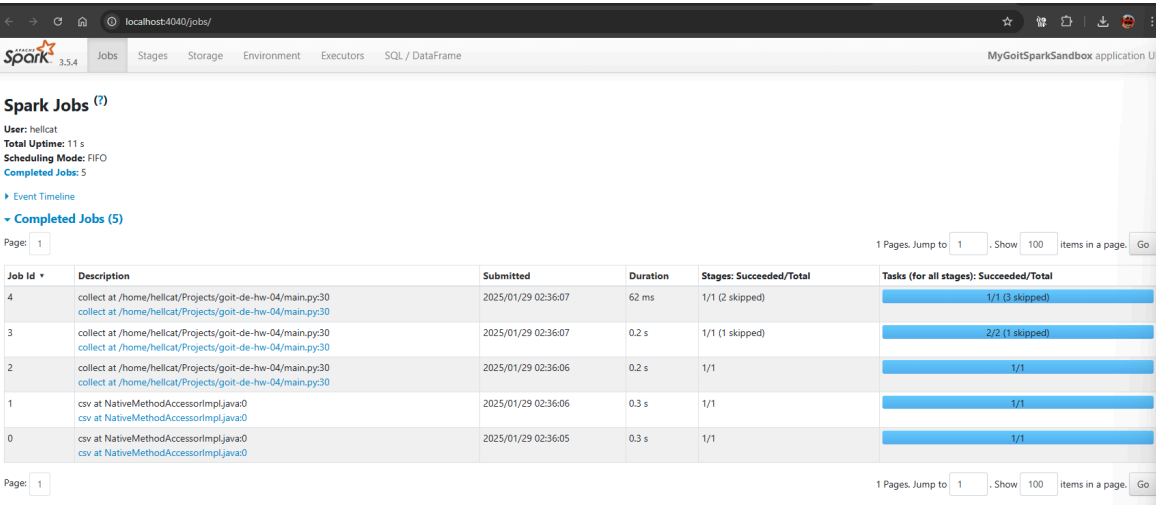


5 JOBS



Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
4	collect at /home/hellicat/Projects/goit-de-hw-04/main.py:30 collect at /home/hellicat/Projects/goit-de-hw-04/main.py:30	2025/01/29 02:36:07	62 ms	1/1 (2 skipped)	1/1 (3 skipped)
3	collect at /home/hellicat/Projects/goit-de-hw-04/main.py:30 collect at /home/hellicat/Projects/goit-de-hw-04/main.py:30	2025/01/29 02:36:07	0.2 s	1/1 (1 skipped)	2/2 (1 skipped)
2	collect at /home/hellicat/Projects/goit-de-hw-04/main.py:30 collect at /home/hellicat/Projects/goit-de-hw-04/main.py:30	2025/01/29 02:36:06	0.2 s	1/1	1/1
1	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2025/01/29 02:36:06	0.3 s	1/1	1/1
0	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2025/01/29 02:36:05	0.3 s	1/1	1/1

```
# Обробляємо дані
nuek_processed = nuek_repart \
    .where("final_priority < 3") \
    .select("unit_id", "final_priority") \
    .groupBy("unit_id") \
    .count()

# Фільтруємо за умовою count > 2
nuek_processed = nuek_processed.where("count > 2")

# Виводимо оброблені дані
print(nuek_processed.collect())
```

Пояснення

- Зчитування CSV (читання даних із файлу)
 - У Spark кожне завантаження зовнішніх даних запускає окреме завдання (Job). У цьому випадку воно відобразиться в логах як Job із позначкою `csv at NativeMethodAccessorImpl.java:0`.
- `repartition(2)`
 - Ця команда змінює кількість партицій, що зазвичай викликає процес Shuffle. У Spark Shuffle розділяє план виконання на окремі стадії, через що створюється додатковий Job.
- `groupBy("unit_id").count()`
 - Операція групування та підрахунку (`groupBy + count`) також вимагає Shuffle, тому виконується в межах окремого Job.
- `where("count>2")`
 - Це операція фільтрації після групування. Оптимізатор Spark може побудувати нову стадію для застосування фільтра, що призводить до ще одного окремого Job.
- `collect()`

- Виклик collect() змушує Spark пройти весь попередній план виконання та виконати його. Це фінальний Job, який запускає всі попередні стадії, якщо вони ще не були обчислені.

Висновок:

У логах SparkUI буде відображено 5 завершених Jobs.

8 JOBS

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
7	collect at /home/helicat/Projects/goit-de-hw-04/02.py:30 collect at /home/helicat/Projects/goit-de-hw-04/02.py:30	2025/01/29 02:41:00	44 ms	1/1 (2 skipped)	1/1 (3 skipped)
6	collect at /home/helicat/Projects/goit-de-hw-04/02.py:30 collect at /home/helicat/Projects/goit-de-hw-04/02.py:30	2025/01/29 02:41:00	46 ms	1/1 (1 skipped)	2/2 (1 skipped)
5	collect at /home/helicat/Projects/goit-de-hw-04/02.py:30 collect at /home/helicat/Projects/goit-de-hw-04/02.py:30	2025/01/29 02:41:00	44 ms	1/1	1/1
4	collect at /home/helicat/Projects/goit-de-hw-04/02.py:25 collect at /home/helicat/Projects/goit-de-hw-04/02.py:25	2025/01/29 02:41:00	50 ms	1/1 (2 skipped)	1/1 (3 skipped)
3	collect at /home/helicat/Projects/goit-de-hw-04/02.py:25 collect at /home/helicat/Projects/goit-de-hw-04/02.py:25	2025/01/29 02:41:00	0.2 s	1/1 (1 skipped)	2/2 (1 skipped)
2	collect at /home/helicat/Projects/goit-de-hw-04/02.py:25 collect at /home/helicat/Projects/goit-de-hw-04/02.py:25	2025/01/29 02:40:59	0.2 s	1/1	1/1
1	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2025/01/29 02:40:59	0.3 s	1/1	1/1
0	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2025/01/29 02:40:58	0.3 s	1/1	1/1

```
nuek_repart = nuek_df.repartition(2)

nuek_processed = nuek_repart \
    .where("final_priority < 3") \
    .select("unit_id", "final_priority") \
    .groupBy("unit_id") \
    .count()

# Проміжний action: collect
nuek_processed.collect()
```

Пояснення

1. Читання CSV
 - Це початкова операція зчитування даних, яка створює окремий Job.
2. repartition(2)

- Викликає процес Shuffle, що спричиняє додатковий Job.
- 3. `groupBy("unit_id").count()`
 - Операція групування та підрахунку (`groupBy + count`) також потребує Shuffle, що запускає ще один Job.
- 4. Проміжний `collect()`
 - Оскільки це Action, Spark виконує всі попередні операції, накопичені в DAG, до цього моменту. Це може призвести до об'єднання або розподілу Job-ів залежно від фізичного плану, але в результаті додає додаткові запуски Job.
- 5. Друга фільтрація `where("count>2")`
 - Це нова трансформація, що змінює DAG і створює ще один етап обчислень.
- 6. Фінальний `collect()`
 - Оскільки це Action, Spark повторно виконує всі необхідні обчислення, включно з можливими Shuffle, для оновленого DataFrame.

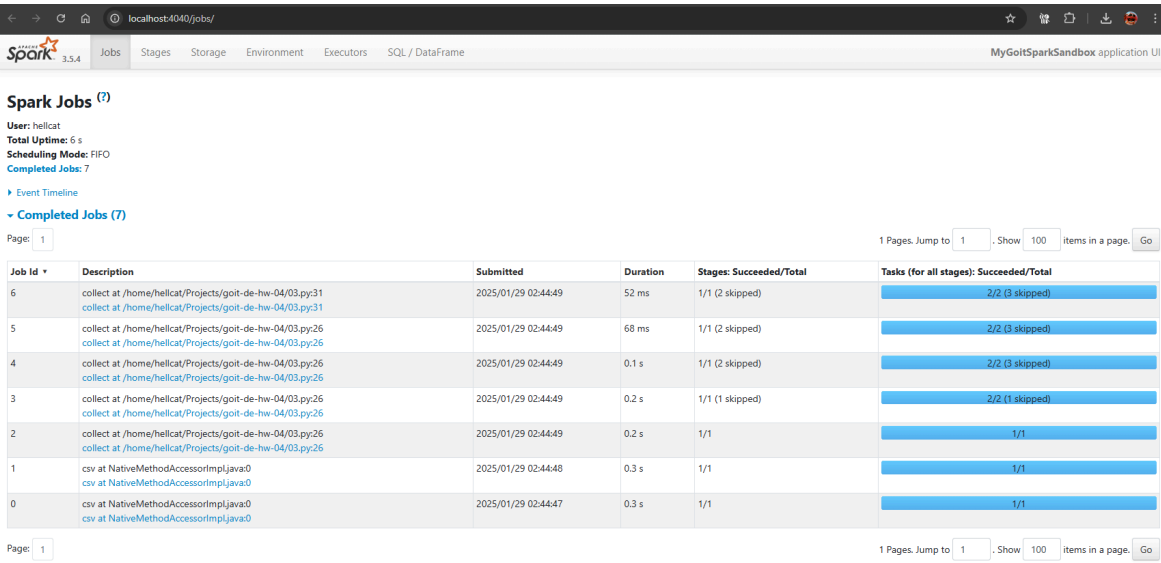
Висновок:

У логах SparkUI буде 8 завершених Jobs.

Чому кількість зросла з 5 до 8?

Проміжний `collect()` викликає обчислення всіх попередніх Job ще до застосування `where("count>2")`. Після цього, коли виконується фільтрація та повторний `collect()`, Spark змушений розпочати новий цикл обчислень, що призводить до збільшення кількості Job.

7 JOBS and CACHE



Spark Jobs ⁽⁷⁾

User: helicat
Total Uptime: 6 s
Scheduling Mode: FIFO
Completed Jobs: 7

► Event Timeline

▼ Completed Jobs (7)

Page: 1 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▼	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
6	collect at /home/helicat/Projects/goit-de-hw-04/03.py31 collect at /home/helicat/Projects/goit-de-hw-04/03.py31	2025/01/29 02:44:49	52 ms	1/1 (2 skipped)	2/2 (3 skipped)
5	collect at /home/helicat/Projects/goit-de-hw-04/03.py26 collect at /home/helicat/Projects/goit-de-hw-04/03.py26	2025/01/29 02:44:49	68 ms	1/1 (2 skipped)	2/2 (3 skipped)
4	collect at /home/helicat/Projects/goit-de-hw-04/03.py26 collect at /home/helicat/Projects/goit-de-hw-04/03.py26	2025/01/29 02:44:49	0.1 s	1/1 (2 skipped)	2/2 (3 skipped)
3	collect at /home/helicat/Projects/goit-de-hw-04/03.py26 collect at /home/helicat/Projects/goit-de-hw-04/03.py26	2025/01/29 02:44:49	0.2 s	1/1 (1 skipped)	2/2 (1 skipped)
2	collect at /home/helicat/Projects/goit-de-hw-04/03.py26 collect at /home/helicat/Projects/goit-de-hw-04/03.py26	2025/01/29 02:44:49	0.2 s	1/1	1/1
1	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2025/01/29 02:44:48	0.3 s	1/1	1/1
0	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2025/01/29 02:44:47	0.3 s	1/1	1/1

Page: 1 1 Pages. Jump to 1 . Show 100 items in a page. Go

```
nuek_processed_cached = nuek_repart \
    .where("final_priority < 3") \
    .select("unit_id", "final_priority") \
    .groupBy("unit_id") \
    .count() \
    .cache() # Додано функцію cache

# Проміжний action: collect
nuek_processed_cached.collect()

# Ось ТУТ додано рядок
nuek_processed = nuek_processed_cached.where("count>2")

nuek_processed.collect()
```

Пояснення

1. Читання CSV + repartition(2) + groupBy("unit_id").count()
 - Аналогічно попереднім випадкам, ці операції спричинять кілька Job через необхідність Shuffle.
2. .cache()
 - Важливий момент: при першому виклику collect() на кешованому DataFrame, Spark виконує всі обчислення лише один раз, а потім зберігає результат (за замовчуванням у пам'яті).
3. Проміжний collect()
 - Запускає виконання всіх попередніх операцій і записує отриманий результат у кеш.
4. .where("count>2")
 - Оскільки обчислення вже кешовані, ця трансформація працює з готовими даними. Spark не повторює групування та попередні обчислення.
5. Фінальний collect()
 - Використовує вже частково обчислені дані з кешу, що дозволяє уникнути зайвих Shuffle та додаткових Job, які виникли б без кешування.

Висновок:

Завдяки кешуванню загальна кількість Job зменшується. Замість 8 Job (як у попередньому випадку), отримуємо 7, оскільки кешування усуває необхідність повторного виконання деяких обчислень.

Чому `cache()` зменшує кількість Job?

- Без кешування Spark щоразу обчислює весь DAG заново при кожному виклику Action.
- `cache()` зберігає результати першого обчислення. При наступних викликах Action на тому ж DataFrame (або його частині) Spark використовує кешовані дані, а не виконує всі обчислення з нуля.
- Як наслідок – зменшується кількість Job, оскільки уникаються зайві Shuffle та перерахунок стадій.