Тема 22. Обработка больших данных с помощью Pandas. Обработка данных с помощью Pandas на кластере PySpark.

Цель занятия:

Ознакомиться с методами и инструментами обработки больших данных с использованием библиотеки Pandas, а также с возможностями масштабирования анализа данных на кластере PySpark.

Учебные вопросы:

1. Введение в Pandas для анализа данных.
2. Работа с большими данными в Pandas.
3. Работа с Pandas на кластере PySpark

1. Введение в Pandas для анализа данных.

Pandas – это библиотека Python, предназначенная для анализа данных и их манипуляций. Она предоставляет высокоуровневые структуры данных и мощные инструменты для обработки данных в табличном формате, что делает ее одной из самых популярных библиотек для работы с данными в Python.

Pandas широко используется в различных областях – от научных исследований и финансов до машинного обучения и обработки данных в индустрии.

Основные возможности Pandas для работы с данными:

- Чтение и запись данных из различных источников: CSV, Excel, SQL, JSON, HDF5, Parquet и др.
- Фильтрация и выборка данных: гибкий выбор строк и столбцов по меткам или индексам, фильтрация на основе условий.
- Манипуляции с данными: сортировка, добавление и удаление столбцов, удаление дубликатов, обработка пропущенных данных.
- Группировка и агрегирование: группировка данных по категориям и выполнение агрегирующих операций, таких как суммирование, среднее, медиана и т. д.
- Работа с временными рядами: поддержка операций с датами и временными метками, включая ресемплирование, интервалы, частоты и временные сдвиги.
- Соединение и объединение данных: слияние и объединение данных из разных источников с использованием методов merge, concat, join.

Преимущества и ограничения Pandas при обработке больших данных

Преимущества:

- Интуитивный интерфейс: Pandas упрощает работу с данными благодаря удобному API, понятному даже для начинающих пользователей.
- Универсальность: библиотека поддерживает широкий спектр операций, включая статистический анализ, визуализацию, агрегирование и объединение данных.
- Совместимость: Pandas интегрируется с другими библиотеками Python для анализа данных, машинного обучения и визуализации, такими как NumPy, Matplotlib и Scikit-Learn.

Ограничения:

- Работа в оперативной памяти: Pandas загружает данные целиком в память, поэтому при работе с очень большими данными (обычно свыше 1-2 ГБ) возникают проблемы с производительностью и ограничением памяти.
- Однопоточные вычисления: большинство операций в Pandas выполняются в одном потоке, что ограничивает эффективность на многопроцессорных системах.
- Сложность масштабирования: Pandas не поддерживает распределенные вычисления, поэтому для обработки больших данных приходится использовать другие инструменты, такие как Dask или PySpark.

Pandas основывается на двух основных структурах данных: DataFrame и Series.

DataFrame:

- Двумерная табличная структура, где данные организованы в строки и столбцы.
- Столбцы могут содержать разные типы данных (числовые, строки, даты и т. д.).
- DataFrame поддерживает индексирование по строкам и столбцам, что позволяет быстро выполнять операции с данными.

Series:

- Одномерная структура данных, представляющая собой список значений с метками.
- Series можно рассматривать как столбец в таблице DataFrame или как словарь с индексами, что позволяет удобно манипулировать данными.
- Используется как основа для работы с отдельными столбцами или временными рядами.

Эти структуры данных обеспечивают гибкие и мощные возможности для анализа и обработки данных в Pandas, делая их ключевыми элементами при работе с этой библиотекой.

Для работы с библиотекой Pandas, сначала нужно установить ее на ваш компьютер. Установка Pandas — это простой процесс, так как библиотека доступна через менеджер пакетов pip, стандартный инструмент Python для установки и управления пакетами.

Установка Pandas с помощью pip:

```
pip install pandas
```

Если вы используете Jupyter Notebook для анализа данных, можно установить Pandas прямо в нем, выполнив ячейку кода:

```
In [1]: !pip install pandas

        Requirement already satisfied: pandas in /opt/conda/lib/python3.7/site-packages (1.0.5)
        Requirement already satisfied: pytz>=2017.2 in /opt/conda/lib/python3.7/site-packages (from pandas) (2020.1)
        Requirement already satisfied: python-dateutil>=2.6.1 in /opt/conda/lib/python3.7/site-packages (from pandas) (2.8.1)
        Requirement already satisfied: numpy>=1.13.3 in /opt/conda/lib/python3.7/site-packages (from pandas) (1.18.5)
        Requirement already satisfied: six>=1.5 in /opt/conda/lib/python3.7/site-packages (from python-dateutil>=2.6.1->pandas) (1.15.
        0)
```

Чтобы убедиться, что Pandas успешно установился, можно открыть Python интерактивную оболочку и попробовать импортировать библиотеку:

```
In [3]: import pandas as pd
```

Если ошибок нет, установка прошла успешно. Вы также можете проверить версию Pandas, введя:

```
In [4]: print(pd.__version__)

        1.0.5
```

2. Работа с большими данными в Pandas.

Pandas позволяет загружать и обрабатывать данные из различных источников, таких как CSV, Excel, SQL, JSON и др. Для работы с большими наборами данных, которые не помещаются в оперативную память, можно использовать параметр chunksize.

Он позволяет загружать данные по частям и обрабатывать их итеративно, снижая нагрузку на память.

Загрузка и чтение данных

Pandas предоставляет удобные методы для чтения данных:

pd.read_csv() – для чтения CSV-файлов;

pd.read_excel() – для чтения данных из Excel;

pd.read_sql() – для загрузки данных из SQL базы данных;

pd.read_json() – для работы с JSON-файлами.

Загрузка и чтение данных по частям

Для чтения больших файлов с помощью этих функций можно воспользоваться параметром **chunksize**.

Параметр chunksize позволяет считывать файл по частям, каждая из которых представляет собой отдельный DataFrame. Установив chunksize, можно задать количество строк, которое будет загружаться за один раз. Это особенно полезно при анализе больших наборов данных, так как обрабатываемый объем данных уменьшается, что предотвращает превышение объема оперативной памяти.

Рассмотрим пример, где мы читаем CSV-файл с 1 миллионом строк и обрабатываем его по 100 000 строк за раз:

```
import pandas as pd

# Указываем путь к большому CSV-файлу
file_path = 'large_dataset.csv'

# Устанавливаем размер чанка
#(количество строк, которое будет загружено за раз)
chunk_size = 100000

# Итеративно обрабатываем данные
for chunk in pd.read_csv(file_path, chunksize=chunk_size):
    # Пример операции: подсчет среднего значения в каждом чанк-фрейме
    mean_values = chunk.mean()
    print(mean_values)
```

Обработка данных по частям с накоплением результатов

Иногда требуется сохранить результат обработки каждого чанка для дальнейшего использования, например, при агрегации данных.

В этом случае можно использовать список или словарь для накопления результатов и объединить их после обработки всех чанков.

```python
import pandas as pd

file_path = 'large_dataset.csv'
chunk_size = 100000
results = []

for chunk in pd.read_csv(file_path, chunksize=chunk_size):
    # Выполняем нужные операции с каждым чанк-фреймом
    # Например, суммируем значения в каждом чанке
    result = chunk['column_name'].sum()
    results.append(result)

# Суммируем результаты из всех чанков
total_sum = sum(results)
print("Сумма значений во всех чанках:", total_sum)
```

Преимущества использования chunksize

- Снижение нагрузки на память: данные загружаются по частям, что позволяет эффективно использовать оперативную память.
- Гибкость обработки: можно выполнять различные операции для каждой части данных.
- Увеличение скорости обработки: несмотря на итерации, общий процесс может стать быстрее за счет уменьшения объема данных, обрабатываемых одновременно.

Примеры с chunksize

Фильтрация данных по частям: например, загрузка строк, где значение в определенном столбце превышает заданный порог.

```
for chunk in pd.read_csv(file_path, chunksize=chunk_size):
    filtered_chunk = chunk[chunk['column_name'] > threshold]
    # Обработка или сохранение filtered_chunk
```

Сохранение обработанных данных: если необходимо сохранить обработанные чанки в новый файл, можно использовать метод to_csv с параметром mode='a' (добавление).

```
for chunk in pd.read_csv(file_path, chunksize=chunk_size):
    chunk.to_csv('processed_data.csv', mode='a', header=False, index=False)
```

При работе с большими объемами данных в Pandas важно оптимизировать использование памяти.

Одним из эффективных способов является конвертация типов данных, а также использование типа **category** для строковых данных с ограниченным количеством уникальных значений.

1. Конвертация типов данных для оптимизации памяти

Pandas по умолчанию использует типы данных, которые могут занимать много места, особенно для чисел и строк. Конвертация типов позволяет сократить объем занимаемой памяти, не теряя при этом точности данных.

Оптимизация числовых типов

Целочисленные данные: по умолчанию Pandas использует тип int64, который занимает больше памяти, чем другие целочисленные типы. Если известно, что значения в столбце не превышают определенных диапазонов, можно использовать int8, int16 или int32, которые занимают меньше памяти.

Числа с плавающей запятой: для экономии памяти можно использовать float32 вместо float64, если данные допускают меньшую точность.

Пример:

```python
import pandas as pd

# Создаем пример DataFrame
df = pd.DataFrame({
    'big_int': [1000, 2000, 3000],
    'small_float': [1.5, 2.5, 3.5]
})

# Оптимизируем типы данных
df['big_int'] = df['big_int'].astype('int16')  # вместо int64
df['small_float'] = df['small_float'].astype('float32')  # вместо float64
```

Автоматическая конвертация типов

Можно использовать функцию pd.to_numeric() для автоматического выбора оптимальных типов данных, указав параметр downcast для уменьшения типа до минимального возможного:

```
df['big_int'] = pd.to_numeric(df['big_int'], downcast='integer')
df['small_float'] = pd.to_numeric(df['small_float'], downcast='float')
```

2. Использование типа category для строковых данных

Столбцы с текстовыми значениями часто занимают много памяти, особенно если в них содержится множество повторяющихся значений. Использование типа category позволяет сократить объем памяти, заменяя каждое уникальное строковое значение уникальным числовым кодом.

Когда использовать category?

category лучше всего подходит для строковых столбцов с небольшим числом уникальных значений (категорий). Примеры: столбцы с названиями городов, статусами заказов, типами продуктов и т. д.

Пример конвертации в category:

```
df = pd.DataFrame({
    'city': ['Москва', 'Санкт-Петербург', 'Москва', 'Новосибирск']
})

# Конвертация типа данных в category
df['city'] = df['city'].astype('category')
```

Проверка использования памяти до и после оптимизации

Для наглядности можно проверить использование памяти до и после конвертации типов данных с помощью метода memory_usage(deep=True):

```
# Использование памяти до конвертации
print("До оптимизации:")
print(df.memory_usage(deep=True))

# Конвертация типа данных в category
df['city'] = df['city'].astype('category')

# Использование памяти после конвертации
print("\nПосле оптимизации:")
print(df.memory_usage(deep=True))
```

Пример: Оптимизация типов данных для большого набора данных

```python
import pandas as pd

# Загружаем данные по частям и оптимизируем каждый чанк
chunk_size = 100000
file_path = 'large_dataset.csv'

for chunk in pd.read_csv(file_path, chunksize=chunk_size):
    # Оптимизируем числовые типы
    for col in chunk.select_dtypes(include=['int', 'float']).columns:
        chunk[col] = pd.to_numeric(chunk[col],
                downcast='integer' if chunk[col].dtype == 'int' else 'float')

    # Оптимизируем текстовые типы, если число уникальных значений
    # меньше 50% от общего числа строк
    for col in chunk.select_dtypes(include=['object']).columns:
        if chunk[col].nunique() / len(chunk) < 0.5:
            chunk[col] = chunk[col].astype('category')

    # Выполняем операции с оптимизированным чанк-фреймом
    # Например, сохраняем в новый файл или обрабатываем данные
```

Оптимизация типов данных в Pandas позволяет значительно сократить использование памяти, что особенно полезно при работе с большими наборами данных.

Конвертация числовых типов и использование category для строк помогают экономить память без потери точности и гибкости.

Фильтрация данных при загрузке (параметры usecols и dtype).

При загрузке больших данных в Pandas мы можем сразу же оптимизировать процесс с помощью параметров **usecols** и **dtype**, что позволяет загружать только необходимые столбцы и указывать подходящие типы данных. Это сокращает использование памяти и ускоряет обработку.

1. Параметр usecols: загрузка только нужных столбцов

Параметр usecols позволяет выбрать конкретные столбцы, которые нужно загрузить из файла, игнорируя остальные.

Это особенно полезно, когда работаем с файлами, содержащими множество столбцов, но для анализа нам нужны лишь некоторые из них.

Пример использования usecols.

Допустим, у нас есть файл с большим числом столбцов, но нам нужны только несколько ключевых для анализа:

```
import pandas as pd

# Читаем только необходимые столбцы из файла CSV
df = pd.read_csv('large_dataset.csv',
                 usecols=['column1', 'column2', 'column5'])
```

Здесь будут загружены только столбцы column1, column2 и column5. Остальные столбцы игнорируются, что позволяет сэкономить память и время загрузки.

2. Параметр dtype: указание типов данных для экономии памяти

Параметр dtype позволяет задать типы данных для столбцов при загрузке.

Это помогает сократить объем занимаемой памяти, так как Pandas не будет автоматически присваивать типы, которые могут занимать больше места.

Рассмотрим пример, в котором оптимизируем типы данных для числовых и строковых столбцов:

```
import pandas as pd

# Задаем типы данных для определенных столбцов
dtype_dict = {
    'column1': 'int32',         # целочисленный столбец
    'column2': 'float32',       # вещественный столбец
    'category_column': 'category'  # текстовый столбец, преобразуемый в category
}

# Загружаем данные с использованием типов данных
df = pd.read_csv('large_dataset.csv', dtype=dtype_dict)
```

3. Совместное использование usecols и dtype

Эти параметры можно применять одновременно, чтобы загрузить только нужные столбцы с оптимизированными типами данных. Например:

```
# Читаем только необходимые столбцы
# и задаем для них оптимальные типы данных
df = pd.read_csv('large_dataset.csv',
                 usecols=['column1', 'column2', 'category_column'],
                 dtype=dtype_dict)
```

Использование параметров usecols и dtype помогает снизить потребление памяти при загрузке данных. Ниже показано, как можно оценить влияние этих параметров на объем памяти:

```
# Загрузка данных без указания параметров
df_full = pd.read_csv('large_dataset.csv')
print("Использование памяти без оптимизации:",
      df_full.memory_usage(deep=True).sum())

# Загрузка данных с параметрами usecols и dtype
df_optimized = pd.read_csv('large_dataset.csv',
                           usecols=['column1', 'column2', 'category_column'],
                           dtype=dtype_dict)

print("Использование памяти с оптимизацией:",
      df_optimized.memory_usage(deep=True).sum())
```

Объединение и агрегирование данных для снижения объема.

Объединение и агрегирование данных в Pandas — это мощные инструменты для снижения объема данных и улучшения анализа.

Эти операции позволяют обрабатывать данные более эффективно, группируя их и применяя функции агрегации для получения обобщенной информации.

1. Объединение данных

Объединение (или слияние) данных позволяет комбинировать несколько DataFrame на основе общих столбцов. В Pandas есть несколько функций для объединения данных, наиболее популярные из которых — merge() и concat().

a. Функция merge()

Функция merge() используется для объединения двух DataFrame по одному или нескольким ключам (столбцам). Это похоже на SQL JOIN.

Пример:

```
import pandas as pd

# Создаем два DataFrame
df1 = pd.DataFrame({
    'id': [1, 2, 3],
    'name': ['Alice', 'Bob', 'Charlie']
})

df2 = pd.DataFrame({
    'id': [1, 2, 4],
    'age': [25, 30, 22]
})

# Объединяем DataFrame по столбцу 'id'
# 'inner', 'outer', 'left', 'right'
merged_df = pd.merge(df1, df2, on='id', how='inner')
print(merged_df)
```

b. Функция concat()

Функция concat() объединяет DataFrame по оси (по строкам или по столбцам).

Она полезна, когда нужно объединить данные с одинаковыми столбцами.

Пример:

```
# Создаем два DataFrame с одинаковыми столбцами
df3 = pd.DataFrame({
    'name': ['Alice', 'Bob'],
    'age': [25, 30]
})

df4 = pd.DataFrame({
    'name': ['Charlie', 'David'],
    'age': [22, 28]
})

# Объединяем DataFrame по строкам
concatenated_df = pd.concat([df3, df4], ignore_index=True)
print(concatenated_df)
```

2. Агрегирование данных

Агрегирование позволяет группировать данные и применять функции, такие как сумма, среднее, максимум и минимум. Это помогает уменьшить объем данных, сохраняя при этом важную информацию.

a. Использование groupby()

Функция groupby() используется для группировки данных по одному или нескольким столбцам, после чего можно применять агрегационные функции.

Пример:

```
# Создаем DataFrame с данными о продажах
sales_data = pd.DataFrame({
    'store': ['A', 'B', 'A', 'B', 'A'],
    'sales': [100, 200, 150, 300, 200]
})

# Группируем данные по магазину и считаем общую сумму продаж
aggregated_sales = sales_data.groupby('store')['sales'].sum().reset_index()
print(aggregated_sales)
```

3. Работа с Pandas на кластере PySpark

Работа с Pandas на кластере PySpark позволяет эффективно обрабатывать большие объемы данных, используя преимущества распределенной обработки данных и возможности библиотеки Pandas.

Ниже приведен обзор процесса подключения к кластеру Spark и настройки среды для работы с PySpark и Pandas.

1. Установка необходимых библиотек

Перед началом работы убедитесь, что у вас установлены необходимые библиотеки. Для работы с PySpark и Pandas вам потребуется:

- Apache Spark
- PySpark
- Pandas

Вы можете установить PySpark и Pandas с помощью pip:

```
pip install pyspark pandas
```

2. Подключение к кластеру Spark

Чтобы подключиться к кластеру Spark, вы можете использовать различные методы, в зависимости от того, как настроен ваш кластер.

Например, если у вас есть локальный кластер Spark или удаленный кластер, подключение будет различаться.

Пример для локального кластера:

```
from pyspark.sql import SparkSession

# Создание SparkSession
spark = SparkSession.builder \
      .appName("Pandas with PySpark") \
      .getOrCreate()
```

Если вы подключаетесь к удаленному кластеру, вам может понадобиться указать дополнительные параметры, такие как URL вашего кластера, например:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("Pandas with PySpark") \
    .master("spark://<your-cluster-url>:7077") \
    .getOrCreate()
```

3. Настройка среды для работы с PySpark и Pandas

a. Импорт библиотек

Импортируйте необходимые библиотеки для работы с Pandas и PySpark:

```
import pandas as pd
from pyspark.sql import SparkSession
```

b. Настройка конфигурации Spark

Если вам нужно настроить конфигурацию Spark (например, размер памяти, количество ядер), вы можете сделать это следующим образом:

```
spark = SparkSession.builder \
    .appName("Pandas with PySpark") \
    .config("spark.executor.memory", "2g") \  # Память для исполнителей
    .config("spark.driver.memory", "2g") \    # Память для драйвера
    .getOrCreate()
```

Пример работы с данными

а. Загрузка данных в PySpark DataFrame

Вы можете загрузить данные в DataFrame PySpark с помощью функции read:

```
# Загрузка CSV файла в DataFrame
spark_df = spark.read.csv("path/to/your/data.csv",
                          header=True,
                          inferSchema=True)
```

b. Конвертация PySpark DataFrame в Pandas DataFrame

После загрузки данных в DataFrame PySpark вы можете преобразовать его в DataFrame Pandas для дальнейшего анализа:

```
# Конвертация PySpark DataFrame в Pandas DataFrame
pandas_df = spark_df.toPandas()
```

c. Обработка данных с использованием Pandas

Теперь вы можете использовать все возможности Pandas для анализа данных:

```
# Пример анализа с использованием Pandas
print(pandas_df.describe())
```

5. Завершение работы

Не забудьте завершить сессию Spark после завершения работы:

```
spark.stop()
```

Заключение

Подключение к кластеру Spark и настройка среды для работы с PySpark и Pandas позволяет вам эффективно обрабатывать большие объемы данных, используя мощные возможности обеих библиотек.

С помощью PySpark вы можете масштабировать обработку данных, а Pandas предоставляет удобный интерфейс для анализа данных

Основные методы PySpark для работы с данными. Чтение данных (read_csv, read_json, read_parquet).

PySpark предоставляет мощные методы для работы с различными форматами данных. Чтение данных является одной из основных операций при обработке данных с использованием PySpark. Ниже рассмотрим основные методы для чтения данных в различных форматах: CSV, JSON и Parquet.

1. Чтение данных из CSV

Для чтения данных из файлов CSV используется метод read.csv(). Этот метод позволяет загружать данные с указанием различных параметров, таких как наличие заголовков и тип данных.

Пример:

```
from pyspark.sql import SparkSession

# Создаем SparkSession
spark = SparkSession.builder \
    .appName("Read CSV Example") \
    .getOrCreate()

# Чтение CSV файла
df_csv = spark.read.csv("path/to/your/file.csv",
                        header=True,
                        inferSchema=True)

# Вывод первых 5 строк
df_csv.show(5)
```

Параметры:

header=True: указывает, что первая строка содержит заголовки столбцов.

inferSchema=True: автоматически определяет типы данных для каждого столбца.

2. Чтение данных из JSON

Для чтения данных из файлов JSON используется метод read.json(). Этот метод позволяет загружать данные из файлов формата JSON и обрабатывать их как DataFrame.

```
# Чтение JSON файла
df_json = spark.read.json("path/to/your/file.json")

# Вывод первых 5 строк
df_json.show(5)
```

Параметры:

multiline=True: если JSON файл содержит многострочные объекты, вы можете указать этот параметр для корректного чтения.

3. Чтение данных из Parquet

Формат Parquet является колонковым форматом хранения данных, который оптимизирован для работы с большими объемами данных и поддерживает схему.

Для чтения Parquet файлов используется метод read.parquet().

Пример:

```
# Чтение Parquet файла
df_parquet = spark.read.parquet("path/to/your/file.parquet")

# Вывод первых 5 строк
df_parquet.show(5)
```

4. Дополнительные параметры чтения

При чтении данных можно указывать дополнительные параметры для управления процессом загрузки:

Чтение определенных столбцов:

Вы можете указать, какие столбцы нужно загрузить, используя метод select() после чтения данных.

```
df_selected = df_csv.select("column1", "column2")
```

Установка пользовательских схем:

Вы можете определить схему для вашего DataFrame, чтобы указать типы данных для столбцов. Это можно сделать с помощью класса StructType.

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType

schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True)
])

df_custom_schema = spark.read.csv("path/to/your/file.csv", schema=schema)
```

Заключение

Методы read.csv(), read.json() и read.parquet() в PySpark позволяют удобно и эффективно загружать данные из различных форматов.

Параметры чтения позволяют гибко управлять процессом, оптимизируя его под ваши нужды.

Работа с PySpark открывает возможности для обработки больших объемов данных и эффективного анализа информации.

Операции фильтрации, агрегации и сортировки.

Фильтрация данных в PySpark позволяет извлекать строки DataFrame на основе определенных условий. Для фильтрации используются методы filter() и where().

Пример использования filter():

```python
from pyspark.sql import SparkSession

# Создаем SparkSession
spark = SparkSession.builder \
    .appName("Filter Example") \
    .getOrCreate()

# Пример DataFrame
data = [("Alice", 25), ("Bob", 30), ("Charlie", 22)]
columns = ["name", "age"]
df = spark.createDataFrame(data, columns)

# Фильтрация данных (возраст больше 25)
filtered_df = df.filter(df.age > 25)

# Вывод результата
filtered_df.show()
```

Метод where() работает аналогично filter(), и вы можете использовать любой из них в зависимости от предпочтений.

```
# Фильтрация данных с использованием where
filtered_df_where = df.where(df.age < 30)

# Вывод результата
filtered_df_where.show()
```

Операции агрегации

Агрегация позволяет группировать данные и применять к ним функции, такие как sum(), avg(), count(), и другие. Для этого используется метод groupBy().

Пример использования groupBy()

```
# Пример DataFrame
data = [("Alice", 25, "F"),
        ("Bob", 30, "M"),
        ("Charlie", 22, "M"),
        ("David", 30, "M")]
columns = ["name", "age", "gender"]
df = spark.createDataFrame(data, columns)

# Группировка по полу и подсчет средней оценки
aggregated_df = df.groupBy("gender").agg(
    {"age": "avg"}  # Вычисление среднего возраста
)

# Вывод результата
aggregated_df.show()
```

Использование agg() для множественных агрегаций

Вы можете применять несколько агрегационных функций одновременно, используя метод agg().

```
from pyspark.sql import functions as F

# Группировка и применение нескольких функций агрегации
aggregated_multiple_df = df.groupBy("gender").agg(
    F.avg("age").alias("average_age"),
    F.count("name").alias("count")
)

# Вывод результата
aggregated_multiple_df.show()
```

Операции сортировки

Сортировка данных в PySpark выполняется с помощью метода orderBy() или sort(). Вы можете сортировать данные по одному или нескольким столбцам, указывая порядок сортировки (возрастание или убывание).

Пример использования orderBy()

```
# Сортировка по возрасту (по возрастанию)
sorted_df = df.orderBy("age")

# Вывод результата
sorted_df.show()
```

Сортировка по убыванию

Чтобы отсортировать данные по убыванию, используйте параметр ascending=False.

```
# Сортировка по возрасту (по убыванию)
sorted_desc_df = df.orderBy(df.age.desc())

# Вывод результата
sorted_desc_df.show()
```

Преобразование данных с использованием методов PySpark: select, filter, groupBy.

Преобразование данных в PySpark — это ключевая операция для анализа и обработки больших объемов данных. PySpark предоставляет различные методы, такие как select(), filter(), и groupBy(), которые позволяют выполнять преобразования и манипуляции с данными.

1. Метод select()

Метод select() используется для выбора одного или нескольких столбцов из DataFrame. Вы также можете применять функции к столбцам, такие как арифметические операции и функции агрегации.

Пример использования select()

```
from pyspark.sql import SparkSession

# Создаем SparkSession
spark = SparkSession.builder \
    .appName("Select Example") \
    .getOrCreate()

# Пример DataFrame
data = [("Alice", 25), ("Bob", 30), ("Charlie", 22)]
columns = ["name", "age"]
df = spark.createDataFrame(data, columns)

# Выбор одного столбца
selected_df = df.select("name")

# Вывод результата
selected_df.show()
```

Пример выбора нескольких столбцов и применения функции

```
from pyspark.sql import functions as F

# Выбор имени и возраста, а также добавление
# нового столбца с увеличением возраста на 1
selected_with_new_column = df.select("name",
                                     "age",
                                     (F.col("age") + 1).alias("age_plus_one"))

# Вывод результата
selected_with_new_column.show()
```

Метод filter()

Метод filter() используется для фильтрации строк DataFrame на основе определенных условий. Вы можете использовать логические операторы, такие как >, <, ==, & и |, для создания сложных условий.

Пример использования filter()

```
# Фильтрация данных (возраст больше 25)
filtered_df = df.filter(df.age > 25)

# Вывод результата
filtered_df.show()
```

Пример с несколькими условиями

```
# Фильтрация данных по нескольким условиям
# (возраст больше 20 и имя не равно "Bob")
filtered_complex_df = df.filter((df.age > 20) & (df.name != "Bob"))

# Вывод результата
filtered_complex_df.show()
```

Метод groupBy()

Метод groupBy() используется для группировки данных по одному или нескольким столбцам и применения агрегационных функций, таких как sum(), avg(), count() и другие.

Пример использования groupBy()

```
# Пример DataFrame с дополнительными данными
data = [("Alice", 25, "F"),
        ("Bob", 30, "M"),
        ("Charlie", 22, "M"),
        ("David", 30, "M")]
columns = ["name", "age", "gender"]
df = spark.createDataFrame(data, columns)

# Группировка по полу и подсчет количества
grouped_df = df.groupBy("gender").count()

# Вывод результата
grouped_df.show()
```

Пример с агрегацией

```
# Группировка по полу и вычисление средней оценки
aggregated_df = df.groupBy("gender").agg(
    F.avg("age").alias("average_age"),
    F.count("name").alias("count")
)

# Вывод результата
aggregated_df.show()
```

Заключение

Методы select(), filter(), и groupBy() в PySpark являются основными инструментами для преобразования данных.

С помощью этих методов вы можете извлекать необходимые столбцы, фильтровать данные на основе условий и группировать их для применения агрегационных функций.

Эти операции позволяют проводить глубокий анализ и извлечение полезной информации из больших наборов данных.

Пример обработки данных с Pandas API в Spark

Импорт библиотек и создание SparkSession

```
import pyspark.pandas as ps  # Импортируем Pandas API on Spark

# Создаем SparkSession
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("Pandas on Spark Example") \
    .getOrCreate()
```

Загрузка данных

Загрузим данные в DataFrame с помощью Pandas API on Spark. Допустим, у нас есть CSV файл.

```
# Загрузка данных из CSV файла
df = ps.read_csv('path/to/your/file.csv')

# Вывод первых 5 строк
print(df.head())
```

Фильтрация данных

Фильтрация данных с использованием Pandas API on Spark работает аналогично Pandas.

```
# Фильтрация строк, где возраст больше 25
filtered_df = df[df['age'] > 25]

# Вывод результата
print(filtered_df)
```

Агрегация данных

Агрегация также осуществляется как в Pandas.

```
# Группировка по полу и подсчет средней оценки
aggregated_df = df.groupby('gender')['age'].mean().reset_index()

# Переименование столбца
aggregated_df.rename(columns={'age': 'average_age'}, inplace=True)

# Вывод результата
print(aggregated_df)
```

Преобразование данных

Добавим новый столбец с увеличением возраста на 1.

```
# Добавление нового столбца с увеличением возраста на 1
df['age_plus_one'] = df['age'] + 1

# Вывод результата
print(df.head())
```

Сохранение результатов

После обработки данных вы можете сохранить результаты в файл.

```
# Сохранение результирующего DataFrame в CSV файл
aggregated_df.to_csv('path/to/save/aggregated_data.csv', index=False)
```

Домашнее задание:

1. Повторить материал лекции.

Список литературы:

1. В. Ю. Кара-ушанов SQL — язык реляционных баз данных
2. А. Б. ГРАДУСОВ. Введение в технологию баз данных
3. А.Мотеев. Уроки MySQL

Материалы лекций:

https://github.com/ShViktor72/Education

Обратная связь:

colledge20education23@gmail.com