Тема 21. Выполнение сложных трансформаций с помощью функций PySpark.

Цель занятия:

Изучить различные подходы к интеграции MongoDB с приложениями.

Учебные вопросы:

1. Введение в функции PySpark
2. Примеры сложных трансформаций
3. Оптимизация работы с данными

1. Введение в функции PySpark

В PySpark функции позволяют выполнять различные операции над данными.

1. withColumn

Используется для добавления нового столбца или замены существующего.

```
from pyspark.sql.functions import col, lit

# Добавление нового столбца с константным значением
df = df.withColumn("new_column", lit(10))

# Изменение существующего столбца
df = df.withColumn("existing_column", col("existing_column") * 2)
```

2. select

Позволяет выбирать определенные столбцы из DataFrame.

```
# Выбор нескольких столбцов
df_selected = df.select("column1", "column2")

# Выбор с выражениями
df_selected = df.select(col("column1"), (col("column2") * 2).alias("double_column2"))
```

3. filter

Фильтрует строки на основе условия.

```
# Фильтрация строк, где значение в столбце больше 100
df_filtered = df.filter(df["column"] > 100)

# Фильтрация с использованием нескольких условий
df_filtered = df.filter((df["column1"] > 100) & (df["column2"] < 50))
```

4. groupBy

Группирует данные по одному или нескольким столбцам.

```
# Группировка по одному столбцу
grouped_df = df.groupBy("category")

# Группировка по нескольким столбцам
grouped_df = df.groupBy("category", "type")
```

5. agg

Выполняет агрегатные функции на сгруппированных данных.

```
from pyspark.sql.functions import sum, avg

# Агрегация с суммой и средним значением
agg_df = grouped_df.agg(
    sum("sales").alias("total_sales"),
    avg("profit").alias("average_profit")
)
```

2. Примеры сложных трансформаций

1. Использование UDF (User Defined Functions)

UDF позволяет создавать пользовательские функции для обработки данных.

Пример UDF:

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

# Определение пользовательской функции
def convert_case(text):
    return text.upper()

# Регистрация UDF
convert_case_udf = udf(convert_case, StringType())

# Применение UDF
df = df.withColumn("uppercase_column", convert_case_udf(df["original_column"]))
```

2. Примеры группировки и агрегации

```
from pyspark.sql.functions import sum, avg, count

# Группировка по категории и подсчет количества элементов в каждой категории
grouped_df = df.groupBy("category").agg(
    count("*").alias("count"),
    sum("sales").alias("total_sales"),
    avg("profit").alias("average_profit")
)

# Показать результат
grouped_df.show()
```

Группировка по нескольким столбцам

```
# Группировка по категории и типу с агрегацией
multi_grouped_df = df.groupBy("category", "type").agg(
    sum("sales").alias("total_sales"),
    avg("rating").alias("average_rating")
)

# Показать результат
multi_grouped_df.show()
```

3. Оптимизация работы с данными

Оптимизация работы с данными в PySpark важна для повышения производительности.

Методы cache() и persist() помогают в этом, сохраняя данные в памяти для повторного использования.

1. cache(). Назначение: Сохраняет данные в памяти. Это полезно, когда вы планируете многократно использовать один и тот же DataFrame или RDD.

Особенности:

- По умолчанию сохраняет данные только в оперативной памяти.
- Подходит для часто используемых небольших наборов данных.

Использование:

```
df_cached = df.cache()
```

2. persist(). Назначение: Позволяет указать уровень хранения данных.

Уровни хранения:

- MEMORY_ONLY: Только в памяти.
- MEMORY_AND_DISK: В памяти и на диске (если не помещается в память).
- DISK_ONLY: Только на диске.

Другие уровни включают репликацию и серилизацию.

Особенности:

- Гибкость в выборе уровня хранения.
- Полезно для больших наборов данных, которые не помещаются только в память.

Использование:

```
from pyspark import StorageLevel

df_persisted = df.persist(StorageLevel.MEMORY_AND_DISK)
```

```python
from pyspark.sql import SparkSession
from pyspark import StorageLevel

spark = SparkSession.builder.appName("Optimization").getOrCreate()

# Пример DataFrame
df = spark.read.csv("path/to/file.csv", header=True, inferSchema=True)

# Кэширование
df_cached = df.cache()

# Выполнение операций
df_cached.count()
df_cached.show()

# Использование persist с MEMORY_AND_DISK
df_persisted = df.persist(StorageLevel.MEMORY_AND_DISK)

# Выполнение операций
df_persisted.count()
df_persisted.show()

spark.stop()
```

Заключение

Использование cache() и persist() помогает улучшить производительность, минимизируя повторные вычисления и улучшая доступ к данным. Выбор между ними зависит от размера данных и доступных ресурсов.

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