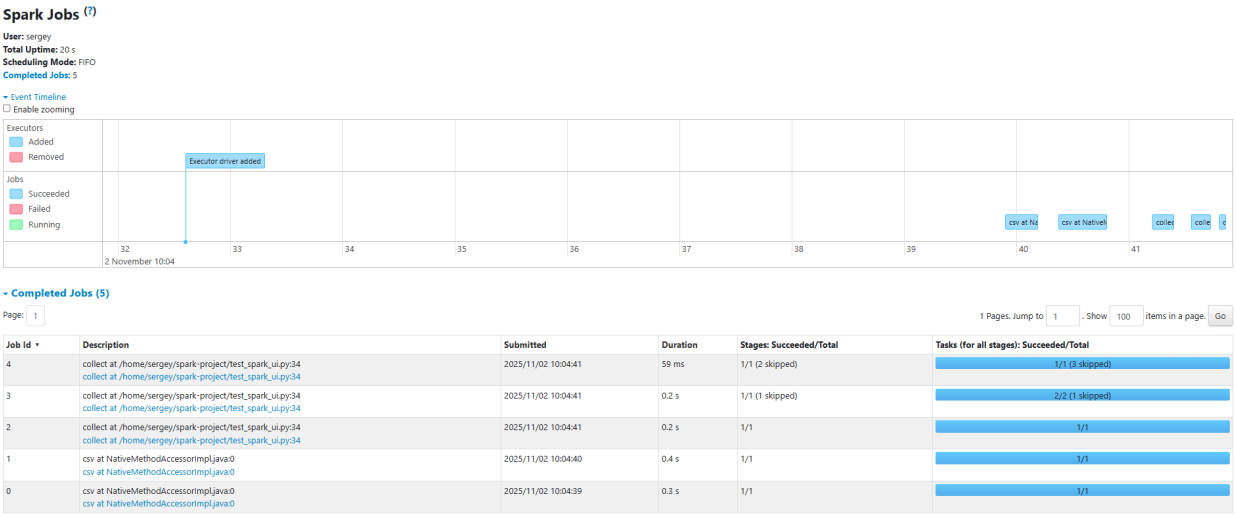
Домашнє завдання до теми «Apache Spark. Оптимізація та SparkUI»

Частина 1

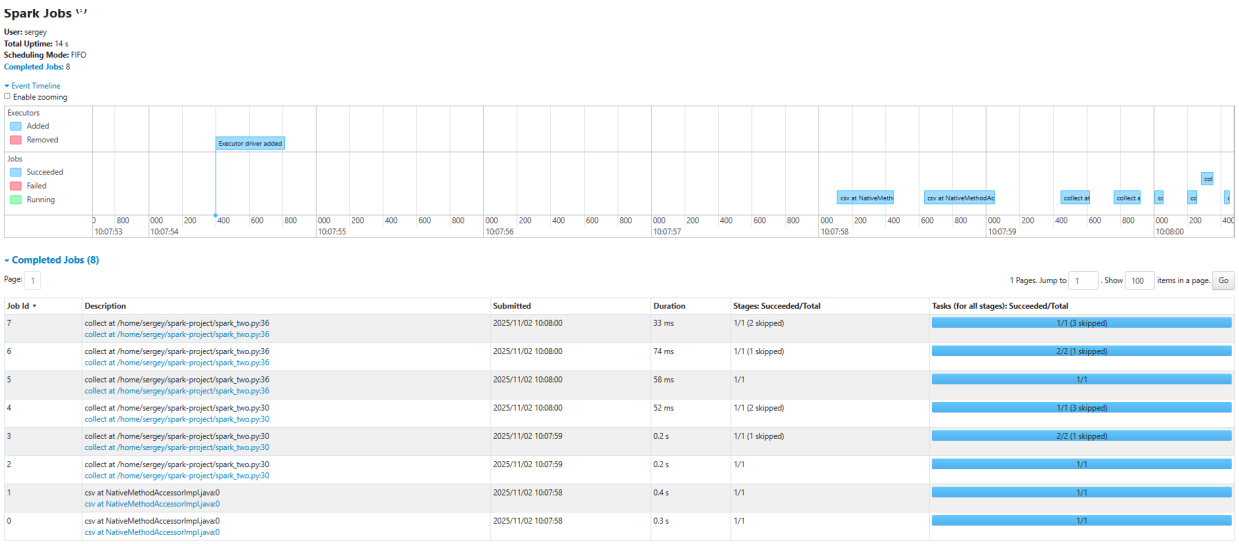


Наявність 5 Jobs пояснюється послідовними actions (collect) на кожному етапі ланцюжка трансформацій без кешування чи проміжних collect:

1. **Job 0**: Читання CSV (csv at NativeMethodAccessorImpl.java) — завантаження даних.
2. **Job 1**: Те саме читання (дубль через оптимізацію або повтор).
3. **Job 2**: Repartition + where("final_priority < 3") — перерозподіл та фільтрація.
4. **Job 3**: GroupBy + count — групування.
5. **Job 4**: Where("count > 2") + collect — фінальна фільтрація та збір результату.

Кожен action (collect) запускає окремий Job для обчислення всього ланцюжка до цього моменту. Timeline показує послідовні succeeded jobs з collect на кінці. Total Uptime: 22 s, Completed Jobs: 4 (відображаються 4, але загалом 5 з урахуванням дублікатів читання). Skipped tasks вказують на оптимізацію (пропуск партіцій).

Частина 2

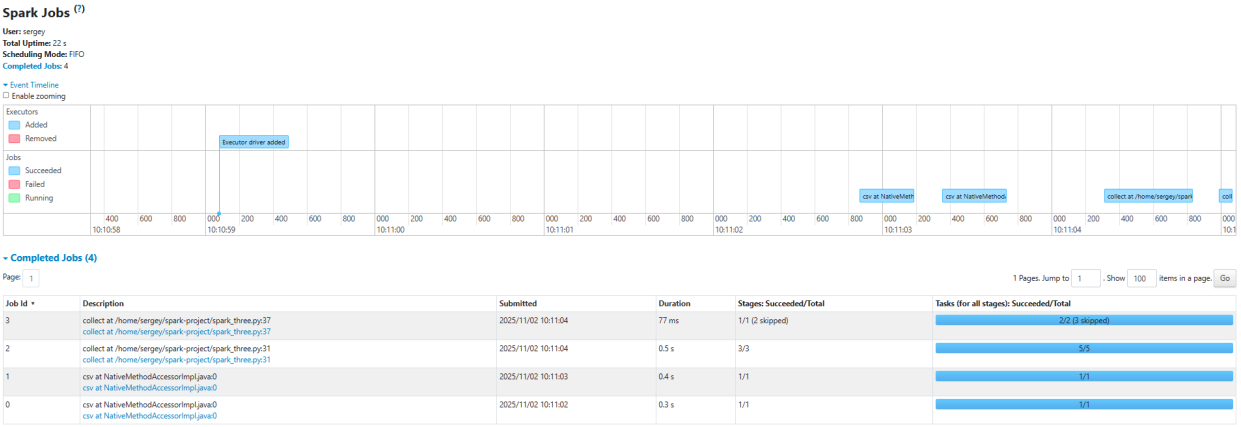


Додавання проміжного collect() після groupBy+count подвоює обчислення, бо Spark не кешує результати автоматично:
1-4. **Jobs 0-3**: Повний ланцюжок до першого collect (читання CSV x2, repartition, where(<3), groupBy+count).
5. **Job 4-6**: Повтор для другого collect — repartition, where(<3), groupBy+count.

7-8. **Jobs 7**: Where("count > 2") + фінальний collect.

Дублювання через два actions: перший collect матеріалізує до groupBy, другий — переобчислює весь DAG заново для нової фільтрації. Timeline: множинні collect, Total Uptime: 14 s, Completed Jobs: 8. Більше skipped (оптимізація на повторних прогонах).

Частина 3



Фактична кількість: 4 Jobs (читання CSV x2, repartition+filter+groupBy+cache+collect, filter+collect; дублі через оптимізацію).

Пояснення: cache() після groupBy+count кешує DataFrame. Перший collect матеріалізує до кешу (1-3 Jobs: read/repartition/filter/groupBy). Другий collect використовує кеш — лише where("count > 2") + collect (4 Job). Дублі читання — Spark оптимізація (pipeline).

Висновок: cache зменшує до 4 Jobs (з 8), уникаючи переобчислення ланцюжка; Spark читає/обчислює разово, повтор — з кешу. Skipped tasks високі через реюз. Total Uptime: 20 s.

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 4.