

Geokodowanie danych adresowych w pythonie

Dokument opisuje kompletny pipeline do geokodowania adresów w środowisku Python z wykorzystaniem Google Maps Geocoding API oraz integracją wyników w Google BigQuery. Celem rozwiązania jest masowe przetwarzanie rekordów adresowych, optymalizacja liczby zapytań poprzez deduplikację i cache'owanie oraz transakcyjne uzupełnianie brakujących współrzędnych geograficznych w produkcyjnej bazie danych.

Proces obejmuje generowanie deterministycznych kluczy adresowych, budowę warstwy cache redukującej zapytania do API, wielostopniowy fallback zapytań (od adresów pełnych do uproszczonych), implementację mechanizmów retry z narastającym backoffem oraz walidację zakresów współrzędnych. Wyniki geokodowania są ładowane partiami do tabel tymczasowych i scalane z tabelą źródłową za pomocą operacji MERGE, co zapewnia idempotentność oraz pełną spójność danych. Pipeline został zaprojektowany jako odporny, skalowalny i zoptymalizowany pod względem kosztów API.

Normalizacja adresów, geokodowanie i integracja z tabelą produkcyjną.

Pierwszym etapem procesu jest wstępne oczyszczenie danych wejściowych polegające na usunięciu zbędnych spacji i ujednoliceniu wielkości liter. Jeżeli adres jest zapisany w postaci listy elementów (*adres_array*), tworzony jest z niego jednoznaczny klucz tekstowy, łączący wszystkie niepuste elementy listy w stałej kolejności. Takie podejście eliminuje różnice wynikające z kolejności i interpunkcji, a tym samym ogranicza fałszywe duplikaty.

```
def to_arr_key(v):
    # v może być listą/ndarray/None
    try:
        seq = list(v) if v is not None else []
    except TypeError:
        seq = []
    # wyczyść puste elementy i połącz
    items = [str(x).strip() for x in seq if str(x).strip()]
    return " | ".join(items) if items else ""

df_ids["adres_array_key"] = df_ids["adres_array"].apply(to_arr_key)
```

Normalizacja adresów

Kolejnym etapem jest redukcja liczby zapytań do geokodera poprzez wyodrębnienie zbiorów o unikalnych kombinacjach (*adres, miasto, adres_array_key*), z

których każdy zbiór reprezentuje ten sam adres zapisany w różnych wariantach. W praktyce oznacza to, że jedno zapytanie zasila wiele wierszy w bazie co znaczco obniża koszty i czas procesu.

```
# unikalne wiersze po (adres, miasto, adres_array_key)
df_unique = (
    df_ids[["adres", "miasto", "adres_array_key"]]
    .drop_duplicates()
    .reset_index(drop=True)
)
# mapowanie: (adres, miasto, adres_array_key) -> Lista row_id
key_map = (
    df_ids.groupby(["adres", "miasto", "adres_array_key"])["row_id"]
    .apply(list).to_dict()
)

print("Unikalnych (adres, miasto, adres_array_key):", len(df_unique))
```

Wyodrębnienie unikalnych kombinacji adresów

Wywołania do usługi geokodowania są wykonywane z ustawionym językiem polskim oraz z parametrami ograniczającymi wyszukiwania do Polski i miasta Poznań co podnosi precyze wyników. Dla każdego rekordu generowany jest zestaw zapytań weryfikujących począwszy od pełnego adresu, a kończąc na coraz prostszych formatach. Procedura kończy się po uzyskaniu pierwszego wiarygodnego rezultatu.

W trakcie jednego uruchomienia procesu wykorzystywana jest pamięć podręczna oparta na kluczu (*adres, miasto, adres_array_key*). Każda unikalna kombinacja pól jest geokodowana tylko raz, a uzyskany wynik przypisywany jest równocześnie do wszystkich odpowiadających jej rekordów.

Mechanizm obsługi błędów działa w oparciu o ponawianie prób z coraz dłuższymi przerwami. System radzi sobie z brakiem wyniku, przekroczeniem limitów zapytań i błędami sieci. W każdej z tych sytuacji zapisywany jest bezpieczny status, tak aby nie nadpisać poprawnych danych i zachować pełną historię działania. Dodatkowo sprawdzany jest zakres współrzędnych, co pozwala uniknąć błędnych lokalizacji.

Każdy wynik geokodowania zawiera współrzędne (*lat* i *lng*), sformatowany adres, identyfikator miejsca (*place_id*), pole określające źródło lub tryb działania (np. *fallback*) oraz znacznik czasu. Dzięki temu dane można wykorzystać zarówno do precyzyjnych analiz przestrzennych, jak i do śledzenia ich pochodzenia oraz jakości. Takie podejście sprawia, że proces jest powtarzalny, odporny na chwilowe błędy i oszczędny w użyciu.

```

def geocode_with_fallback(adres: str, miasto: str, arr_key: str):
    key = (_lc(adres), _lc(miasto), arr_key)
    if key in _cache:
        return _cache[key]

    for q in candidate_queries(adres, miasto, arr_key):
        delay = SLEEP_BETWEEN
        for attempt in range(1, MAX_RETRIES + 1):
            try:
                res = gmaps.geocode(
                    q, language="pl",
                    components={"country": "PL", "locality": "Poznań"})
            )
            if res:
                loc = res[0]["geometry"]["location"]
                out = {
                    "lat": float(loc["lat"]),
                    "lng": float(loc["lng"]),
                    "formatted_address": res[0].get("formatted_address"),
                    "place_id": res[0].get("place_id"),
                    "source": "maps_geocoding_v1",
                    "geocode_ts": datetime.now(timezone.utc)
                }
                _cache[key] = out
                return out
            break
        except Exception as e:
            if attempt == MAX_RETRIES:
                _cache[key] = {
                    "lat": None, "lng": None,
                    "formatted_address": None, "place_id": None,
                    "source": f"error:{e}",
                    "geocode_ts": datetime.now(timezone.utc)
                }
                return _cache[key]
            time.sleep(delay); delay *= RETRY_BACKOFF
        finally:
            time.sleep(SLEEP_BETWEEN)

    out = {
        "lat": None, "lng": None,
        "formatted_address": None, "place_id": None,
        "source": "no_result",
        "geocode_ts": datetime.now(timezone.utc)
    }
    _cache[key] = out
    return out

```

Geokodowanie adresów przy użyciu API Google

Po uzyskaniu wyników geokodowania dane są scalane z tabelą produkcyjną w *BigQuery*. Początkowo partie rekordów trafiają do tabeli tymczasowej, po czym wykonywana jest operacja *MERGE*, w ramach której do odpowiednich wierszy za pomocą stabilnego identyfikatora *row_id* uzupełniane są wartości *lat* i *lng* na podstawie których wyliczany jest również *geo_point* zapisywany w typie *GEOGRAPHY* oraz dodatkowe pola jak sformatowany adres, id miejsca, źródło geokodowania i znacznik czasu

```

merge_sql = f"""
MERGE `{SRC_TABLE}` AS T
USING `{tmp_table}` AS G
ON COALESCE(
    CAST(T.offer_id AS STRING),
    CAST(FARM_FINGERPRINT(
        CONCAT(IFNULL(T.adres,''), IFNULL(STR(T.miasto,''), IFNULL(STR(T.cena AS STRING),'')))
        AS STRING)
    ) = G.row_id,
    AND LOWER(T.miasto) IN ('poznan','poznań')
)
WHEN MATCHED AND G.lat IS NOT NULL AND G.lng IS NOT NULL THEN
UPDATE SET
    T.lat      = G.lat,
    T.lng     = G.lng,
    T.formatted_address = G.formatted_address,
    T.place_id = G.place_id,
    T.geocode_source = G.source,
    T.geocode_ts   = G.geocode_ts,
    T.geo_point   = ST_GEOGPOINT(G.lng, G.lat)
"""

bq.query(merge_sql).result()
bq.delete_table(tmp_table, not_found_ok=True)

```

Scalanie wyników geokodowania z tabelą produkcyjną w BigQuery

Działanie procesu było śledzone na bieżąco dzięki logom kontrolnym wypisywanym w ustalonych odstępach. Komunikaty w formacie „*Checkpoint X/Y | OK=... / NORES=...*” informowały o postępie i skuteczności pomiędzy punktami kontrolnymi, natomiast komunikat „*Final flush*” podsumowywał końcowe zapisanie wyników. Logi stanowią podstawę do oceny stabilności całego *pipeline’u*.

W przypadku Poznania rezultaty wskazują wysoką jakość i pełne pokrycie danych. Zidentyfikowano 727 unikalnych kluczy geokodowania, co odpowiada rzeczywistej liczbie zapytań wysyłanych do usługi. Skuteczność dla Poznania wynosiła 100%.

```

Do geokodowania (wiersze): 5029
Unikalnych (adres, miasto, adres_array_key): 727
Checkpoint 41/727 | OK=41 / NORES=0
Checkpoint 81/727 | OK=81 / NORES=0
Checkpoint 122/727 | OK=122 / NORES=0
Checkpoint 176/727 | OK=176 / NORES=0
Checkpoint 231/727 | OK=231 / NORES=0
Checkpoint 299/727 | OK=299 / NORES=0
Checkpoint 432/727 | OK=432 / NORES=0
Checkpoint 544/727 | OK=544 / NORES=0
Checkpoint 635/727 | OK=635 / NORES=0
Checkpoint 684/727 | OK=684 / NORES=0
Checkpoint 689/727 | OK=689 / NORES=0
Checkpoint 693/727 | OK=693 / NORES=0
Checkpoint 700/727 | OK=700 / NORES=0
Checkpoint 705/727 | OK=705 / NORES=0
Checkpoint 710/727 | OK=710 / NORES=0
Checkpoint 715/727 | OK=715 / NORES=0
Checkpoint 722/727 | OK=722 / NORES=0
Checkpoint 726/727 | OK=726 / NORES=0
Final flush | OK=727 / NORES=0
      wiersze_pozn  z_lat_lng  z_geo_point
      0            5300       5300

```

Monitorowanie postępu geokodowania

Proces tworzenia kafli geograficznych na mapie i metryki cen za m².

Celem tego etapu jest utworzenie warstwy zagregowanych kafli do wykorzystania w procesie tworzenia map cieplnych oraz wykresów przestrzennych. Jednostką agregacji jest kafel o rozmiarze około 0,001° co w przybliżeniu przekłada się na ≈ 111 metrów w kierunku północ-południe oraz ≈ 68-70 metrów w kierunku wschód - zachód. Należy pamiętać, że rozmiar długości geograficznej, kierunku wschód-zachód nie jest stały, zależy on od szerokości geograficznej i maleje zgodnie z cosφ. Na podstawie wzoru z wikipedii „Length of a degree of longitude” obliczono szerokość siatki:

- $\Delta_{[long]}^{(1)} = \frac{\pi}{180} a \cos \phi$
- $\Delta_{[long]}^{(1)}$ - Długość odcinka równoleżnika odpowiadająca zmianie 1° długości geograficznej na szerokości φ.
- $\pi/180$ - przelicznik stopni na radiany.
- $\cos \phi$ - współczynnik skrócenia długości równoleżnika wraz ze wzrostem szerokości.
- $a=6\ 378\ 137\text{ m}$ (promień równikowy WGS-84)
- $\phi= 52$ (szerokość geograficzna)
- $(\pi/180)a = 111\ 320\text{ m}$
- $\cos 52 \approx 0,61566$

Dostosowując wzór do wielkości kafla:

- $\times 111\ 320\text{ m} \times 0,61566 \approx 68.5\text{ m.}$

Współrzędne ofert są zaokrąglane do trzech miejsc po przecinku, co wyznacza centroid kafla dla którego tworzony jest punkt geograficzny służący do wizualizacji i połączeń przestrzennych. W każdym kaflu wyznaczane są podstawowe takie jak: liczba ofert, średnia, mediana, cena minimalna i maksymalna za m². Raportowane są jedynie kafle z co najmniej trzema obserwacjami.

```

    WHERE lat IS NOT NULL AND lng IS NOT NULL
    | AND (cena_za_m2 IS NOT NULL OR (cena IS NOT NULL AND metraz_m2 IS NOT NULL AND metraz_m2 > 0))
    | AND LOWER(miasto) IN ('poznan', 'poznań')
),
binned AS (
    SELECT
        ROUND(lat, 3) AS lat_bin,
        ROUND(lng, 3) AS lng_bin,
        cena_m2
    FROM src
)
SELECT
    ST_GEOPOINT(lng_bin, lat_bin) AS geo_point, -- pole geograficzne
    lat_bin, lng_bin,
    COUNT(*) AS liczba_ofert,
    APPROX_QUANTILES(cena_m2, 101)[OFFSET(50)] AS sr_m2,
    MIN(cena_m2) AS mediana_m2,
    MAX(cena_m2) AS min_m2,
    AS max_m2
FROM binned
GROUP BY lat_bin, lng_bin
HAVING COUNT(*) >= 3
ORDER BY lat_bin, lng_bin;

```

Proces tworzenia kafli geograficznych

Podsumowując dane dla Poznania są kompletne i obejmują 5300 ofert z pełnym uzupełnieniem o współrzędne i punkt geograficzny. Agregacja kafli wygenerowała 304 komórki. Mediana z median to 11 500 PLN/m², średnia z median to 12 066 PLN/m² (zakres median: 6 991 - 27 362). Globalne wartości średnie wskazują na spójny obraz rynku: mediana ≈ 11 849 PLN/m², średnia ≈ 12 303 PLN/m², a najwyższa średnia cena za m² dla kafla wynosiła ≈ 24 911 PLN.

Najgęstsze kafle skupiają od 81 do 177 ofert a ich mediany mieszczą się w przedziale ≈ 10.6-14.5 tys. PLN/m². Tak duża liczba obserwacji w wybranych obszarach zapewnia stabilne i wiarygodne wyniki.

Test stabilności median przy zmianie siatki z 0,001° na 0,0005° pokazał średnią różnicę około 66.5 PLN/m², przy maksymalnej różnicy 4 241 PLN/m² co wskazuje, że zagregowane metryki są odporne na zmianę rozdzielczości z wyjątkiem obszarów o małej liczbie obserwacji.

```

    === Poznań: pokrycie lat/lng/geom ===
      rows_pozn  rows_with_latlng  rows_with_geom
0            5300              5300            5300

    === Unikatowe klucze do geokodowania (braki) ===
      rows_to_geocode  unique_keys
0                  0                0

    === Kafle (n>=3): statystyki ceny m2 ===
      tiles_ge3  mediana_min  mediana_med  mediana_mean  mediana_max  sr_min \
0            304       6991.0     11500.0      12066.0     27362.0   6567.0

      sr_med  sr_mean  sr_max
0  11849.0  12303.0  24911.0

    === TOP 10 najczęstszych kafli ===
      lat_bin  lng_bin  liczba_ofert  mediana_m2  sr_m2
0  52.431  16.936        177    11663.0  11852.0
1  52.369  16.941        170    11453.0  11701.0
2  52.451  16.948        146    11927.0  12020.0
3  52.446  16.954        143    12100.0  12075.0
4  52.383  16.839        141    10993.0  11023.0
5  52.387  16.951        138    10596.0  10802.0
6  52.418  16.902        131    14600.0  14641.0
7  52.444  16.894         95    11457.0  11921.0
8  52.416  16.902         88    12281.0  12432.0
9  52.398  16.908         81    14500.0  15475.0

    === Stabilność mediany (0.001° vs 0.0005°) ===
      matched_tiles  delta_med_avg  delta_med_max
0             358       66.541899        4241.0

    === Szacowany czas przebiegu wg geocode_ts ===
      first_ts           last_ts  elapsed_s
0  2025-09-04 10:51:39.074768+00:00 2025-09-04 11:23:30.022603+00:00        1910

```

Podsumowanie mapy kafli geograficznych

Kod python

Geokodowanie

```
import os, time, uuid
from datetime import datetime, timezone
import pandas as pd
from google.cloud import bigquery
import googlemaps

PROJECT  = "realstate-market-poland"
DATASET  = "prod"
TABLE    = "offers_sample_per_city"
SRC_TABLE = f"{PROJECT}.{DATASET}.{TABLE}"

API_KEY = os.environ["MAPS_API_KEY"].strip()
assert API_KEY.startswith("AIza")

QPS, SLEEP_BETWEEN = 8, 1/8
MAX_RETRIES, RETRY_BACKOFF = 4, 1.8
FLUSH_EVERY = 250

bq    = bigquery.Client(project=PROJECT)
gmaps = googlemaps.Client(key=API_KEY)

# 1) Rekordy do geokodowania –Poznań
query_fetch = f"""
SELECT
    COALESCE(
        CAST(offer_id AS STRING),
        CAST(FARM_FINGERPRINT(
            CONCAT(IFNULL(adres,""), IFNULL(miasto,""), IFNULL(CAST(cena AS STRING), ""))
        ) AS STRING)
    ) AS row_id,
    adres,
    miasto,
```

```

adres_array
FROM `{{SRC_TABLE}}`
WHERE (lat IS NULL OR lng IS NULL)
    AND adres IS NOT NULL AND TRIM(adres) <> ""
    AND LOWER(miasto) IN ('poznan','poznań')
"""

df_ids = bq.query(query_fetch).result().to_dataframe()
print("Do geokodowania (wiersze):", len(df_ids))
if df_ids.empty:
    raise SystemExit("Brak rekordów do geokodowania.")

```

```

def to_arr_key(v):
    # v może być listą/ndarray/None
    try:
        seq = list(v) if v is not None else []
    except TypeError:
        seq = []
    # wyczyść puste elementy i połącz
    items = [str(x).strip() for x in seq if str(x).strip()]
    return " | ".join(items) if items else ""

```

```
df_ids["adres_array_key"] = df_ids["adres_array"].apply(to_arr_key)
```

```

# unikalne wiersze po (adres, miasto, adres_array_key)
df_unique = (
    df_ids[["adres", "miasto", "adres_array_key"]]
    .drop_duplicates()
    .reset_index(drop=True)
)
# mapowanie: (adres, miasto, adres_array_key) -> lista row_id
key_map = (
    df_ids.groupby(["adres", "miasto", "adres_array_key"])["row_id"]
    .apply(list).to_dict()
)

```

```

print("Unikalnych (adres, miasto, adres_array_key):", len(df_unique))

def _lc(x): return (x or "").strip().lower()

def candidate_queries(adres: str, miasto: str, arr_key: str):
    base = (adres or "").strip()
    city = (miasto or "Poznań").strip()
    joined = arr_key or ""
    first = joined.split(" | ")[0] if joined else ""

    cands = [
        f"{base}, {city}, Polska" if base else None,
        f"{base}, Polska" if base else None,
        f"{joined}, {city}, Polska" if joined else None,
        f"{first}, {city}, Polska" if first else None,
        f"{city}, Polska",
    ]
    out, seen = [], set()
    for q in cands:
        if q and q not in seen:
            out.append(q); seen.add(q)
    return out

_cache = {}

def geocode_with_fallback(adres: str, miasto: str, arr_key: str):
    key = (_lc(adres), _lc(miasto), arr_key)
    if key in _cache:
        return _cache[key]

    for q in candidate_queries(adres, miasto, arr_key):
        delay = SLEEP_BETWEEN
        for attempt in range(1, MAX_ATTEMPTS + 1):
            try:

```

```
res = gmaps.geocode(
    q, language="pl",
    components={"country": "PL", "locality": "Poznań"}
)
if res:
    loc = res[0]["geometry"]["location"]
    out = {
        "lat": float(loc["lat"]),
        "lng": float(loc["lng"]),
        "formatted_address": res[0].get("formatted_address"),
        "place_id": res[0].get("place_id"),
        "source": "maps_geocoding_v1",
        "geocode_ts": datetime.now(timezone.utc)
    }
    _cache[key] = out
    return out
break
except Exception as e:
    if attempt == MAX_ATTEMPTS:
        _cache[key] = {
            "lat": None, "lng": None,
            "formatted_address": None, "place_id": None,
            "source": f"error:{e}",
            "geocode_ts": datetime.now(timezone.utc)
        }
    return _cache[key]
time.sleep(delay); delay *= RETRY_BACKOFF
finally:
    time.sleep(SLEEP_BETWEEN)

out = {
    "lat": None, "lng": None,
    "formatted_address": None, "place_id": None,
    "source": "no_result",
    "geocode_ts": datetime.now(timezone.utc)
```

```
        }

        _cache[key] = out
        return out

def flush_chunk(df_chunk: pd.DataFrame):
    if df_chunk.empty:
        return

    df_chunk = (
        df_chunk.sort_values('geocode_ts')
        .drop_duplicates(subset=['row_id'], keep='last')
    )

    tmp_table = f'{PROJECT}.{DATASET}.tmp_geo_{uuid.uuid4().hex[:10]}'
    job = bq.load_table_from_dataframe(
        df_chunk, tmp_table,
        job_config=bigquery.LoadJobConfig(
            write_disposition="WRITE_TRUNCATE",
            schema=[
                bigquery.SchemaField("row_id", "STRING"),
                bigquery.SchemaField("adres", "STRING"),
                bigquery.SchemaField("miasto", "STRING"),
                bigquery.SchemaField("lat", "FLOAT64"),
                bigquery.SchemaField("lng", "FLOAT64"),
                bigquery.SchemaField("formatted_address", "STRING"),
                bigquery.SchemaField("place_id", "STRING"),
                bigquery.SchemaField("source", "STRING"),
                bigquery.SchemaField("geocode_ts", "TIMESTAMP"),
            ],
        ),
    )
    job.result()

    merge_sql = f"""
MERGE `{{SRC_TABLE}}` AS T
```

```

USING `tmp_table` AS G
ON COALESCE(
    CAST(T.offer_id AS STRING),
    CAST(FARM_FINGERPRINT(
        CONCAT(IFNULL(T.adres,""), IFNULL(T.miasto,""), IFNULL(CAST(T.cena AS
STRING),"")))
    AS STRING)
) = G.row_id
AND LOWER(T.miasto) IN ('poznan','poznań')
WHEN MATCHED AND G.lat IS NOT NULL AND G.lng IS NOT NULL THEN
UPDATE SET
T.lat      = G.lat,
T.lng      = G.lng,
T.formatted_address = G.formatted_address,
T.place_id     = G.place_id,
T.geocode_source = G.source,
T.geocode_ts    = G.geocode_ts,
T.geo_point     = ST_GEOPOINT(G.lng, G.lat)
"""

bq.query(merge_sql).result()
bq.delete_table(tmp_table, not_found_ok=True)

```

```

# 2) Pętla geokodowania + MERGE w paczkach
buf, ok, nores = [], 0, 0
for i, row in df_unique.iterrows():
    addr, city, arr_key = row["adres"], row["miasto"], row["adres_array_key"]
    g = geocode_with_fallback(addr, city, arr_key)

    for rid in key_map.get((addr, city, arr_key), []):
        buf.append({
            "row_id": str(rid),
            "adres": addr,
            "miasto": city,
            "lat": g["lat"], "lng": g["lng"],
            "formatted_address": g["formatted_address"],

```

```
        "place_id": g["place_id"],
        "source": g["source"],
        "geocode_ts": g["geocode_ts"],
    })

ok += int(g["lat"] is not None)
nores += int(g["lat"] is None)

if len(buf) >= FLUSH_EVERY:
    flush_chunk(pd.DataFrame(buf)); buf = []
    print(f"Checkpoint {i+1}/{len(df_unique)} | OK={ok} / NORES={nores}")

if buf:
    flush_chunk(pd.DataFrame(buf))
    print(f"Final flush | OK={ok} / NORES={nores}")

# 3) Podsumowanie
summary = bq.query(f"""
SELECT
    COUNT(*) AS wiersze_pozn,
    COUNTIF(lat IS NOT NULL AND lng IS NOT NULL) AS z_lat_lng,
    COUNTIF(geo_point IS NOT NULL) AS z_geo_point
FROM `{{SRC_TABLE}}`
WHERE LOWER(miasto) IN ('poznan','poznań')
""").result().to_dataframe()
print(summary)
```

Weryfikacja geokodowania

```
from google.cloud import bigquery
import pandas as pd

# Klient BigQuery
bq = bigquery.Client(project="realstate-market-poland")

SQLS = {
    "poz_nan_counts": """
SELECT
    COUNTIF(LOWER(miasto) IN ('poznan','poznań')) AS rows_pozn,
    COUNTIF(LOWER(miasto) IN ('poznan','poznań') AND lat IS NOT NULL AND lng IS NOT
NULL) AS rows_with_latlng,
    COUNTIF(LOWER(miasto) IN ('poznan','poznań') AND geo_point IS NOT NULL) AS
rows_with_geom
FROM `realstate-market-poland.prod.offers_sample_per_city` ;
    """,
    "unique_keys_missing": """
WITH prep AS (
    SELECT
        adres,
        miasto,
        ARRAY_TO_STRING(
            ARRAY(
                SELECT TRIM(CAST(x AS STRING))
                FROM UNNEST(IFNULL(adres_array, [])) AS x
                WHERE TRIM(CAST(x AS STRING)) <> ""
            ), ' | '
        ) AS adres_array_key
    FROM `realstate-market-poland.prod.offers_sample_per_city`
    WHERE (lat IS NULL OR lng IS NULL)
        AND adres IS NOT NULL AND TRIM(adres) <> ""
        AND LOWER(miasto) IN ('poznan','poznań')
    )
    """}
    """
```

```
),
uniq AS (
    SELECT adres, miasto, adres_array_key
    FROM prep
    GROUP BY adres, miasto, adres_array_key
)
SELECT
    (SELECT COUNT(*) FROM prep) AS rows_to_geocode,
    (SELECT COUNT(*) FROM uniq) AS unique_keys;
"""

,"tiles_stats": """
SELECT
    COUNT(*) AS tiles_ge3,
    ROUND(MIN(mediana_m2), 0) AS mediana_min,
    ROUND(APPROX_QUANTILES(mediana_m2,101)[OFFSET(50)], 0) AS mediana_med,
    ROUND(AVG(mediana_m2), 0) AS mediana_mean,
    ROUND(MAX(mediana_m2), 0) AS mediana_max,
    ROUND(MIN(sr_m2), 0) AS sr_min,
    ROUND(APPROX_QUANTILES(sr_m2,101)[OFFSET(50)], 0) AS sr_med,
    ROUND(AVG(sr_m2), 0) AS sr_mean,
    ROUND(MAX(sr_m2), 0) AS sr_max
FROM `realstate-market-poland.analytics.poznan_m2_map_tiles_100m`
WHERE liczba_ofert >= 3;
"""

,"tiles_top10": """
SELECT lat_bin, lng_bin, liczba_ofert, mediana_m2, sr_m2
FROM `realstate-market-poland.analytics.poznan_m2_map_tiles_100m`
ORDER BY liczba_ofert DESC
LIMIT 10;
"""

,"stability": """
WITH src AS (

```

```

SELECT lat, Ing,
       COALESCE(cena_za_m2, SAFE_DIVIDE(cena, NULLIF(metraz_m2,0))) AS cena_m2
  FROM `realstate-market-poland.prod.offers_sample_per_city`
 WHERE lat IS NOT NULL AND Ing IS NOT NULL
   AND LOWER(miasto) IN ('poznan','poznań')
   AND (cena_za_m2 IS NOT NULL OR (cena IS NOT NULL AND metraz_m2 > 0))
),
b1 AS (
  SELECT ROUND(lat,3) AS lat1, ROUND(Ing,3) AS Ing1,
         APPROX_QUANTILES(cena_m2,101)[OFFSET(50)] AS med_b1
    FROM src GROUP BY lat1, Ing1
),
b2 AS (
  SELECT ROUND(lat/0.0005)*0.0005 AS lat2,
         ROUND(Ing/0.0005)*0.0005 AS Ing2,
         cena_m2
    FROM src
),
b2_parent AS (
  SELECT ROUND(lat2,3) AS lat1, ROUND(Ing2,3) AS Ing1,
         APPROX_QUANTILES(cena_m2,101)[OFFSET(50)] AS med_b2
    FROM b2 GROUP BY lat1, Ing1
)
SELECT
  COUNT(*) AS matched_tiles,
  AVG(ABS(b1.med_b1 - b2_parent.med_b2)) AS delta_med_avg,
  MAX(ABS(b1.med_b1 - b2_parent.med_b2)) AS delta_med_max
  FROM b1 JOIN b2_parent USING (lat1, Ing1);
"""
,
```

"elapsed": """

```

SELECT
  MIN(geocode_ts) AS first_ts,
  MAX(geocode_ts) AS last_ts,
  TIMESTAMP_DIFF(MAX(geocode_ts), MIN(geocode_ts), SECOND) AS elapsed_s
```

```
FROM `realstate-market-poland.prod.offers_sample_per_city`  
WHERE LOWER(miasto) IN ('poznan','poznań')  
    AND geocode_ts IS NOT NULL  
-- AND geocode_ts >= TIMESTAMP('2025-09-05 00:00:00+00') -- opcjonalnie: tylko  
ostatni run  
;  
....  
}
```

```
def run(sql, location="europe-west1"):  
  
    job = bq.query(sql, location=location)  
    return job.result().to_dataframe()  
  
# Uruchom wszystkie zapytania  
summary = { name: run(sql) for name, sql in SQLS.items() }  
  
print("==== Poznań: pokrycie lat/Ing/geom ====")  
print(summary["poz_nan_counts"])  
  
print("\n==== Unikatowe klucze do geokodowania (braki) ====")  
print(summary["unique_keys_missing"])  
  
print("\n==== Kafle (n>=3): statystyki ceny m2 ====")  
print(summary["tiles_stats"])  
  
print("\n==== TOP 10 najczęstszych kafli ====")  
print(summary["tiles_top10"])  
  
print("\n==== Stabilność mediany (0.001° vs 0.0005°) ====")  
print(summary["stability"])  
  
print("\n==== Szacowany czas przebiegu wg geocode_ts ====")  
print(summary["elapsed"])
```