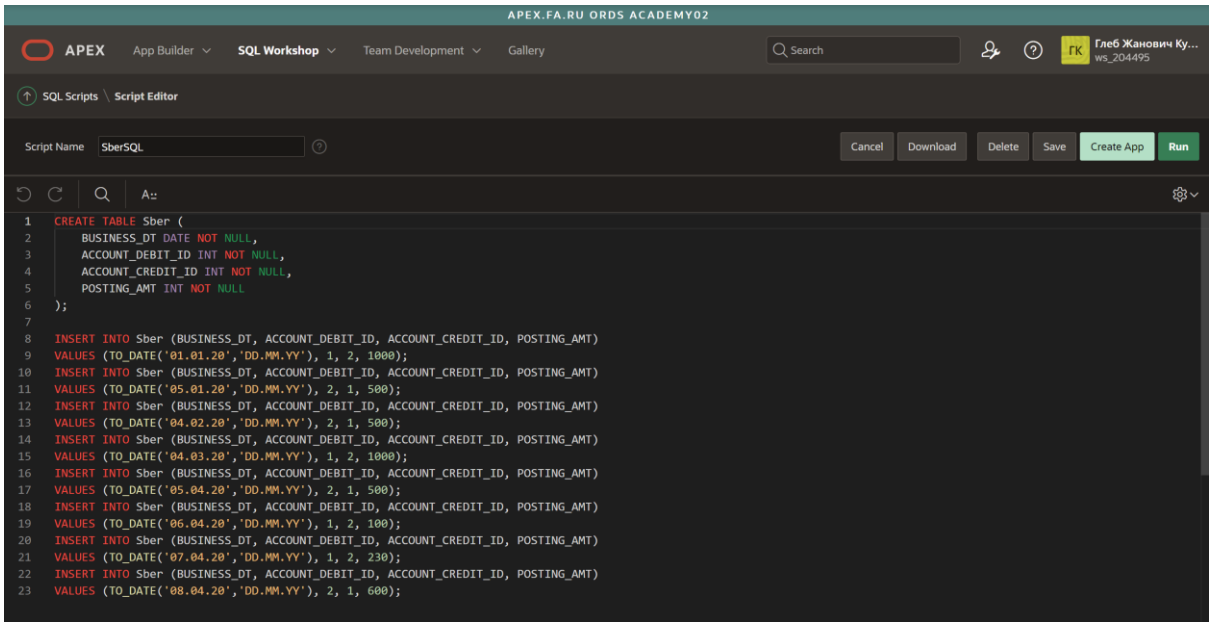
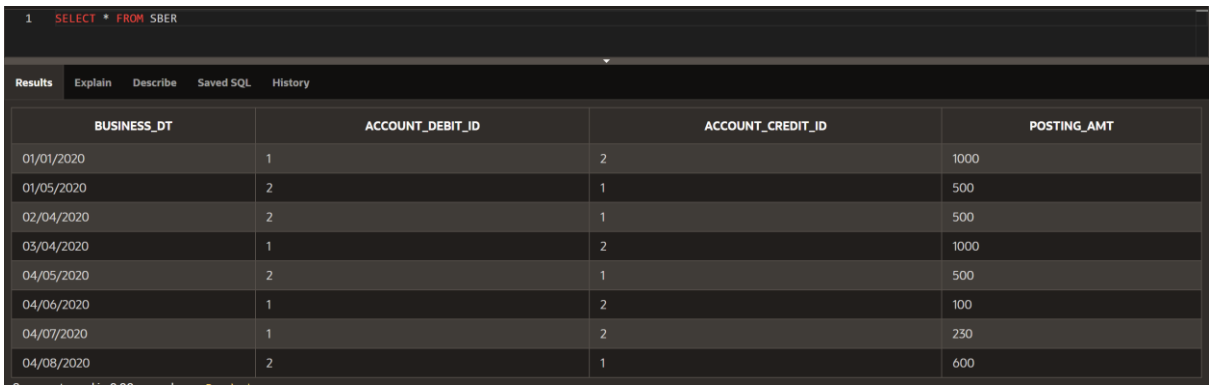
Задание по SQL. Остатки по счетам.

Дано

Имеется таблица проводок по счетам, отражающая факты начисления или списания денежных средств по счетам клиента. Состоит из столбцов:

- BUSINESS_DT - дата проводки
- ACCOUNT_DEBIT_ID - идентификатор счета дебита, с которого списываются средства
- ACCOUNT_CREDIT_ID - идентификатор счета кредита, на который начисляются средства
- POSTING_AMT - сумма проводки в рублях. Т.е. сумма средств, которая списывается со счета дебита и зачисляется на счет кредита. Значение всегда положительное.





Задание 1

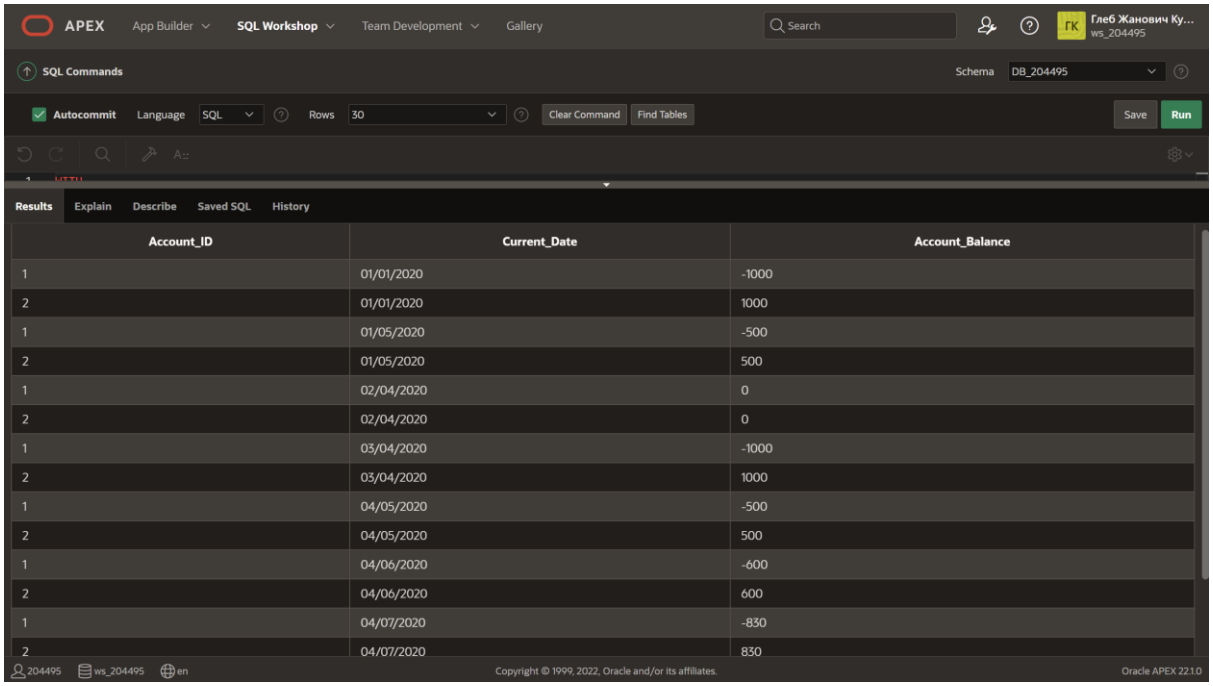
Написать SQL-запрос, который на основании таблицы проводок рассчитает остатки по счетам на текущую дату. Начальные остатки по всем счетам полагаем равными нулю. Таблица должна состоять из следующих столбцов:

- Account_ID – идентификатор счета
- Current_Date – дата, на которую посчитан остаток
- Account_Balance – остаток на счете, может быть отрицательным.

SQL-Запрос с комментариями:

```sql
WITH

-- SBER_DEBIT - это таблица, которая хранит в себе накопительную сумму проводок по дебиту счёта
SBER_DEBIT AS
(SELECT
    ACCOUNT_DEBIT_ID,
    BUSINESS_DT,
    SUM(POSTING_AMT) OVER (PARTITION BY -- Для накопительной суммы используется оконная функция SUM с группировкой по счетам и сортировкой по датам
                                ACCOUNT_DEBIT_ID
                           ORDER BY
                                BUSINESS_DT) AS SUM_POST
FROM
    SBER),



-- SBER_CREDIT - это таблица, которая хранит в себе накопительную сумму проводок по кредиту счёта
SBER_CREDIT AS
(SELECT
    ACCOUNT_CREDIT_ID,
    BUSINESS_DT,
    SUM(POSTING_AMT) OVER (PARTITION BY -- Для накопительной суммы используется оконная функция SUM с группировкой по счетам и сортировкой по датам
                                ACCOUNT_CREDIT_ID
                           ORDER BY
                                BUSINESS_DT) AS SUM_POST
FROM
    SBER),



-- ACCOUNTS - это множество всех используемых счетов
ACCOUNTS AS
(SELECT
    DISTINCT ACCOUNT_DEBIT_ID AS ACCOUNT_ID
FROM
    SBER
UNION -- Используется UNION, так как существует возможность, что счёт использовался только по дебиту или только по кредиту
SELECT
    DISTINCT ACCOUNT_CREDIT_ID AS ACCOUNT_ID
FROM
    SBER),


-- DATES - это множество уникальных дат, в которые были совершены проводки по различным счетам
DATES AS
(SELECT
    DISTINCT BUSINESS_DT AS BUSINESS_DATE
FROM
    SBER)

SELECT
    ACCOUNT_ID AS "Account_ID",
    BUSINESS_DATE AS "Current_Date",
    -- Баланс счёта рассчитывается как разница между накопительными суммами проводок по кредиту и дебиту
    -- Подзапрос ищет накопительную сумму проводок по кредиту для счета ACCOUNT_ID на дату, соответствующую BUSINESS_DATE
    -- (наиболее близкую к BUSINESS_DATE, но не превосходящую её)
    -- Если на текущую дату нет никакой информации о счёте, то по умолчанию используется ноль
    NVL((SELECT
            SUM_POST
        FROM
            SBER_CREDIT
        -- Отбор строки по номеру счёта и по непревосходящей дате
        WHERE
            BUSINESS_DT <= BUSINESS_DATE
        AND
            ACCOUNT_CREDIT_ID = ACCOUNT_ID
        -- Отбор строки с наиболее близкой к BUSINESS_DATE дате
        ORDER BY
            BUSINESS_DT DESC
        FETCH FIRST 1 ROWS ONLY), 0)
    -
    -- Подзапрос ищет накопительную сумму проводок по дебиту для счета ACCOUNT_ID на дату, соответствующую BUSINESS_DATE
    -- (наиболее близкую к BUSINESS_DATE, но не превосходящую её)
    -- Если на текущую дату нет никакой информации о счёте, то по умолчанию используется ноль
    NVL((SELECT
            SUM_POST
        FROM
            SBER_DEBIT
        -- Отбор строки по номеру счёта и по непревосходящей дате
        WHERE
            BUSINESS_DT <= BUSINESS_DATE
        AND
            ACCOUNT_DEBIT_ID = ACCOUNTS.ACCOUNT_ID
        -- Отбор строки с наиболее близкой к BUSINESS_DATE дате
        ORDER BY
            BUSINESS_DT DESC
        FETCH FIRST 1 ROWS ONLY), 0) AS "Account_Balance"
FROM
    ACCOUNTS, DATES
ORDER BY
    BUSINESS_DATE, ACCOUNT_ID
```

Вывод после выполнения SQL-запроса:



Задание 2

Написать SQL-запрос, который выводит таблицу остатков за весь период, за который нам известны проводки (например, с января по март). Начальные остатки по всем счетам полагаем равными нулю. Таблица должна состоять из следующих столбцов:

- Account_ID – идентификатор счета
- Business_From_DT – дата, начиная с которой (включительно) сформировался указанный остаток на счете.
- Business_To_DT – дата, по которую (включительно) на счете находится указанный остаток. Если остаток действует по текущий момент (дата следующего изменения остатка не известна), то поле следует заполнить датой 9999-12-31.
- Account_Balance – остаток на счете Account_ID в период с Business_From_DT по Business_To_DT включительно.

SQL-Запрос с комментариями:

```sql
WITH

-- SBER_DEBIT - это таблица, которая хранит в себе накопительную сумму проводок по дебиту счёта
SBER_DEBIT AS
(SELECT
    ACCOUNT_DEBIT_ID,
    BUSINESS_DT,
    SUM(POSTING_AMT) OVER (PARTITION BY -- Для накопительной суммы используется оконная функция SUM с группировкой по счетам и сортировкой по датам
                                ACCOUNT_DEBIT_ID
                           ORDER BY
                                BUSINESS_DT) AS SUM_POST
FROM
    SBER),


-- SBER_CREDIT - это таблица, которая хранит в себе накопительную сумму проводок по кредиту счёта
SBER_CREDIT AS
(SELECT
    ACCOUNT_CREDIT_ID,
    BUSINESS_DT,
    SUM(POSTING_AMT) OVER (PARTITION BY -- Для накопительной суммы используется оконная функция SUM с группировкой по счетам и сортировкой по датам
                                ACCOUNT_CREDIT_ID
                           ORDER BY
                                BUSINESS_DT) AS SUM_POST
FROM
    SBER),


-- DATES - это объединение множества уникальных дат и всех счетов
DATES AS
(SELECT
    DISTINCT ACCOUNT_DEBIT_ID AS ACCOUNT_ID, BUSINESS_DT
FROM
    SBER_DEBIT
UNION -- Используется UNION, так как существует возможность, что счёт использовался только по дебиту или только по кредиту
SELECT
    DISTINCT ACCOUNT_CREDIT_ID AS ACCOUNT_ID, BUSINESS_DT
FROM
    SBER_CREDIT)


SELECT
    DT.ACCOUNT_ID AS "Account_ID",
    DT.BUSINESS_DT AS "Business_From_DT",
    -- Business_To_DT - это дата, по которую (включительно) на счете находится указанный остаток
    -- Подзапрос выбирает следущую дату после изменения счёта, отнимает от нее один день и при помощи отбора подставляет в нужную строку Business_To_DT
    -- Если остаток действует по текущий момент (дата следующего изменения остатка не известна), то по умолчанию в Business_To_DT вставляется дата 31-12-9999
    NVL((SELECT
            DATES.BUSINESS_DT - NUMTODSINTERVAL(1, 'day')
        FROM
            DATES
        -- Отбор строки по номеру счёта и по превосходящей дате
        WHERE
            DATES.BUSINESS_DT > DT.BUSINESS_DT
        AND
            DATES.ACCOUNT_ID = DT.ACCOUNT_ID
        -- Отбор строки с наиболее близкой к BUSINESS_DT дате
        ORDER BY
            BUSINESS_DT
        FETCH FIRST 1 ROWS ONLY), TO_DATE('9999-12-31','YYYY-MM-DD')) AS "Business_To_DT" ,

    -- Баланс счёта рассчитывается как разница между накопительными суммами проводок по кредиту и дебиту

    -- Подзапрос ищет накопительную сумму проводок по кредиту для счета ACCOUNT_ID на дату, соответствующую BUSINESS_DT
    -- (наиболее близкую к BUSINESS_DT, но не превосходящую её)
    -- Если на текущую дату нет никакой информации о счёте, то по умолчанию используется ноль
    NVL((SELECT
            SBER_CREDIT.SUM_POST
        FROM
            SBER_CREDIT
        -- Отбор строки по номеру счёта и по непревосходящей дате
        WHERE
            SBER_CREDIT.BUSINESS_DT <= DT.BUSINESS_DT
        AND
            SBER_CREDIT.ACCOUNT_CREDIT_ID = DT.ACCOUNT_ID
        -- Отбор строки с наиболее близкой к BUSINESS_DT дате
        ORDER BY
            SBER_CREDIT.BUSINESS_DT DESC
        FETCH FIRST 1 ROWS ONLY), 0)
    -
    -- Подзапрос ищет накопительную сумму проводок по дебиту для счета ACCOUNT_ID на дату, соответствующую BUSINESS_DT
    -- (наиболее близкую к BUSINESS_DT, но не превосходящую её)
    -- Если на текущую дату нет никакой информации о счёте, то по умолчанию используется ноль
    NVL((SELECT
            SBER_DEBIT.SUM_POST
        FROM
            SBER_DEBIT
        WHERE
        -- Отбор строки по номеру счёта и по непревосходящей дате
            SBER_DEBIT.BUSINESS_DT <= DT.BUSINESS_DT
        AND
            SBER_DEBIT.ACCOUNT_DEBIT_ID = DT.ACCOUNT_ID
        -- Отбор строки с наиболее близкой к BUSINESS_DT дате
        ORDER BY
            SBER_DEBIT.BUSINESS_DT DESC
        FETCH FIRST 1 ROWS ONLY), 0) AS "Account_Balance"
FROM
    DATES DT
ORDER BY
    BUSINESS_DT, ACCOUNT_ID
```

Вывод после выполнения SQL-запроса:

Account_ID	Business_From_DT	Business_To_DT	Account_Balance
1	01/01/2020	01/04/2020	-1000
2	01/01/2020	01/04/2020	1000
1	01/05/2020	02/03/2020	-500
2	01/05/2020	02/03/2020	500
1	02/04/2020	03/03/2020	0
2	02/04/2020	03/03/2020	0
1	03/04/2020	04/04/2020	-1000
2	03/04/2020	04/04/2020	1000
1	04/05/2020	04/05/2020	-500
2	04/05/2020	04/05/2020	500
1	04/06/2020	04/06/2020	-600
2	04/06/2020	04/06/2020	600
1	04/07/2020	04/07/2020	-830
2	04/07/2020	04/07/2020	830

Задание 3

Написать SQL запрос, который выведет остатки на каждый день за период с января по март. Т.е. по каждому счету таблица должна содержать около 90 записей. Начальные остатки по всем счетам полагаем равными нулю. Столбцы таблицы:

- Account_ID – идентификатор счета
- Business_Date – дата, на которую посчитан остаток
- Account_Balance – остаток на счете, может быть отрицательным. Например, по счету 2 должный быть такие записи:

2, 2020-01-01, 1000

2, 2020-01-02, 1000

2, 2020-01-03, 1000

2, 2020-01-04, 1000

2, 2020-01-05, 500

SQL-Запрос с комментариями:

```sql
WITH

-- SBER_DEBIT - это таблица, которая хранит в себе накопительную сумму проводок по дебиту счёта
SBER_DEBIT AS
(SELECT
    ACCOUNT_DEBIT_ID,
    BUSINESS_DT,
    SUM(POSTING_AMT) OVER (PARTITION BY -- Для накопительной суммы используется оконная функция SUM с группировкой по счетам и сортировкой по датам
                                ACCOUNT_DEBIT_ID
                           ORDER BY
                                BUSINESS_DT) AS SUM_POST
FROM
    SBER),



-- SBER_CREDIT - это таблица, которая хранит в себе накопительную сумму проводок по кредиту счёта
SBER_CREDIT AS
(SELECT
    ACCOUNT_CREDIT_ID,
    BUSINESS_DT,
    SUM(POSTING_AMT) OVER (PARTITION BY -- Для накопительной суммы используется оконная функция SUM с группировкой по счетам и сортировкой по датам
                                ACCOUNT_CREDIT_ID
                           ORDER BY
                                BUSINESS_DT) AS SUM_POST
FROM
    SBER),



-- ACCOUNTS - это множество всех используемых счетов
ACCOUNTS AS
(SELECT
    DISTINCT ACCOUNT_DEBIT_ID AS ACCOUNT_ID
FROM
    SBER
UNION -- Используется UNION, так как существует возможность, что счёт использовался только по дебиту или только по кредиту
SELECT
    DISTINCT ACCOUNT_CREDIT_ID AS ACCOUNT_ID
FROM
    SBER),


-- DATE_RANGE - это таблица, которая хранит в себе строго ограниченный набор дат (в данном случае с января по март)
-- Чтобы получить нужное количество дней, первая дата (2020-01-01) вычитается из последней (2020-03-31)
-- При помощи оператора LEVEL создается полученное количество дат
-- В запросе присутствует вычитание (- 1) и сложение (+ 1), с помощью данных операций в диапазон дат включаются даты начала и окончания периода
-- (2020-01-01 и 2020-03-31 соответственно)
DATE_RANGE AS
(SELECT
    TO_DATE('2020-01-01', 'YYYY-MM-DD') + LEVEL - 1 AS BUSINESS_DATE
FROM
    DUAL
CONNECT BY LEVEL <= (
TO_DATE('2020-03-31', 'YYYY-MM-DD') - TO_DATE('2020-01-01', 'YYYY-MM-DD') + 1
))


SELECT
    ACCOUNT_ID as "Account_ID",
    BUSINESS_DATE AS "Business_Date",
    -- Баланс счёта рассчитывается как разница между накопительными суммами проводок по кредиту и дебиту

    -- Подзапрос ищет накопительную сумму проводок по кредиту для счета ACCOUNT_ID на дату, соответствующую BUSINESS_DATE
    -- (наиболее близкую к BUSINESS_DATE, но не превосходящую её)
    -- Если на текущую дату нет никакой информации о счёте, то по умолчанию используется ноль
    NVL((SELECT
            SUM_POST
        FROM
            SBER_CREDIT
        -- Отбор строки по номеру счёта и по непревосходящей дате
        WHERE
            BUSINESS_DT <= BUSINESS_DATE
        AND
            ACCOUNT_CREDIT_ID = ACCOUNT_ID
        -- Отбор строки с наиболее близкой к BUSINESS_DATE дате
        ORDER BY
            BUSINESS_DT DESC
        FETCH FIRST 1 ROWS ONLY), 0)
    -
    -- Подзапрос ищет накопительную сумму проводок по дебиту для счета ACCOUNT_ID на дату, соответствующую BUSINESS_DATE
    -- (наиболее близкую к BUSINESS_DATE, но не превосходящую её)
    -- Если на текущую дату нет никакой информации о счёте, то по умолчанию используется ноль
    NVL((SELECT
            SUM_POST
        FROM
            SBER_DEBIT
        -- Отбор строки по номеру счёта и по непревосходящей дате
        WHERE
            BUSINESS_DT <= BUSINESS_DATE
        AND
            ACCOUNT_DEBIT_ID = ACCOUNTS.ACCOUNT_ID
        -- Отбор строки с наиболее близкой к BUSINESS_DATE дате
        ORDER BY
            BUSINESS_DT DESC
        FETCH FIRST 1 ROWS ONLY), 0) AS "Account_Balance"
FROM
    ACCOUNTS, DATE_RANGE
ORDER BY
    ACCOUNT_ID, BUSINESS_DATE
```

Вывод после выполнения SQL-запроса:

Results | Explain | Describe | Saved SQL | History

1	03/27/2020	-1000
1	03/28/2020	-1000
1	03/29/2020	-1000
1	03/30/2020	-1000
1	03/31/2020	-1000
2	01/01/2020	1000
2	01/02/2020	1000
2	01/03/2020	1000
2	01/04/2020	1000
2	01/05/2020	500
2	01/06/2020	500
2	01/07/2020	500
2	01/08/2020	500
2	01/09/2020	500
2	01/10/2020	500

Задание по Python

Задание предлагается сделать, используя язык Python3

На входе есть таблица отделений банка (кол-во записей 100 – 10 000) с полями:

- DIV_ID (ID отделения),
- CUR_CAPACITY (целое – текущее кол-во клиентов в отделении),
- MAX_CAPACITY (целое – максимальное кол-во клиентов в отделении)

И таблица с клиентами (кол-во записей 1000 – 1 000 000), которым нужно рекомендовать отделение для посещения, с полями:

- CLIENT_ID (ID клиента),
- DIV1_ID (самое предпочтительное отделение для клиента),
- DIV2_ID (второе предпочтительное отделение для клиента),
- DIV3_ID (третье предпочтительное отделение для клиента)

Результатом работы программы должна быть таблица CLIENT_ID, DIV_ID в которой мы для каждого клиента рекомендуем одно из отделений DIV1_ID, DIV2_ID, DIV3_ID

При этом новая CUR_CAPACITY для каждого отделения (считаем, что каждый клиент согласился на рекомендацию) не должна превышать MAX_CAPACITY. (в качестве входных и выходных таблиц можно использовать Pandas DataFrame или иную подходящую структуру данных)

BANK:

	A	B	C
1	DIV_ID	CUR_CAPACITY	MAX_CAPACITY
2	1	4903	5000
3	2	2605	5000
4	3	1249	5000
5	4	2380	5000
6	5	4806	5000
7	6	1423	5000
8	7	1176	5000
9	8	4928	5000
10	9	1603	5000
11	10	1122	5000
12	11	4344	5000
13	12	4019	5000
14	13	1710	5000
15	14	2373	5000

KLI:

	A	B	C	D
1	CLIENT_ID	DIV1_ID	DIV2_ID	DIV3_ID
2	1	10	41	38
3	2	65	96	65
4	3	1	46	35
5	4	95	33	69
6	5	71	64	91
7	6	80	34	36
8	7	96	34	36
9	8	46	30	52
10	9	87	94	25
11	10	74	8	34
12	11	56	51	96
13	12	86	23	90
14	13	65	75	83

В BANK 100 записей

В KLI 1000 записей

Программа с комментариями:

Подготовка среды

```
import numpy as np
import pandas as pd
```

[1] ✓ 3.2s Python

Считывание входных данных, их обработка и создание DataFrame для итогового распределения

```
# Загрузка данных клиентов, установка ID клиента в качестве индекса
clients = pd.read_excel("./KLI.xlsx").set_index('CLIENT_ID')

# Загрузка данных отделений, установка ID отделения в качестве индекса
banks = pd.read_excel("./BANK.xlsx").set_index('DIV_ID')

# Создание вспомогательного столбца FREE_CAPACITY, который соответствует количеству свободных мест в отделении
banks['FREE_CAPACITY'] = banks['MAX_CAPACITY'] - banks['CUR_CAPACITY']

# Создание DataFrame для итогового распределния клиентов по отделениям
# В качестве индекса устонавливается ID клиента
distribution = pd.DataFrame(columns=['CLIENT_ID', 'DIV_ID'], dtype='int64').set_index('CLIENT_ID')
```

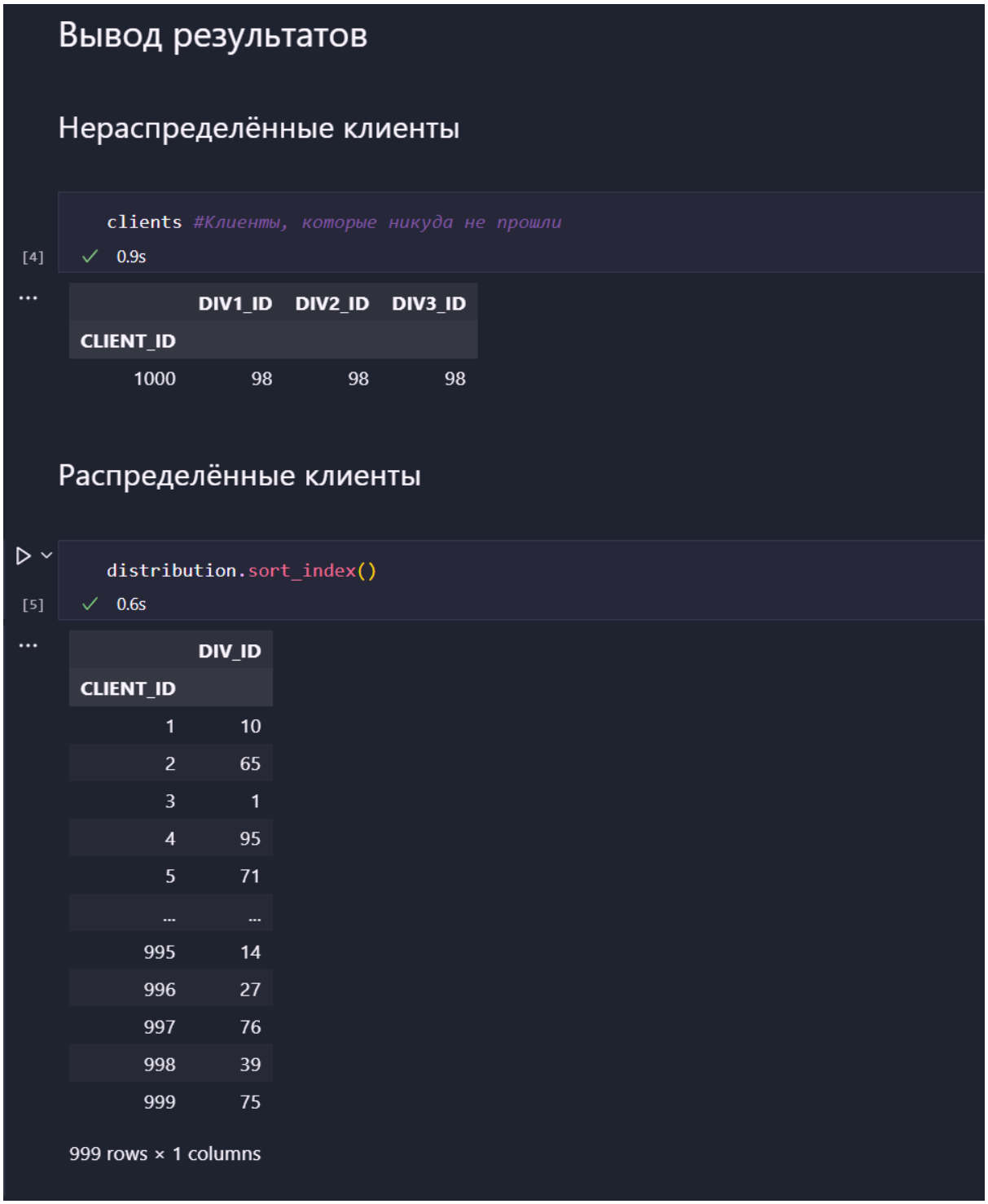
[2] ✓ 0.7s Python

Распределение клиентов по их приоритетным отделениям

```
# Алгоритм заключается в следующем:
# Каждый клиент распределяется в свое первое по приоритету отделение
# Отбираются те клиенты, которым не хватило места в своем приоритетном отделении
# Далее эти клиенты распределяются в свое второе по приоритету отеделение
# Если и в этот раз для некоторых из клиентов не хватило места в этих отделениях
# Они распределяются в свое третье по приоритету отеделение
for div in ['DIV1_ID', 'DIV2_ID', 'DIV3_ID']:

    # grouped_clients - это клиенты сгруппированные по ID текущих приоритетных направлений (DIV1_ID, DIV2_ID, DIV3_ID),
    # которым присвоены номера в своей группе
    grouped_clients = clients.groupby(div).cumcount().to_frame().rename(columns={0: 'NUMBER_IN_DIV'})

    # div_capacities - количество свободных мест в каждом из отделений (используется для маски)
    div_capacities = banks.loc[clients.loc[grouped_clients.index.values][div]]['FREE_CAPACITY'].values

    # numbers_in_divs - это порядковые номера клиентов в своем приоритетном отделении (используется для маски)
    numbers_in_divs = grouped_clients['NUMBER_IN_DIV'].values

    # Отбор тех клиентов, для которых нашлись свободные места в отделениях
    grouped_clients = grouped_clients[numbers_in_divs < div_capacities]

    # Создается новый столбец, который предстовляет собой ID отделения, в которое попал клиент
    grouped_clients = pd.merge(grouped_clients, clients.rename(columns={div: 'DIV_ID'})[
        ['DIV_ID']], left_index=True, right_index=True)

    # Учёт изменения количества свободных мест в отделениях:
    # Для изменения количества свободных мест в отделениях создается пустой DataFrame с индексами отделений, заполненный нулями
    # Из этого DataFrame вычитается количество клиентов попавших в каждое из отделений
    # Так как на этом этапе могут возникнуть NaN значения (не все отделения могли быть задействованы)
    # Эти значения заменяются нулями
    # Получившийся DataFrame представляет собой изменение количества свободных мест в каждом из отделений,
    # которые в последствии складываются со столбцом FREE_CAPACITY
    banks['FREE_CAPACITY'] += (pd.DataFrame(data=np.zeros(shape=banks.index.size),
                                            dtype='int64',
                                            index=banks.index)
                               .iloc[:, 0] - grouped_clients.groupby('DIV_ID').count().iloc[:, 0]).fillna(0)

    # Сохранение информации о клиентах, которые были распределены
    distribution = pd.concat([distribution['DIV_ID'], grouped_clients['DIV_ID']]).to_frame()

    # Удаление информации о распределённых клиентах и переход к следующим по приоритету отделениям
    # (для нераспределённых клиентов)
    clients.drop(grouped_clients.index, inplace=True)
```

[3] ✓ 0.9s

Результат работы программы:



CLIENT_ID	DIV1_ID	DIV2_ID	DIV3_ID
1000	98	98	98

CLIENT_ID	DIV_ID
1	10
2	65
3	1
4	95
5	71
...	...
995	14
996	27
997	76
998	39
999	75