Optimizing Apache Spark Performance by Managing Query Plan Size

Abstract

As Apache Spark matures into a robust and powerful data processing platform, increasingly complex workloads expose performance bottlenecks related to query plan size. This white paper delves into a specific Spark performance issue triggered by excessive growth of the query plan tree—particularly in workloads built through iterative DataFrame operations or deeply nested SQL views. We highlight how Spark's internal plan cloning mechanisms amplify this issue, and propose a solution that enables early plan simplification without breaking Spark's caching mechanism.

---

1. Introduction

Spark workloads in production are rarely flat or simple. In real-world scenarios, users often compose queries over layers of views or build DataFrames iteratively, appending projections and filters in loops. These practices, while flexible, lead to deeply nested and voluminous query plans. In extreme cases, plans can include tens of millions of nodes, severely affecting both memory usage and query performance.

---

2. Problem Overview: Query Plan Tree Explosion

From Spark 3.x onwards, Spark's planner architecture clones the query plan at multiple stages:

Unanalyzed Logical Plan
  ↓ (clone)
Analyzed Logical Plan
  ↓ (clone)
Optimized Logical Plan
  ↓ (clone)
Spark Plan
  ↓ (clone)
Physical Plan

Each stage operates on a clone of the previous plan, and the cost of plan duplication and transformation increases with the size of the tree. Analysis-phase rules, such as the Deduplication rule (introduced in Spark 3), exacerbate this by traversing the entire plan tree. The problem compounds when new DataFrames are built upon already-analyzed plans—causing reapplication of heavy analysis rules on ever-larger trees.

---

3. Why Optimization Comes Too Late

Spark includes a rule called `CollapseProject` which merges redundant projection nodes—but only during the **optimization phase**. However, new DataFrames are created using already-analyzed plans. Since the optimization hasn't yet run, these analyzed plans accumulate unnecessary projection nodes.

The intuitive solution of collapsing projects earlier during analysis has not been implemented in stock spark because early collapse causes incompatibilities with Spark's **caching mechanism**.

---

4. The Caching Challenge

Let's examine a simple case:

```
val df = baseDF.filter($"x" > 7).select(($"x" + $"y").as("A"), $"x", $"y", $"z")
df.cache()
```

Caching stores the result of `df` with the following plan:

```
Project (A = x + y, x, y, z)
  |
Filter (x > 7)
  |
BaseRelation (x, y, z)
```

If another projection is applied:

```
val df1 = df.select($"A", $"x", $"y", $"z", ($"y" + $"z").as("B"))
```

Collapsing projections during analysis would yield:

```
Project (A = x + y, x, y, z, B = y + z)
  |
Filter (x > 7)
  |
BaseRelation (x, y, z)
```

This structure no longer matches the cached key (the original analyzed plan), causing the cache lookup to fail.

---

5 The Solution: Retaining Cache Compatibility During

The details of the solution can be provided on personal request. Please reach out to us at asif.shahid@kwikquery.com.

6 Limitations and Future Work

Currently, the PR is applicable only to code submitted directly to the driver, and does not yet support Spark Connect environments. Handling interspersed filters or more deeply nested transformations will require further enhancements.

7. Conclusion

Managing query plan size is crucial for achieving scalable performance in Apache Spark, especially as workloads become increasingly complex. By intelligently collapsing projection nodes earlier in the lifecycle—without breaking the caching layer—Spark can significantly reduce memory usage and improve planning performance.

This white paper outlines both the problem and a practical solution implemented in PR #49124, paving the way for more efficient Spark applications.