Optimizing Constraint Propagation in Apache Spark: Eliminating Permutational Explosion with a Novel Algorithm

1. Executive Summary

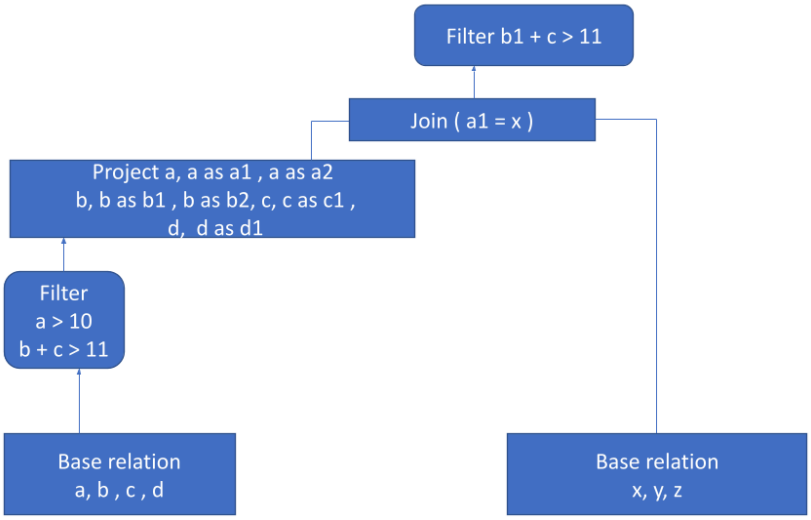
Apache Spark's Catalyst optimizer plays a pivotal role in enhancing query performance by applying various optimization techniques. One such technique is constraint propagation, which infers and propagates data constraints throughout the logical plan to enable further optimizations. However, the existing implementation of constraint propagation in Spark has been identified to cause significant performance issues, including exponential growth in inferred constraints, leading to increased compilation times and potential out-of-memory (OOM) errors. This paper introduces a novel algorithm designed to address these challenges by optimizing the constraint propagation process, thereby improving the efficiency and scalability of Spark's query optimization.

2. Background and Motivation

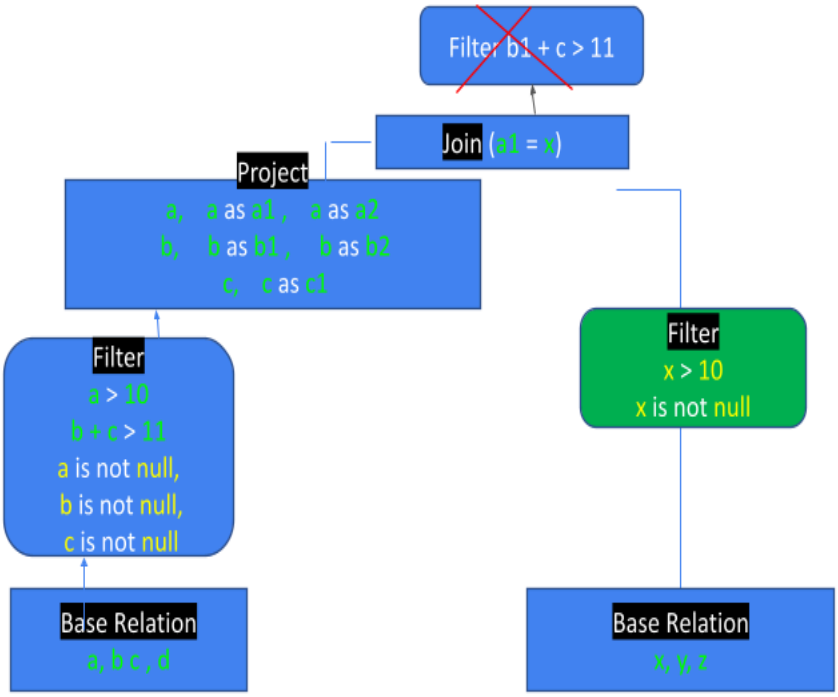
Constraint propagation in Spark is intended to enhance query optimization:

- By inferring additional constraints from existing ones and propagating them through the logical plan. For example, if a filter condition specifies ***a > 10***, the optimizer can infer that ***a IS NOT NULL*** and propagate this constraint to other parts of the plan.
- Removing redundant filters in the plan also known as “Filter Pruning”
- Pushing new Filters below either side of the Join Operator.

Consider a starting plan before application of Constraints Propagation rule:



After the Constraint Propagation rule, the plan will be optimized as



So the effect of Constraint Propagation Rule causes the plan

1) To have new ***is not null*** filters ***(a is not null, b is not null, c is not null)***
2) Filter ***b1 + c > 11*** getting removed, as it is redundant due to the existing (***b + c > 11***)
3) New Filter ***x > 10 and x is not null*** being created on the Right leg of Join

However, the current implementation has notable shortcomings:

- **Creation of EqualsNullSafe Constraint for every combination of aliased projection:**Those constraints are not needed in the new algorithm
- **Performance Bottlenecks**: In complex queries with numerous constraints and aliases, the propagation process can result in a combinatorial explosion of inferred constraints, significantly increasing compilation time.
- Despite pessimistically creating all permutations of constraints, **it may still miss out on constraints.**
- **OOM Errors**: Being permutational logic, the actual queries have been known to generate billions of constraints, resulting in queries failing in compilation stage due to OutOfMemoryErrors, after spending hours (have seen queries failing after 8 hours of compiling)

These issues have been documented in Spark's JIRA under SPARK-33152, highlighting the need for an improved constraint propagation mechanism.

3. Problem Statement

The core problem lies in the **permutational computation of constraints** during the propagation process. The existing algorithm considers all possible combinations of constraints, leading to an exponential increase in the number of inferred constraints as the complexity of the query grows. This not only hampers performance but also poses scalability challenges for large-scale data processing.

Illustrative Example

Consider the above unoptimized plan

- traverses the tree from bottom to top
- On encountering filter node (a > 10 and b + c > 11), it stores following constraints
- a > 10, b + c > 11 and also 3 new constraints
    - a is not null, b is not null, c is not null
- ON encountering Project node, where the incoming attributes are aliased **, create all the possible combinations of the constraints so far. So the overall constraints now becomes**
- **a > 10, a1 > 10, a2 > 10, b + c > 11, b + c1 > 11, b1 + c > 11, b1 + c1 > 11, a is not null, a1 is not null, a2 is not null, b is not null, b1 is not null, c is not null, c1 is not null, EqualNullSafe(a, a1), EqualNullSafe(a, a2), EqualNullSafe(a1, a2), EqualNullSafe(c, c1), EqualNullSafe(b, b1),**
-

total number of constraints emanating from project node = 19, which is a problem

The logic through which these generated constraints help in pruning the redundant filer b1 + c > 11 or adding a new filter on the right leg of the join is

- To prune filter b1 + c > 11, it consults the incoming constraints to see if Filter already exists & so can be removed.
- To add a new filter to the RHS of the Join, it checks if any constraint exists on a1 (because join condition is a1 = x). It finds a1 > 10 & a1 is Not Null hence generates a new filter (x > 10 && x is not null) on the RHS table.
- For Mathematically inclined:

  So if a Filter expression is a function of F(a, b), where

  - Attribute ‘a’ or any of its aliases repeating ‘m’ times
  - Attribute ‘b’ or any of its aliases repeating ‘n’ times
  - if count of attribute a + all aliases of ‘a’ = p
  - if count of attribute b + all aliases of ‘b’ = q

  Total constraints created for 1 such Filter expression = p ^ m * q ^ n

4. Proposed Algorithm

To mitigate the issues associated with the existing constraint propagation mechanism, a new algorithm has been proposed which uses Alias Tracking and canonicalization

- **Avoidance of Redundant Inference**: The algorithm tracks inferred constraints to prevent redundant computations, ensuring each constraint is inferred only once.
- No need for EqualNullSafe constraints

- **Efficient Propagation Strategy**: By employing a worklist-based approach, the algorithm systematically propagates constraints without exploring all permutations, thereby reducing computational overhead.
- **Integration with Catalyst Optimizer**: The new algorithm seamlessly integrates with Spark's Catalyst optimizer, maintaining compatibility with existing optimization rules.

Algorithm Overview

- traverses the tree from bottom to top as before
- On encountering filter nodes (a > 10 and b + c > 11), store in data structure, the following expressions:
- a > 10, b + c > 11 and also 3 new constraints
    - a is not null, b is not null, c is not null
- ON encountering Project node, where the incoming attributes are aliased, create additional List structures which are

List_1 : a, a1, a2

List _2 : b, b1, b2

List_3 : c, c1

So total Constraints remain at : 5 vs 19

5. Working of the new Algorithm

- To prune filter b1 + c > 11, refer to the incoming Constraints Data Structure:
    - For each attribute in the expression b1 + c > 11, find out List to which it belongs
    - From that List pick the 0^{th} element and replace the attribute in the above filter with that
    - So we get b + c > 11. (This is called canonicalization). Since b + c > 11 is part of constraint, filter can be pruned
- To add a new filter to the RHS of the Join, we need to find if any constraint exists on a1 (because join condition is a1 = x).
    - For a1, find out List to which it belongs
    - From that List pick the 0^{th} element, which will be 'a'
    - Check the base constraints which exclusively depend only on 'a'
    - We get a > 10 & a is not null.
    - Replace 'a' with 'x' and we get filters to push down x > 10 && x is not null

Benchmark Results

Performance benchmarks demonstrate significant improvements:

- **Compilation Time**: Reduced from approximately 13.9 seconds to 247 milliseconds for complex queries, representing a more than 50x improvement.
- **Memory Usage**: Substantial reduction in memory consumption during query compilation, mitigating the risk of OOM errors.

In real world examples with complex case statements and aliases, the new algorithm has managed to bring query compilation times from > 10 Hours to under few seconds

These results underscore the effectiveness of the new algorithm in enhancing Spark's query optimization performance.

6. Evaluation and Results

The new constraint propagation algorithm has been evaluated across various scenarios:

- **Complex Queries**: Demonstrated significant reductions in compilation time and memory usage for queries with multiple joins and filters.
- **Scalability**: Improved scalability for large datasets and complex query plans, enabling more efficient processing in big data environments.
- **Compatibility**: Maintained compatibility with existing optimization rules and query semantics, ensuring seamless integration.

7. Use Cases and Implications

The optimized constraint propagation mechanism offers substantial benefits for:

- **Data Engineers and Analysts**: Faster query compilation times lead to improved productivity and reduced latency in data analysis workflows.
- **Large-Scale Data Processing**: Enhanced scalability and efficiency for processing large datasets with complex query logic.
- **System Stability**: Reduced risk of OOM errors during query compilation, improving overall system reliability.

8. Conclusion

The existing constraint propagation mechanism in Apache Spark's Catalyst optimizer, while beneficial, suffers from performance and scalability issues due to permutational computation of constraints. The newly proposed algorithm addresses these challenges by eliminating redundant inferences and employing an efficient propagation strategy. The implementation demonstrates significant improvements in compilation time and memory usage, enhancing Spark's capability to handle complex queries in large-scale data processing environments.

9. References

- SPARK-33152: Constraint Propagation code causes OOM issues or increasing compilation time to hours
- Apache Spark Catalyst Optimizer Documentation
- YouTube Presentation: Optimizing Constraint Propagation in Apache Spark

Note: Diagrams and code snippets illustrating the algorithm's workflow and implementation details can be included in the final document to provide visual clarity and practical examples.