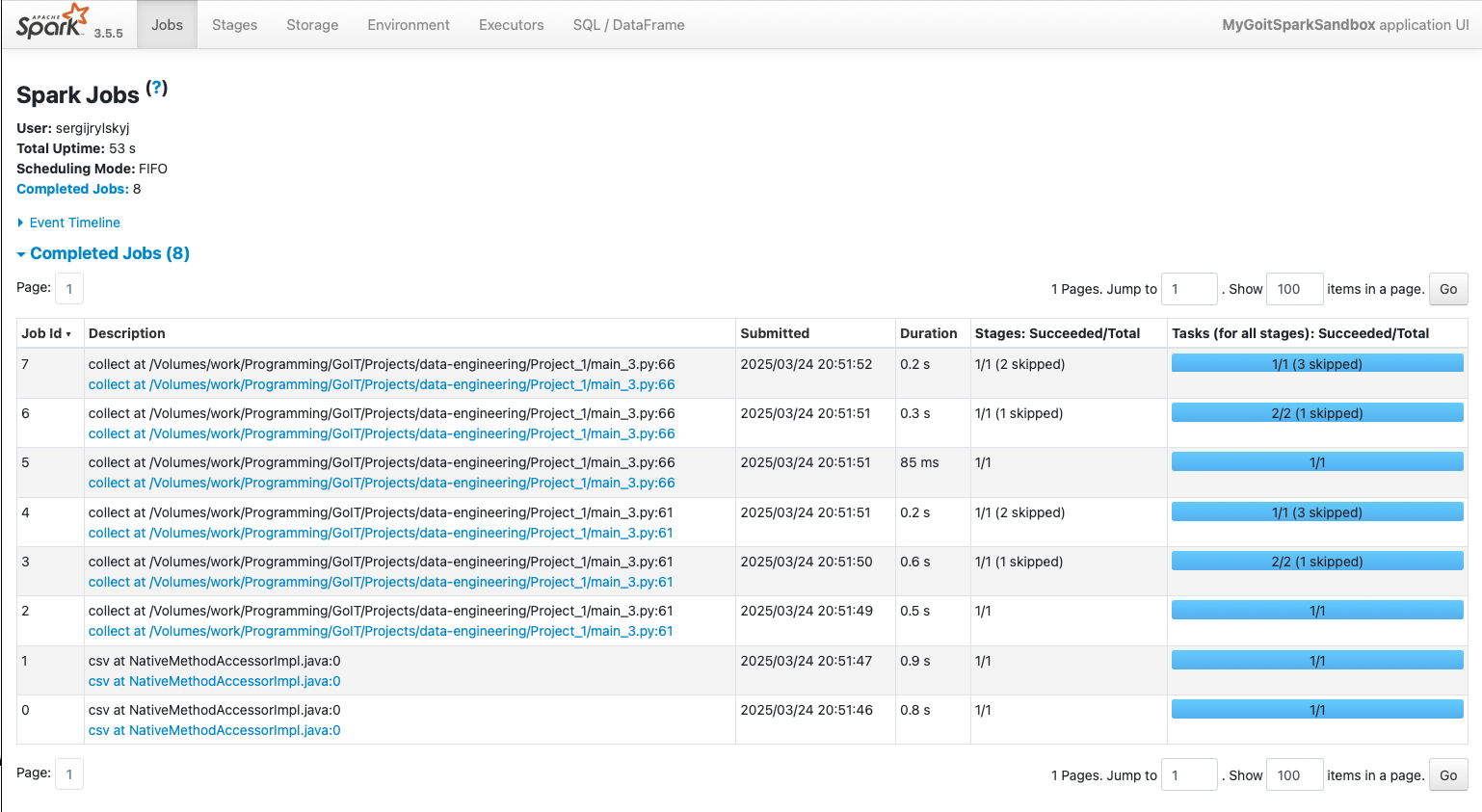


## **Сценарій 1: Базовий код з доданим фільтром**

У першому сценарії код робить таке:

1. Завантажує CSV файл з даними про екстрені виклики
2. Розділяє дані на 2 частини
3. Фільтрує рядки де final\_priority < 3
4. Вибирає потрібні стовпці
5. Групує за unit\_id і рахує кількість
6. Додає ще один фільтр: лише записи, де кількість > 2
7. Збирає результати командою collect()

Це створює 5 Spark-завдань (jobs), бо коли ви викликаєте collect(), Spark виконує всі попередні трансформації за один раз.



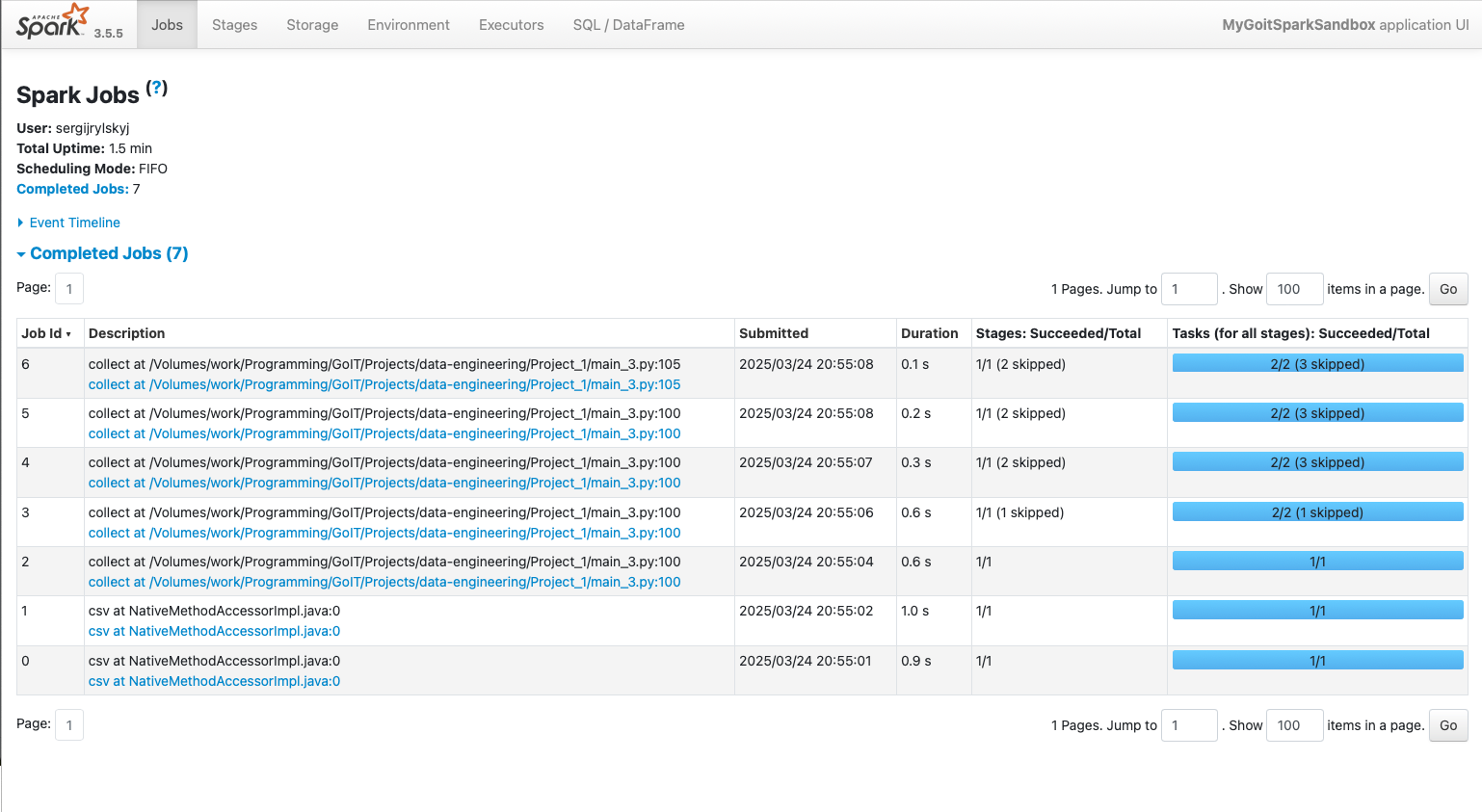
## **Сценарій 2: Код з проміжною дією**

У другому сценарії додано проміжний виклик collect():

1. Завантажуємо і трансформуємо дані до групування та підрахунку
2. Викликаємо collect(), щоб отримати і показати результати
3. Додаємо ще один фільтр (count > 2)
4. Знову викликаємо collect()

Це створює 8 Spark-завдань, бо:

* Перший collect() запускає завдання для початкових трансформацій
* Другий collect() змушує Spark повторно обчислювати всі попередні трансформації плюс новий фільтр
* По суті, ви запускаєте весь процес двічі, що неефективно



## **Сценарій 3: Використання кешування**

У третьому сценарії додано функцію cache():

1. Завантажуємо і трансформуємо дані до групування і підрахунку
2. Кешуємо результат у пам'яті
3. Викликаємо collect(), щоб матеріалізувати і зберегти результати у кеші
4. Додаємо фільтр count > 2
5. Знову викликаємо collect()

Це створює 7 Spark-завдань, бо:

* Перший collect() запускає завдання для обчислення і кешування результатів
* Другий collect() використовує вже кешовані дані і застосовує лише новий фільтр
* Це дозволяє уникнути повторного обчислення всього процесу з нуля
* Сама операція cache() потребує додаткового завдання для збереження даних у пам'яті

Перевага кешування очевидна - ви зменшили кількість завдань з 8 до 7, бо не потрібно заново обробляти вихідний CSV-файл, перерозподіляти, фільтрувати, вибирати і групувати дані. Замість цього використовуються вже збережені в пам'яті проміжні результати.