TECHNICAL WHITE PAPER

KwikQuery TabbyDB

Beyond Spark's Built-in Optimizer:

Targeted Common Subexpression Elimination and Immutable Projection Blocks

Abstract

Apache Spark's Catalyst optimizer employs a rule-based, batch-oriented pipeline that, while highly effective in general, exhibits two related inefficiencies when processing queries containing complex, deeply nested expressions. First, Spark's CollapseProject rule can produce a single flattened project that replicates identical subexpressions across multiple Alias definitions—and existing Common Subexpression Elimination (CSE) logic cannot correct this when the project was never the product of collapsing in the first place. Second, although Spark tracks per-rule plan modifications to avoid redundant re-application, the batch iteration model renders this tracking largely ineffective in practice: any modification by any rule within a batch triggers a full re-run of all rules in that batch, including those whose application would leave the plan unchanged.

This paper describes the optimizations implemented in TabbyDB, a high-performance Apache Spark fork developed by KwikQuery, that address both problems through a unified mechanism: a controlled, semantics-preserving expansion of collapsed projections combined with an immutability guarantee that permits expensive optimizer rules to be applied exactly once to stable subplan regions, regardless of modifications elsewhere in the plan tree.

1. Introduction

Modern analytical queries executed on Apache Spark frequently involve complex transformations: multi-level projections containing conditional logic, arithmetic chains, type casts, date manipulations, and deeply nested function calls. These expressions appear throughout the query plan as children of Project and Filter operators. As the Catalyst optimizer processes such plans, its rule-based transformations interact in ways that can introduce subtle but meaningful inefficiencies—particularly around the evaluation of repeated subexpressions and the repeated traversal of stable portions of the plan tree.

TabbyDB extends Spark's Catalyst optimizer with two complementary techniques that address these inefficiencies. The first technique—controlled projection

expansion with deterministic CSE—ensures that identical subexpressions within a collapsed project are evaluated at most once, filling a gap in Spark's own CSE logic. The second technique—immutable projection blocks with targeted rule skipping—ensures that expensive optimizer rules performing full plan tree traversals are applied only once to regions of the plan that are guaranteed not to change, directly mitigating the inefficiency of Spark's batch re-evaluation model.

Both techniques operate conservatively: they are applied only to deterministic expressions, they preserve query semantics exactly, and they impose no observable behavioral changes for end users.

2. Background: Spark's Catalyst Optimizer

Catalyst is Spark's extensible query optimization framework. It operates on a tree of logical plan nodes—each representing a relational operation such as Filter, Project, Join, or Aggregate—and applies transformation rules to simplify, rewrite, and optimize this tree before code generation.

2.1 Rule Batches and Fixed-Point Iteration

Rules are organized into named batches. Each batch is applied iteratively until either a fixed point is reached (no rule in the batch modifies the plan) or a configurable maximum iteration count is exceeded. Rules within a batch are applied in sequence on each pass. Spark maintains a per-rule flag indicating whether the most recent application of that rule produced a change; this flag is intended to allow unchanged rules to be skipped on subsequent passes. In practice, however, this optimization provides limited benefit: the fixed-point check operates at the batch level, so any modification by any rule resets the iteration and all rules in the batch are re-evaluated from the start.

2.2 The CollapseProject Rule

One of Catalyst's most frequently applied rules is CollapseProject, which merges a sequence of stacked Project nodes into a single Project. Collapsing reduces the depth of the plan tree and simplifies downstream rule application. However, it can introduce a problem: when two or more Alias expressions in the collapsed project share a common subexpression, that subexpression is physically duplicated in the tree. Each Alias independently owns its own copy, and the optimizer has no inherent mechanism to recognize the duplication and share the computation.

2.3 Common Subexpression Elimination in Spark

Spark's optimizer includes logic to detect common subexpressions and, in some cases, prevent them from being collapsed together. This logic is most effective when multiple Project nodes exist in the plan and can be kept separate. The gap

arises when a project already arrives in collapsed form—not as the result of CollapseProject acting on multiple nodes, but as a single complex project from the outset. In this case, Spark's CSE has no multi-project structure to work with and cannot split or expand the existing project. The replicated subexpressions are left in place and evaluated redundantly at runtime.

3. Problem Statement

3.1 Redundant Subexpression Evaluation in Collapsed Projects

Consider a query in which a single Project node contains several Alias expressions, two or more of which reference the same non-trivial subexpression—for example, a conditional date calculation or a multi-step arithmetic formula. After CollapseProject runs, these aliases exist as siblings within a single Project node. Each alias independently traverses and evaluates its own copy of the shared subexpression during the physical execution phase.

The runtime cost is proportional to the number of replications multiplied by the per-row evaluation cost of the subexpression. For expressions involving conditional logic, string manipulation, type conversion, or user-defined functions, this cost can be substantial. For large fact tables—the primary workload in analytical query engines—the cumulative effect across all rows is significant.

> *Key gap: Spark's CSE cannot split or expand an already-collapsed project.*
>
> *If a complex project arrives in collapsed form, Spark has no basis on which to perform subexpression sharing, and the redundancy persists into physical execution.*

3.2 Inefficiency of Per-Rule Change Tracking Under Batch Iteration

Spark's optimizer attempts to reduce wasted work by tracking, for each rule, whether its last application produced a change to the plan. The intention is that an unchanged rule need not be re-applied on the next batch pass. However, this optimization is largely negated by the batch execution model.

Because batch iteration restarts from the first rule whenever any rule in the batch produces a change, a single productive rule application causes all preceding rules to be re-run—regardless of whether those rules would find anything to do. In a batch containing many rules, only one of which is productive on a given pass, the remaining rules perform complete plan tree traversals and return without making changes. For expensive rules that traverse the entire logical plan—such as NullPropagation, ConstantFolding, and similar inference-based simplifications—

this repeated traversal represents wasted optimizer compilation time that scales with plan complexity.

> *Key gap: Per-rule change tracking cannot prevent re-traversal when any sibling rule in the same batch is productive. Expensive rules traverse the full plan tree on every batch iteration pass, even when their target subplan has not changed.*

4. TabbyDB's Solution: Controlled Expansion with Immutable Blocks

TabbyDB addresses both problems through a single unified mechanism consisting of three phases. The core insight is that the two problems are connected: the same projection structure that benefits from CSE expansion also benefits from being treated as immutable after its first optimization pass, thereby solving the redundant rule-traversal problem at the same time.

Phase 1: Complete Collapse

In the first pass of the relevant optimizer batch, TabbyDB permits project collapsing to proceed to completion, just as standard Spark would. This produces the minimal plan tree with the smallest possible number of nodes. The trade-off—replicated subexpressions within the collapsed project—is accepted at this stage as a necessary intermediate state, because a fully collapsed project provides the clearest starting point for subsequent analysis.

Phase 2: Controlled Expansion with Deterministic CSE

After the first pass collapses projects fully, TabbyDB performs a controlled, targeted expansion of the resulting project. It traverses the Alias expressions in the collapsed project and identifies subexpressions that appear more than once and are deterministic—that is, they produce the same output for the same input on every invocation, with no dependence on external state, random functions, or side effects. For each such replicated deterministic subexpression, TabbyDB rewrites the project into a chain of projects: the first project in the chain computes the subexpression and assigns it an intermediate Alias; subsequent projects in the chain reference this Alias rather than recomputing the subexpression independently. This guarantee—that every deterministic replicated subexpression is computed exactly once and referenced by all consumers—holds regardless of how complex or deeply nested the original expressions are. The expansion is not a heuristic; it is a complete, exhaustive rewrite for all qualifying subexpressions.

Phase 3: Targeted Rule Application and Immutability Marking

Once the chain of expanded projects is established, a carefully selected set of expensive optimizer rules is applied to these projection blocks exactly once. This set comprises rules that perform full logical plan tree traversal—NullPropagation, ConstantFolding, and related inference-based simplifications among them. After this first application, the expanded projection blocks are marked as immutable.

The immutability guarantee has a precise meaning: the content of these marked blocks will not be changed by any subsequent application of the designated rule set, even if other parts of the plan tree are modified by other rules in the same batch. This guarantee is safe because the blocks contain only deterministic expressions that have already been fully normalized by the first rule pass. There is no new information that subsequent passes could act upon.

On all subsequent batch iterations, the optimizer detects the immutability marker on these blocks and skips the designated expensive rules for their contents entirely. The rest of the plan tree continues to be optimized normally and without restriction. Rules outside the designated expensive set are not affected and continue to operate on all nodes as usual.

5. How the Two Problems Are Addressed

5.1 Common Subexpression Elimination

The controlled expansion in Phase 2 directly resolves the CSE gap described in Section 3.1. By operating on the fully collapsed project itself—rather than relying on a pre-existing multi-project structure—TabbyDB's approach handles the case that Spark's CSE misses. The result is a chain of projects in which each distinct deterministic subexpression appears in exactly one place, and all references to that subexpression resolve through an Alias rather than through independent recomputation. At runtime, each row passes through this chain of projects and each subexpression is evaluated exactly once, regardless of how many downstream Aliases consume it.

5.2 Optimizer Batch Re-evaluation Overhead

The immutability marking in Phase 3 directly resolves the batch re-evaluation inefficiency described in Section 3.2. Even when a productive rule elsewhere in the batch causes the full batch to restart, the expensive rules in the designated set encounter the immutability markers on the expanded projection blocks and skip their traversal immediately. The cost of these rules, for the immutable region, drops to the cost of a single marker check—effectively zero—for all passes after the first. The immutability guarantee means this skipping is always safe: the blocks

were fully normalized on the first pass and cannot be further simplified by those rules.

6. Correctness and Safety

Both optimizations operate with conservative safety bounds:

- Determinism requirement: Only subexpressions that are provably deterministic—those producing the same result for the same input on every call—are eligible for CSE expansion or immutability marking. Non-deterministic functions (such as rand(), uuid(), now(), and user-defined functions not declared deterministic) are excluded and continue to be evaluated on every reference, as the semantics of the original query demand.
- Semantic preservation: The controlled expansion rewrites the project chain in a semantics-preserving manner. The same output columns are produced with the same values for every input row. No column is renamed, reordered, or altered as a result of the transformation.
- Immutability scope: The immutability marker applies only to the designated set of expensive traversal rules. It does not suppress rule application globally. Rules that modify plan structure—such as join reordering or predicate pushdown—are unaffected. The marker is therefore not a blanket optimization fence; it is a targeted skip mechanism for a known-safe subset of rules on a known-stable region of the plan.
- No user-visible behavioral change: The optimizations are transparent to Spark users, DataFrame API consumers, and SQL query authors. Query results are identical to those produced by an unoptimized plan.

7. Performance Implications

The combined effect of these two optimizations is most pronounced in the following workload characteristics:

- Queries with wide, complex projections: Queries that project many output columns from computationally expensive subexpressions—particularly those sharing intermediate calculations—see the greatest benefit from CSE expansion. The more times a subexpression is replicated across aliases, the greater the per-row saving at execution time.
- Large fact tables: The per-row saving from CSE elimination scales directly with row count. For the multi-billion-row fact tables typical in enterprise analytical workloads, the cumulative saving is substantial.
- Long optimizer batch chains: Query plans that require many batch iterations to reach a fixed point benefit most from immutability-based rule skipping, since the number of avoided traversals is proportional to the number of iterations.
- Plans with many expensive rules: Workloads that activate many inference-based optimizer rules—NullPropagation, ConstantFolding, and related simplifications—benefit directly from the reduction in traversal cost per iteration.

8. Conclusion

TabbyDB's controlled projection expansion and immutable block mechanism demonstrate that meaningful performance improvements can be extracted from Apache Spark's optimizer pipeline through targeted, semantics-preserving extensions—without changes to the public API, query language, or execution model.

By addressing the CSE gap for already-collapsed projects and the batch re-evaluation overhead for stable plan regions as a single unified problem, TabbyDB delivers improvements at both the runtime execution layer (fewer subexpression evaluations per row) and the optimizer compilation layer (fewer full-tree traversals per optimization pass). Both improvements compound with query and plan complexity, making them particularly valuable for the large-scale, expression-heavy analytical workloads that represent the core of enterprise data processing on Spark.

These optimizations are part of TabbyDB's broader effort to expose and resolve the performance headroom that exists within Spark's optimizer architecture—delivering faster query compilation and faster query execution without sacrificing correctness, compatibility, or operational simplicity.

