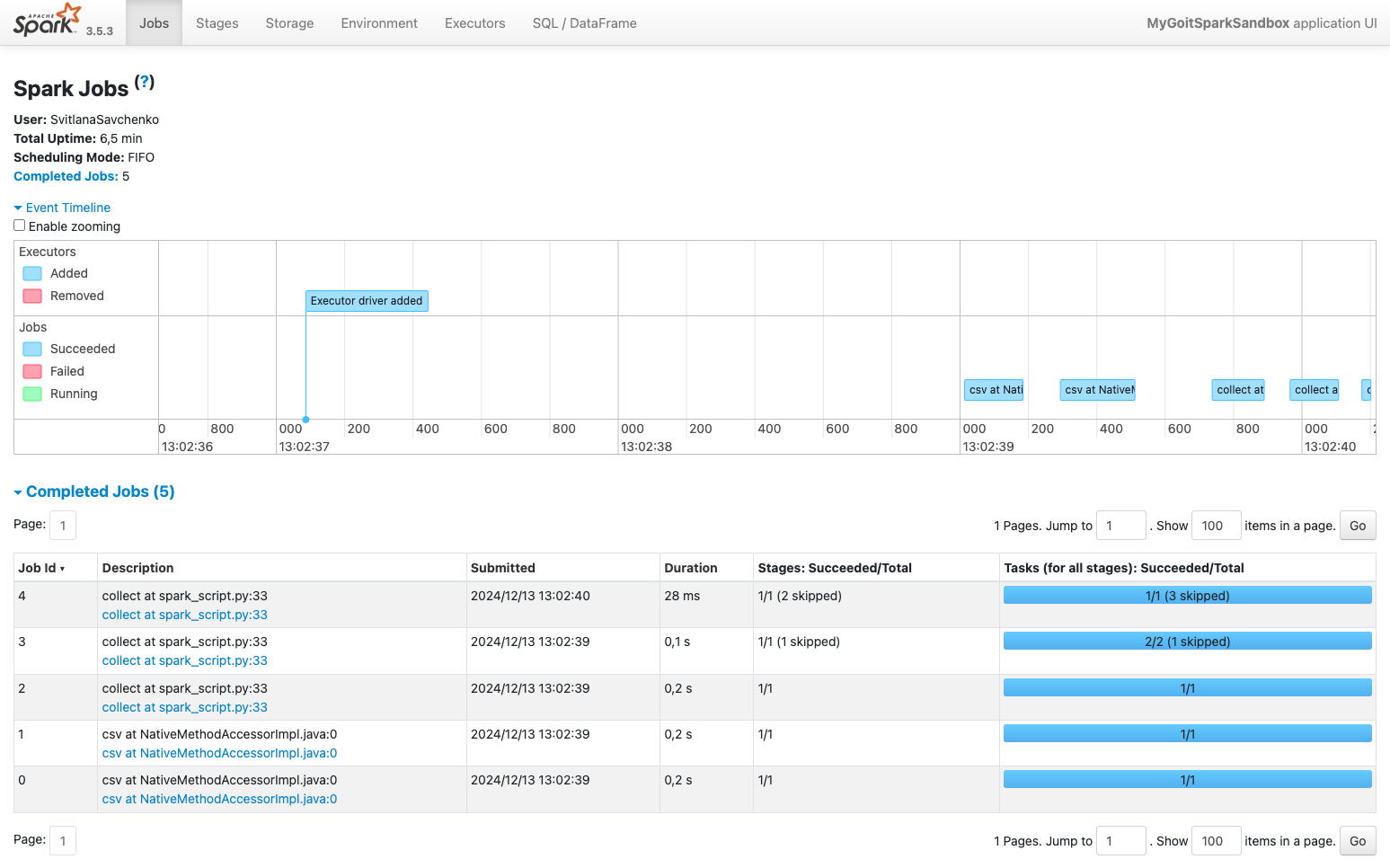
**Домашнє завдання до теми «Apache Spark. Оптимізація та SparkUІ»**

**Частина 1**

****

### **Аналіз виконання Jobs**

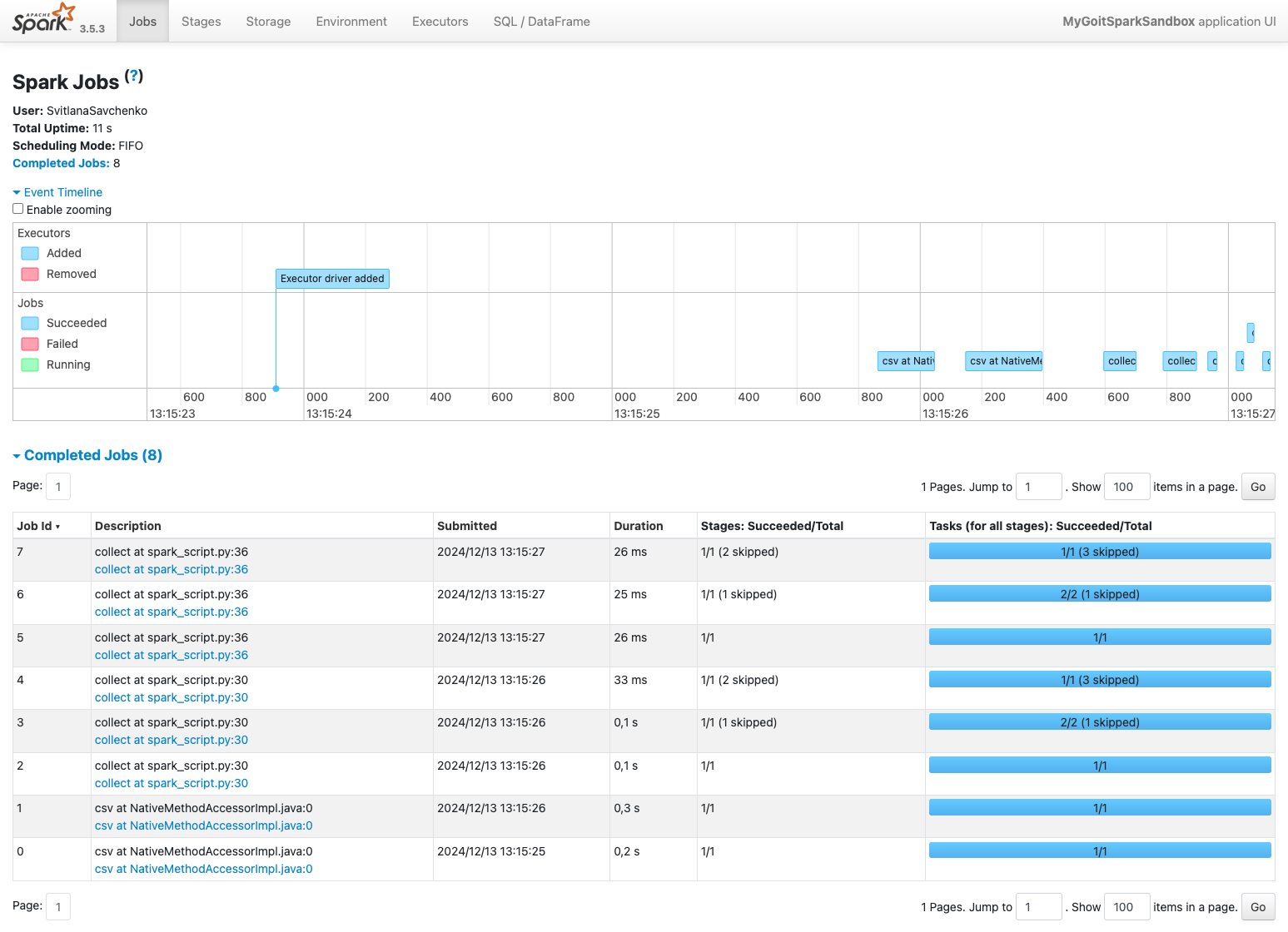
**Що означає наявність 5 Jobs?**

1. Завантаження даних (csv).
2. Перерозподіл (repartition).
3. Фільтрація (where("final\_priority < 3")).
4. Групування (groupBy + count).
5. Фільтрація (where("count > 2")) + collect.

**Висновок:**

Кожен етап обробки даних у Spark створює окремий Job, коли виконується action (у нашому випадку це collect()). Це пояснює, чому в нас є 5 Jobs у SparkUI.

**Частина 2**

****

### **Пояснення, чому додалося 3 Jobs**

* Додано проміжний collect(), що змушує Spark обчислити всі трансформації до цього моменту.
* Тепер Spark двічі виконує обчислення:
  1. Перший collect() викликає обчислення для всіх попередніх трансформацій (до groupBy і count включно).
  2. Другий collect() запускає обчислення ще раз для додаткової фільтрації where("count > 2").

Тому кількість Jobs зросла:

* 1 Job для читання CSV-файлу.
* 1 Job для repartition.
* 1 Job для фільтрації where("final\_priority < 3").
* 1 Job для select + groupBy + count.
* 1 Job для першого collect().
* Після першого collect() Spark завершує всі попередні обчислення.
* Далі, при другому collect():
  + 1 Job для фільтрації where("count > 2").
  + 2 Jobs для повторного обчислення групування (groupBy + count) і фільтрації.

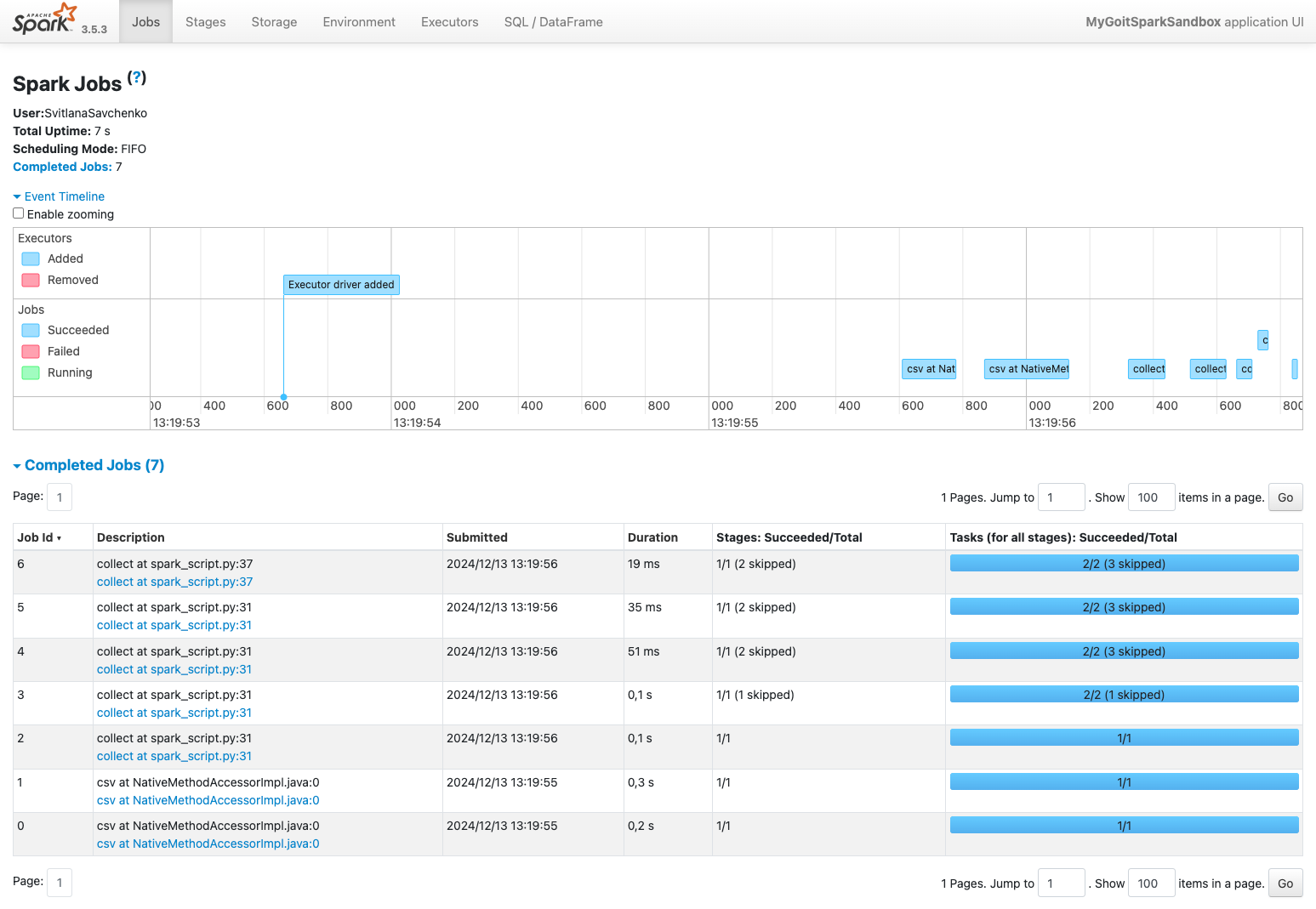
Тобто у результаті отримуємо 8 Jobs:

1. Читання CSV.
2. Перерозподіл на 2 партіції.
3. Фільтрація final\_priority < 3.
4. Групування та підрахунок (groupBy + count).
5. Проміжний collect().
6. Повторне групування.
7. Фільтрація count > 2.
8. Другий collect().

**Висновок:**

Spark не зберігає результати обчислень у пам'яті за замовчуванням. Тому коли ми додаємо другий collect(), Spark повторно обчислює всі попередні кроки для виконання нового запиту. Тобто доданий collect() змушує Spark запускати обчислення двічі, що збільшує кількість Jobs.

**Частина 3**

****

### **Чому стало 7 Jobs після використання cache()**

При використанні cache(), Spark оптимізує виконання запитів, що призводить до скорочення кількості Jobs. У нашому випадку зменшення до 7 Jobs (замість 8) сталося тому, що Spark уникає повторних обчислень, використовуючи кешований результат.

### Детальний аналіз

Ось як розбиваються Jobs у частині 3:

1. 1 Job — читання CSV-файлу.
2. 1 Job — перерозподіл на 2 партіції (repartition).
3. 1 Job — фільтрація final\_priority < 3.
4. 1 Job — вибірка стовпців і групування (select, groupBy, count).
5. 1 Job — кешування і перший collect(). Після цього результат зберігається в кеші.
6. 1 Job — додаткова фільтрація count > 2.
7. 1 Job — другий collect(), який використовує кешовані дані.

### Чому кількість Jobs зменшилася:

* Без кешування Spark кожен раз виконував повне обчислення усього ланцюжка трансформацій при виклику collect().
* З кешуванням Spark запам'ятав результат на етапі кешування, тому на другому етапі collect() Spark не виконує повторно весь ланцюжок, а лише застосовує додаткову фільтрацію та повертає результат.

### Порівняння з частиною 2:

У частині 2 (без cache()) було 8 Jobs, оскільки Spark кожен раз виконував усі трансформації заново:

* Спочатку виконувалася повна обробка даних для першого collect().
* Потім, при повторному collect(), обробка починалася з самого початку.

У частині 3, завдяки кешуванню, Spark уникає дублювання обчислень, і саме це скорочує кількість Jobs.

### **Висновок:**

Кешування оптимізує роботу Spark, особливо коли ми виконуємо кілька дій (actions) на тому самому DataFrame. Це дозволяє уникати зайвих обчислень і знижувати навантаження на систему, що й призвело до зменшення кількості Jobs до 7.