle do Pandas

- Pobieranie danych SQL:

```
import pandas as pd

df = pd.read_sql("select * from dual", con=con)
print(df)
```

- Alternatywnie z użyciem `sqlalchemy`:

```
from sqlalchemy.engine import create_engine

engine = create_engine('oracle+cx_oracle://bruszczak:' + oracle_pass
+ '@217.173.198.135:1521/?service_name=tpdb')
miejscowosci2 = pd.read_sql_query('SELECT * FROM miejscowosci',
engine)
```

4. Operacje na tabelach

- Przykład tabel:
  - `miejscowosci`, `regiony`, `kraje`.
  - Łączenie tabel (np. `merge`):

```
kraje_i_miejscowosci = regiony.merge(kraje, on="ID_KRAJU")
miejsca = miejscowosci.merge(kraje_i_miejscowosci,
on="ID_REGIONU")
```

- Przegląd danych:

```
miejsca.head(3)
miejsca.columns
miejsca.shape
```

- Filtracja:

```
miejsca.query('NAZWA_KRAJU == "Polska"')
miejsca.query("ID_MIEJSCOWOSCI < 5 and ID_KRAJU == 1")
```

- Kodowanie zmiennych kategorycznych:

```
pd.get_dummies(miejsca, columns=["NAZWA_KRAJU"], drop_first=True,
prefix="Kraj")
```

5. Analiza transakcji

- Łączenie tabel dla transakcji:

```
zakupy = zakupy.merge(szczegoly_zakupow, on="ID_ZAKUPU")\
               .merge(produkty, on="ID_PRODUKTU")\
               .merge(kategorie, on="ID_KATEGORII")\
               .drop(columns=["ID_SZCZEGOLOW_ZAKUPU", "ID_PRODUKTU",
"ID_KATEGORII"])
```

- Dodawanie kolumny „SPRZEDAZ”:

```
zakupy["SPRZEDAZ"] = zakupy["ILOSC"] * zakupy["CENA_SPRZEDAZY"]
```

6. Grupowanie danych

- **Najwięksi klienci**:

```
table1 = df.groupby(by=["IMIE_KLIENTA",
"NAZWISKO_KLIENTA"])["SPRZEDAZ"].sum().sort_values(ascending=False).h
ead(5)
```

- **Średnie ceny w kategoriach**:

```
table2 =
df.groupby(by=["NAZWA_KATEGORII"])["CENA_SPRZEDAZY"].mean().sort_valu
es(ascending=False).head(3)
```

- **Produkty z największą sprzedażą**:

```
table3 =
df.groupby(by="NAZWA")["SPRZEDAZ"].sum().sort_values(ascending=False)
.head(5)
```

7. Wizualizacja danych

- Rozkład sprzedaży:

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.displot(data=df, x="SPRZEDAZ")
plt.xlabel("Kwota sprzedaży")
plt.ylabel("Ilość")
plt.title("Wartość sprzedaży produktów")
plt.grid()
plt.show()
```

- Rozkład ilości zakupów w zależności od kategorii:

```
sns.displot(data=df, x="ILOSC", hue="NAZWA_KATEGORII", kind="kde")
plt.xlim(0, 5)
plt.show()
```

8. Zamknięcie połączenia

- Po zakończeniu pracy z bazą danych należy zamknąć połączenie:

```
con.close()
```

Wykład 4 – Tidy Data vs Messy Data

Czym jest zestaw danych?

- Większość statystycznych zestawów danych to **prostokątne tabele** złożone z wierszy i kolumn:
  - **Kolumny** są prawie zawsze oznaczone etykietami.
  - **Wiersze** czasami mają etykiety.
- **Zestaw danych** to zbiór wartości:
  - **Liczbowych** (jeśli dane są ilościowe).
  - **Łańcuchów znaków** (jeśli dane są jakościowe).

Organizacja wartości w zestawie danych

- Każda wartość należy do:
  - **Zmiennej** – reprezentuje jedną cechę, np. wysokość, temperatura, czas trwania.
  - **Obserwacji** – zawiera wartości dla jednego obiektu, np. osoby, dnia, wyścigu.

Tidy Data – Definicja

- **Tidy Data** to standard organizacji danych, który mapuje znaczenie danych na ich strukturę.
- Dane są **czyste (tidy)**, jeśli spełniają poniższe zasady:
  1. Każda zmienna tworzy **kolumnę**.
  2. Każda obserwacja tworzy **wiersz**.
  3. Każdy typ jednostki obserwacyjnej tworzy **tabelę**.
- Dane nieczyste (**messy**) to każde inne ułożenie danych.

Inspiracja normalizacją danych

- Zasady Tidy Data są zbliżone do **3. postaci normalnej (3NF)** w modelu relacyjnym, opisanej przez E.F. Codda (1990).
  Różnice:
  - Ramy są opisane w języku statystyki.
  - Skupiają się na pojedynczym zbiorze danych, a nie wielu połączonych zbiorach.

Problemy z nieczystymi danymi i ich rozwiązania

1. **Nagłówki kolumn są wartościami, a nie nazwami zmiennych**.
   Rozwiązanie: Przekształć dane tak, aby każda zmienna miała własną kolumnę.
2. **Wiele zmiennych zapisanych w jednej kolumnie**.
   Rozwiązanie: Rozdziel dane na oddzielne kolumny.
3. **Zmienne są przechowywane zarówno w wierszach, jak i kolumnach**.
   Rozwiązanie: Przekształć dane na spójny układ kolumnowy.
4. **Wiele typów jednostek obserwacyjnych przechowywanych w jednej tabeli**.
   Rozwiązanie: Rozdziel dane na osobne tabele.
5. **Jednostka obserwacyjna rozdzielona na wiele tabel**.
   Rozwiązanie: Połącz dane w jedną tabelę.

Dlaczego Tidy Data jest ważne?

- Tidy Data ułatwia:
  - **Manipulację danymi**.
  - **Modelowanie danych**.
  - **Wizualizację**.
- Dzięki spójnej strukturze:
  - Niewielki zestaw narzędzi wystarczy do czyszczenia danych.
  - Narzędzia wejściowe i wyjściowe mogą operować na spójnej strukturze.

Kluczowe korzyści

- **Łatwiejsze czyszczenie danych**:
  - Mniej czasu spędzonego na uciążliwej manipulacji.
  - Większa efektywność analizy.
- **Prostota narzędzi**:
  - Jednolite struktury danych pozwalają tworzyć specjalistyczne narzędzia, które upraszczają proces analizy.

Przykłady zastosowań

- Case study z publikacji:
  - Demonstracja korzyści wynikających z **spójnej struktury danych**.
  - Unikanie powtarzalnych i skomplikowanych operacji manipulacji.

Wykład 5 – Wizualizacje dla obiektów DataFrame (Boston Housing)

1. Wprowadzenie do zbioru danych

- **Zbiór danych**: *California Housing Dataset* (wcześniej Boston Housing).
- **Liczba obserwacji**: 20,640.
- **Liczba cech**: 8.
- **Cel analizy**:
  - *Target*: Średnia wartość domów w okręgach Kalifornii (**MedHouseVal**), wyrażona w setkach tysięcy dolarów ($100,000).

2. Cechy (kolumny) zbioru danych

- **MedInc** – średni dochód gospodarstw domowych w okręgu.
- **HouseAge** – mediana wieku domów w okręgu.
- **AveRooms** – średnia liczba pokoi na gospodarstwo domowe.
- **AveBedrms** – średnia liczba sypialni na gospodarstwo domowe.
- **Population** – liczba ludności w okręgu.
- **AveOccup** – średnia liczba członków gospodarstwa domowego.
- **Latitude** – szerokość geograficzna okręgu.
- **Longitude** – długość geograficzna okręgu.

3. Przygotowanie danych

- Załadowanie zbioru danych:

```
from sklearn.datasets import fetch_california_housing
import pandas as pd

housing = fetch_california_housing()
df = pd.DataFrame(data=housing.data, columns=housing.feature_names)
targets = pd.DataFrame(data=housing.target,
columns=housing.target_names)
df_merged = pd.merge(df, targets, left_index=True, right_index=True)
```

- Zmiana i manipulacja danymi:
  - Dodanie nowej kolumny `Age`:

```
df_merged["Age"] = "Old"
df_merged.loc[df_merged["HouseAge"] < 20, "Age"] = "Not so old"
```

4. Podstawowa analiza statystyczna

- Statystyki opisowe:

```
df_merged.describe()
df_merged.info()
```

- Grupowanie danych:
  - Średnie wartości dla grup:

```
df_merged.groupby("Age").mean()
```

  - Wartości minimalne, maksymalne i suma:

```
df_merged.groupby("Age")[["AveRooms", "AveBedrms",
"Population"]].agg([min, max, sum])
```

5. Wizualizacje danych

Scatter Plot (Rozrzut danych)

- Relacja między lokalizacją a ceną domów:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 8))
plt.scatter(x=df_merged.Longitude, y=df_merged.Latitude,
c=df_merged.MedHouseVal, alpha=0.4)
plt.grid()
plt.show()
```

Pairplot

- Zależności między różnymi cechami:

```
import seaborn as sns

plt.figure(figsize=(9, 9))
sns.pairplot(df_merged.sample(250), hue="Age", palette="tab10")
plt.show()
```

Heatmap (Mapa cieplna korelacji)

- Analiza korelacji między cechami:

```
corr = df_merged.iloc[:, :-1].corr()
mask = np.triu(np.ones_like(corr, dtype=bool))
f, ax = plt.subplots(figsize=(7, 6))
sns.heatmap(corr, mask=mask, center=0, square=True, cmap="Spectral",
vmin=-1, vmax=1)
plt.show()
```

6. Model predykcyjny

- **Model regresji Random Forest**:

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, r2_score

X = df_merged[housing['feature_names']]
y = df_merged[housing['target_names']]

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.33, random_state=42)
regr = RandomForestRegressor(max_depth=2, random_state=0)
regr.fit(X_train, y_train)

y_pred = regr.predict(X_test)
print(f"Błąd (MAE): {mean_absolute_error(y_test, y_pred):.2f}")
print(f"Współczynnik determinacji (R²): {r2_score(y_test,
y_pred):.3f}")
```

    - **Błąd (MAE)**: 0.65.
    - **Współczynnik determinacji (R^2)**: 0.453.

7. Dodatkowe wizualizacje

Histogramy i gęstości

- Histogram mediany wartości domów:

```
df_merged["MedHouseVal"].hist(bins=15)
```

- Gęstość wartości domów:

```
sns.displot(data=df_merged, x="MedHouseVal", kind="kde", height=6)
plt.show()
```

- Gęstość 2D między wartością domów a dochodami:

```
sns.kdeplot(data=df_merged, x="MedHouseVal", y="MedInc", fill=True)
plt.show()
```

Regresja liniowa

- Relacja między liczbą sypialni a wartością domów:

```
g = sns.lmplot(data=df_merged, x="AveBedrms", y="MedHouseVal",
hue="Age", height=5)
g.set_axis_labels("Average Bedrooms", "Value (100 000$)")
plt.xlim(0, 15)
plt.ylim(-2, 5)
plt.show()
```

Wykład 6 – Wizualizacje dla obiektów DataFrame (Unicorn Companies)

1. Wprowadzenie do danych

- **Źródło danych**: Zbiór danych dotyczących firm typu unicorn (firmy wyceniane na ponad 1 miliard dolarów).
- **Liczba obserwacji**: 1037.
- **Liczba cech**: 13.
- **Przykładowe kolumny**:
  - `Company` – nazwa firmy.
  - `Valuation ($B)` – wycena firmy w miliardach dolarów.
  - `Date Joined` – data, kiedy firma osiągnęła status unicorn.
  - `Country` i `City` – kraj i miasto siedziby firmy.
  - `Industry` – sektor działalności firmy.
  - `Total Raised` – całkowita kwota zebrana przez firmę w ramach finansowania.
  - `Investors Count` – liczba inwestorów.

2. Przygotowanie danych

- Import bibliotek i wczytanie danych:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

df = pd.read_csv("Unicorn_Companies.csv")
```

- Przetwarzanie danych:
  - Usunięcie braków w kolumnie `Total Raised`:

```
df.dropna(subset=["Total Raised"], inplace=True)
```

  - Konwersja wartości pieniężnych na liczby (usunięcie `$`, zamiana `M`, `B` na liczby):

```
d = {"\$": "", "K": "*1000", "M": "*1000000", "B":
"*1000000000"}
cols = ["Total Raised", "Valuation ($B)"]
for col in cols:
    df[col] = df[col].replace(d, regex=True).map(pd.eval)
```

  - Konwersja daty do formatu `datetime`:

```
df["Date Joined"] = pd.to_datetime(df["Date Joined"],
format='%m/%d/%Y')
```

3. Podstawowe analizy

- **Opis statystyczny danych**:

```
df.describe()
```

- **Najwięcej firm według krajów**:

```
df.groupby(["Country"]).count()["Company"].sort_values(ascending=Fals
e)[:10]
```

    - **Top kraje**:
        1. Stany Zjednoczone: 519 firm.
        2. Chiny: 164 firmy.
        3. Indie: 62 firmy.
- **Najwięcej firm według miast**:

```
df.groupby(["City"]).count()["Company"].sort_values(ascending=False)[
:10]
```

    - **Top miasta**:
        1. San Francisco: 141 firm.
        2. Nowy Jork: 92 firmy.
        3. Pekin: 61 firm.
- **Najpopularniejsze branże**:

```
df.groupby(["Industry"]).count()["Company"].sort_values(ascending=Fal
se)[:10]
```

    - **Top branże**:
        1. Fintech: 198 firm.
        2. Internet software & services: 189 firm.
        3. E-commerce & direct-to-consumer: 107 firm.

4. Wizualizacje danych

Histogram roku założenia firm

- Rozkład liczby firm w zależności od roku założenia:

```
df["Founded Year"].value_counts().sort_index().plot(kind="bar")
plt.show()
```

Scatter plot – Total Raised vs Valuation

- Relacja między łączną zebraną kwotą a wyceną firm:

```
plt.figure(figsize=(9, 5))
sns.scatterplot(data=df, x="Valuation ($B)", y="Total Raised",
hue="Country")
plt.xscale("log")
plt.grid()
plt.show()
```

Pivot table – Liczba inwestorów w zależności od branży i kraju

- Analiza liczby inwestorów dla wybranych krajów i branż:

```
countries = df.Country.value_counts()[:4].index.values
industries = df.Industry.value_counts()[:4].index.values
df2 = df[df.Country.isin(countries) & df.Industry.isin(industries)]
```

```
pivot = df2.pivot_table(index="Country", columns="Industry",
values="Investors Count", aggfunc="count", fill_value="")
print(pivot)
```

Scatter plot dla wybranych krajów i branż

- Wizualizacja wyceny i zebranej kwoty dla najpopularniejszych branż i krajów:

```
plt.figure(figsize=(9, 5))
sns.scatterplot(data=df2, x="Valuation ($B)", y="Total Raised",
hue="Country")
plt.xscale("log")
plt.grid()
plt.show()
```

5. Ranking firm

- **Top 10 firm według wyceny**:

```
df["Valuation ($B)"].sort_values(ascending=False)[:10]
```

  - Najwyższa wycena: ByteDance – 140 mld USD.
- **Top 10 firm według zebranej kwoty**:

```
df["Total Raised"].sort_values(ascending=False)[:10]
```

  - Najwięcej zebranej kwoty: ByteDance – 7.44 mld USD.

Wykład 7 – Importy, Połączenie do Baz Danych, Cache

1. Importy i konfiguracja środowiska

- **Podstawowe biblioteki**:
  - `matplotlib.pyplot`, `seaborn` – wizualizacja danych.
  - `pandas`, `numpy` – manipulacja danymi.
  - `pickle` – serializacja obiektów do formatu binarnego.
- **Biblioteki do połączenia z bazami danych**:
  - `cx_Oracle` – połączenie z bazą Oracle.
  - `sqlalchemy` – silnik do wykonywania zapytań SQL.
  - `redis` – obsługa bazy danych typu cache.
- **Konfiguracja klienta Oracle**:

```
import cx_Oracle
from sqlalchemy import create_engine

cx_Oracle.init_oracle_client(lib_dir=r"D:\work\Oracle\instantclient_2
1_3")

oracle_pass = str(open('pass.txt', "r").readlines()[0])
oracle_engine = create_engine(

f"oracle+cx_oracle://bruszczak:{oracle_pass}@217.173.198.135:1521/?se
rvice_name=tpdb&encoding=UTF-8"
)
```

- **Połączenie z Redis**:

```
import redis
r = redis.Redis(host="10.1.48.189", db=7)
```

2. Przykład – Analiza danych giełdowych (NASDAQ)

- **Dane**:
  - Plik CSV z notowaniami największych spółek technologicznych.
  - Kolumny: `QUOTATION_DATE`, `CLOSE_PRICE`, `STOCK`.
- **Przykładowa wizualizacja notowań spółek**:

```
import plotly.express as px

df = pd.read_csv("data/Quotations.csv")
fig = px.line(df, x="QUOTATION_DATE", y="CLOSE_PRICE", color="STOCK",
title='NASDAQ Quotations')
fig.show()
```

3. Odczyt danych z bazy Oracle

- **Funkcja odczytu danych**:

```
def read_from_database(date, stock):
    query = f"""
        SELECT *
        FROM QUOTATIONS
        WHERE STOCK = '{stock}'
        AND QUOTATION_DATE <= TO_DATE('{date}', 'YYYY-MM-DD')
        ORDER BY QUOTATION_DATE DESC
    """
    record = oracle_engine.execute(query).fetchone()
    return record
```

- **Przykładowy odczyt**:

```
print(read_from_database("2020-09-04", "MSFT"))
```

4. Obsługa cache (Redis)

- **Definicje funkcji dla odczytu i zapisu cache**:

```
import pickle

CACHE_DURATION_SEC = 3600  # Czas przechowywania danych w cache.

def read_from_cache(date, stock):
    data = r.get(stock + ":" + date)
    if data:
        return pickle.loads(data)

def write_to_cache(date, stock, data):
    r.set(stock + ":" + date, pickle.dumps(data),
ex=CACHE_DURATION_SEC)
```

- **Zastosowanie cache w praktyce**:

- Przykład zapisu danych do cache:

```
data = read_from_database("2021-03-12", "TSLA")
write_to_cache("2021-03-12", "TSLA", data)
```

5. Łączenie cache z bazą danych

- **Funkcja odczytu z priorytetem cache**:
  1. Sprawdź dane w cache.
  2. Jeśli brak w cache, pobierz z bazy Oracle.
  3. Zapisz dane do cache na przyszłość.

```
def get_record(date, stock):
    info = "CACHE"
    record = read_from_cache(date, stock)

    if not record:
        info = "SQL DATABASE"
        record = read_from_database(date, stock)
        write_to_cache(date, stock, record)

    return (info, record)
```

- **Przykładowe użycie**:

```
print(get_record("2020-05-10", "META"))
```

6. Testy porównawcze

- **Losowanie przykładowych rekordów**:

```
dates = ["2019-03-12", "2020-07-02", "2021-01-01"]
stocks = ["AAPL", "MSFT", "TSLA", "AMZN"]

examples = [[dates[i], stocks[j]] for i, j in np.random.randint(0, 3,
(5, 2))]
for date, stock in examples:
    print(get_record(date, stock))
```

7. Zaawansowane zapytania do cache

- **Dane w zakresie dat**:

```
def write_to_cache_date_span(date1, date2, stock, data):
    r.set(stock + ":" + date1 + "-" + date2, pickle.dumps(data),
ex=CACHE_DURATION_SEC)

def read_from_cache_date_span(date1, date2, stock):
    data = r.get(stock + ":" + date1 + "-" + date2)
    return pickle.loads(data) if data else None
```

- **Przykład użycia**:

```
write_to_cache_date_span("2021-12-25", "2022-05-25", "IBM", "dummy
data")
```

```
print(read_from_cache_date_span("2021-12-25", "2022-05-25", "IBM"))
```

8. Zapytania z agregacją danych

- **Grupowanie według metod statystycznych (np. średnia)**:

```
def write_to_cache_period_aggr(date1, date2, stock, aggregator, data,
method="AVG"):
    r.set(stock + ":" + aggregator + ":" + date1 + "-" + date2,
pickle.dumps(data), ex=CACHE_DURATION_SEC)

def read_from_cache_period_aggr(date1, date2, stock, aggregator,
method="AVG"):
    data = r.get(stock + ":" + aggregator + ":" + date1 + "-" +
date2)
    return pickle.loads(data) if data else None
```

- **Przykład użycia**:

```
write_to_cache_period_aggr("2011-01-01", "2021-12-31", "IBM", "YEAR",
"aggregated data")
print(read_from_cache_period_aggr("2011-01-01", "2021-12-31", "IBM",
"YEAR"))
```

Wykład 8: Operatory, Projekcje i Sortowanie w MongoDB

MongoDB to nierelacyjna baza danych, która umożliwia manipulację danymi w formacie JSON. W tym wykładzie omawiamy podstawowe operatory, projekcje oraz metody sortowania danych.

Operatory w zapytaniach MongoDB

Operatory logiczne pozwalają na tworzenie warunków filtracji podczas wykonywania zapytań.

- **$eq** – Porównanie: wartość równa podanej.

```
{"field": {"$eq": value}}
```

- **$gt** – Większe niż:

```
{"field": {"$gt": value}}
```

- **$gte** – Większe lub równe:

```
{"field": {"$gte": value}}
```

- **$lt** – Mniejsze niż:

```
{"field": {"$lt": value}}
```

- **$lte** – Mniejsze lub równe:

```
{"field": {"$lte": value}}
```

- **$in** – Dowolna wartość z podanego zbioru:

```
{"field": {"$in": [value1, value2, value3]}}
```

Projekcje

Projekcje pozwalają na określenie, które pola mają być zwracane w wynikach zapytania.

- **Przykład 1**: Wyświetlenie pełnych dokumentów z określonym filtrem:

```
for customer in customers.find(filter_2, {"name",
"gender"}).limit(5):
    print(customer)
```

Wynik:

```
{"_id": "...", "name": "Hannah", "gender": "Female"}
{"_id": "...", "name": "Sandy", "gender": "Female"}
```

- **Przykład 2**: Wykluczenie `_id` z wyników:

```
for customer in customers.find(filter_2, {"_id": 0, "name": 1,
"gender": 1}).limit(5):
    print(customer)
```

Wynik:

```
{"name": "Hannah", "gender": "Female"}
{"name": "Sandy", "gender": "Female"}
```

Sortowanie wyników

Sortowanie wyników zapytania odbywa się za pomocą metody `sort`.

- **Przykład sortowania**: Sortowanie według dwóch kryteriów:
    1. Pole `gender` w kolejności malejącej (wartość -1).
    2. Pole `name` w kolejności rosnącej (wartość 1).

```
for customer in customers.find(
    filter={},
    projection={"gender": 1, "name": 1, "_id": 0},
    sort=[("gender", -1), ("name", 1)]
):
    print(customer)
```

Wykład 9: Podstawowe konfiguracje MongoDB w Pythonie

Instalacja i importowanie modułów

Aby pracować z MongoDB w Pythonie, należy zainstalować odpowiednią bibliotekę, np. `pymongo`:

```
# Instalacja za pomocą pip
!pip install pymongo

# Instalacja za pomocą conda
!conda install -c anaconda pymongo
```

Import wymaganych bibliotek:

```
from pymongo import MongoClient
import pprint
import datetime
```

Połączenie z MongoDB

Istnieje kilka sposobów połączenia się z bazą MongoDB:

```
# Proste połączenie
client = MongoClient()

# Połączenie z podaniem URI
client = MongoClient("mongodb://localhost:27017/")

# Połączenie z podaniem hosta i portu
client = MongoClient('localhost', 27017)
```

Sprawdzenie połączenia:

```
print(client)
# MongoClient(host=['localhost:27017'], document_class=dict,
tz_aware=False, connect=True)
```

Bazy danych

Wyświetlenie listy istniejących baz danych:

```
print(client.list_database_names())
# ['admin', 'config', 'customers_database', 'local']
```

Tworzenie lub wybór istniejącej bazy danych:

```
db = client.test_database
db = client["test-database2"]  # Alternatywny sposób
```

Usunięcie bazy danych:

```
client.drop_database("test-database2")
```

Kolekcje

Tworzenie lub wybór istniejącej kolekcji:

```
collection = db.test_collection
collection = db['test-collection']
```

Sprawdzenie dostępnych kolekcji w bazie:

```
db.list_collection_names()
```

Dodawanie dokumentów

Wprowadzenie pojedynczego dokumentu:

```
post = {
    "autor": "Janusz",
    "tekst": "Siema ziomeczki, poznajcie moje biznesy!",
    "tagi": ["kasa", "super_interesy", "gold"],
    "czas": datetime.datetime.utcnow()
}

post_id = db.posty.insert_one(post).inserted_id
print(post_id)
```

Wprowadzenie wielu dokumentów:

```
try:
    db.posty.insert_many([
        {"autor": "Adam", "tekst": "Biznes się kręci", "tagi": ["biznes",
"gold"]},
        {"autor": "Ewa", "tekst": "Nowe inwestycje", "tagi": ["inwestycje",
"kasa"]},
    ])
except Exception as e:
    print(e)
```

Zapytania

Odczyt pojedynczego dokumentu:

```
pprint.pprint(db.posty.find_one({"autor": "Janusz"}))
```

Odczyt wielu dokumentów:

```
for post in db.posty.find({"tagi": {"$in": ["kasa", "gold"]}}):
    pprint.pprint(post)
```

Agregacje

Agregacje pozwalają na zaawansowane grupowanie i filtrowanie danych.

Przykład grupowania po autorze:

```
results = db.posty.aggregate([
    {
        "$group": {
            "_id": "$autor",
            "ile": {"$sum": 1}
```

```
        }
    }
])

for result in results:
    print(result)
```

Usuwanie danych i kolekcji

Usunięcie kolekcji:

```
db["posty"].drop()
```

Usunięcie bazy danych:

```
client.drop_database("test-database2")
```