Extensible Rule Language for Query Optimizer

Sicheng Pan

Extensible Rule Language for Query Optimizer

by Sicheng Pan

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

[illegible]

Professor Alvin Cheung
Research Advisor

May 17, 2024

(Date)

* * * * * * *

[illegible]

Professor Joseph Hellerstein
Second Reader

May 7, 2024

(Date)

Extensible Rule Language for Query Optimizers

Sicheng Pan
University of California, Berkeley
pansicheng@berkeley.edu

ABSTRACT

A query optimizer searches for an efficient query plan within the space of its alternatives using a search algorithm. The scope of the search space used to be defined as part of the search algorithm, but the need to incorporate new optimization techniques constantly reshapes the search space, and this brings development challenges to correctly maintain the search algorithm while supporting the new optimization techniques.

The extensible query optimizer architecture is proposed to address this challenge by isolating the definition of search space from the search algorithm. Within this architecture, developers implement independent rewrite rules that describe equivalence relations among query plans, and the search space for the query optimizer is implicitly defined by the transitive closure of these equivalence relations. An extensible query optimizer explores the search space by repeatedly applying the rewrite rules it knows. To support a new optimization technique, developers only need to implement the corresponding rewrite rule for the query optimizer, without worrying about any changes to the search algorithm.

This architecture is widely adopted in the designs of modern data pipelines because it simplifies the process of extending a query optimizer with new optimization techniques. However, the growth in the complexity of rewrite rules continues to bring development challenges in terms of implementation correctness and cost.

In this paper, we present RuleScript, a domain-specific language designed for expressing rewrite rules in an extensible query optimizer. Rules expressed in RuleScript are automatically verified for correctness, and they can be used to generate implementations of rewrite rules to be invoked by extensible query optimizers. Moreover, RuleScript can be extended to support custom symbols in rewrite rules, which are defined by users to represent query plan components specific to the backend query engine.

We have implemented a prototype of RuleScript in Java, and performed case studies with this implementation. We have verified rewrites rules in Apache Calcite and generated implementations that can be plugged into Apache Calcite codebase to illustrate the usability of our design.

1 INTRODUCTION

Query processing is the task of processing data in a way that matches the specifications declared in query languages by users. Query optimization is central to this process, because a query optimizer can find an efficient query plan within the space of semantically equivalent counterparts, and this could significantly reduce the runtime cost for query processing.

The pioneering work from the System R project [1, 10] establishes a query optimizer framework, but it is hard to scale with the expansion of the complexity in query optimization over time. More than a decade after the System R project, the extensible optimizer architecture [6, 7] is proposed to address this need. An extensible query optimizer separates the task of enumerating execution plans from the task of finding suitable candidates for enumeration. It provides an abstraction for developers to specify such candidates via rewrite rules, which describe the equivalence relations among execution plans. To incorporate new optimization techniques, developers only need to implement the corresponding rewrite rules for the extensible query optimizer, without worrying about modifying the search algorithm that enumerates query plans. This architecture has been widely adopted today for various data systems, including relational database management systems (RDBMSs) like Apache Calcite [3] and CockroachDB [11], as well as modern data warehouse systems like Snowflake [5].

Yet, the complexity of query optimization continues to scale with the invention of new optimization techniques. Within the extensible query optimizer, this is reflected by the ever-increasing number of rewrite rules. For example, Apache Calcite has more than one hundred rewrite rules, where each rule is implemented with more than two hundred lines of Java code on average. Meanwhile, CockroachDB has more than two hundred rewrite rules, where each rule is implemented with more than thirty lines of CockroachDB DSL (a language designed by CockroachDB to describe its rewrite rules) on average. The complexity of new rewrite rules is also growing both syntactically and semantically.

One major concern for query optimization is the correctness of the query optimizer and its results: the optimized query plan must produce the same results as the original query plan under all possible input relations. The correctness of implemented rewrite rules is critical to the correctness of query optimization, but the new rewrite rules are becoming more tricky to correctly implement from the first attempt. Numerous failures have been spotted for these rewrite rules in the production environment, and such failures are later converted into test cases for these rewrite rules [13]. If the query optimizer can only generate equivalent alternatives for the original query plans, then we do not need to worry about such failures.

These concerns can be addressed by automated query equivalence solvers, which can provably verify equivalent queries. Naively, we can verify the equivalence of each optimized query plan and their corresponding unoptimized plan with these tools, and only execute the optimized plan when this check is successfully passed. This can eliminate any potential faulty execution. Recent progress in this field improves the feasibility of this naive solution. SPES can prove 98 out of 444 rewrites in Apache Calcite's test cases, while QED can prove 293 of them . It seems promising that we will eventually have a powerful solver that can verify the equivalence of any query plan used in practice.

Yet, as of today, it is not very practical to deploy these solvers in production. First, the fragment of query plans that they can support is still limited, which precludes a considerable amount of rewrite rules. Second, they could bring overheads that scale with the complexity of queries, because the query engine has to wait for the verification from these solvers before execution. Modern tools like SPES and QED reduce query equivalence verification into SMT problems, and the complexity of the generated SMT statements grows with the complexity of the queries. This could lead to considerable growth in the runtime of underlying SMT solvers such as CVC5.

A better way to solve this problem is directly verifying rewrite rules themselves, rather than the rewritten query plans from the application of these rewrite rules. The first step toward this goal is made by HoTTSQL, a domain-specific language (DSL) based on SQL that could express the semantics of query plans [4]. It supports a few generic components such as generic schemas, and they could be used to express rewrite rules that are applicable to the corresponding subset of query plans. It is integrated into the Cosette solver so that query plans declared in this DSL can be checked for equivalence. However, the lack of support for SQL features such as Null and primary keys restricts the coverage of SQL features for HoTTSQL, and it does not address the problem of correctly implementing a rewrite rule after Cosette shows that the rewrite rule maintains the semantics of the query plan.

In comparison, CockroachDB also has its own DSL to describe the rewrite rules in its extensible query optimizer based on relational operators, with which it can automatically generate the implementations [15]. However, this DSL can only be used within CockroachDB's query optimizer due to its focus on production needs, and we do not know how to verify the correctness of the rewrite rules involving the CockroachDB-specific query plans.

In this paper, we present RuleScript, an extensible rule language that is both verifiable and can be used for code generation. We design the core syntax of RuleScript based on the structure of query plans, but unlike concrete query plans derived from executable SQL statements, the query plans constructed in RuleScript can involve *uninterpreted symbols*, such as uninterpreted types and uninterpreted functions. Uninterpreted symbols can be viewed as variables that are replaceable by any concrete value of the same type. For example, an uninterpreted predicate function can be replaced with any boolean expression constructed from the arguments of the uninterpreted function and boolean operators. During verification, RuleScript checks for all possible assignments for these variables, and the verification cannot pass if it fails on any particular assignment of these variables. This allows RuleScript users to succinctly express the family of all concrete query plans that one rewrite rule can apply and how it is applied.

A rewrite rule in RuleScript contains two RuleScript query plans, named the match pattern and the transform pattern, respectively. In order to apply a rewrite rule in RuleScript on a concrete query plan, an appropriate instantiation of the uninterpreted symbols must be found so that the match pattern can be evaluated to the concrete query plan. If such an instantiation exists, the concrete query plan can be transformed to the transform pattern where all uninterpreted symbols in the transform pattern are evaluated under the same instantiation.

RuleScript is also extensible for new query plan components, such as engine-specific query plans and scalar expressions, as long as users can reduce these new query plan components to the core syntax we already defined in RuleScript. These custom query plans can be used to represent the custom implementations in the query engine. We also provide meta-variables in RuleScript, which are placeholders in a RuleScript query plan, so that users can easily create similar rewrite rules by specifying the valid assignments of these meta-variables.

To verify the rewrite rule, RuleScript treats the match pattern and the transform pattern as two abstract query plans and sends them to QED, a solver that checks for the equivalence for all possible interpretations of the uninterpreted symbols. The custom query plans defined by users will be reduced to query plans constructed using the core syntax, which can be understood by QED.

To generate the implementation for rewrite rules defined in RuleScript, so that users can run these rewrite rules in any query engine they like, RuleScript asks users to provide an *adapter* to the target query engine, which contains the necessary information to generate the compatible implementation. For each rewrite rule in RuleScript, the adapter recursively traverses the match pattern and the transform pattern, and then it composes the code chunks together into the implementation for the rule.

In summary, we make the following contributions for RuleScript:

- We design RuleScript based on relational operators under bag semantics. RuleScript models a rewrite rule using a match pattern and a transform pattern with uninterpreted symbols, which succinctly represent the collection of concrete query plans where the rewrite rule can apply. Moreover, we verify the correctness of the rewrite rules in RuleScript using the QED solver, which supports the reasoning of uninterpreted symbols.
- We allow RuleScript to be extended with any custom query plan components that are to be used in the rewrite rule, and users can specify their semantics based on the core syntax in RuleScript so that they can be correctly comprehended by QED in the verification process. We provide meta-variables to facilitate the process of composing rewrite rules in RuleScript.
- We design a pipeline to perform code generation for any rewrite rule defined in RuleScript with modular and engine-specific information provided by users.
- We verify the rewrite rules in Apache Calcite with RuleScript and generate working implementations that are compatible with Apache Calcite codebase with the code generation pipeline.

2 OVERVIEW

The motivation for our work comes from the difficulty of correctly implementing a rewrite rule without being too restrictive about when it could apply. We will illustrate this with the Join Associate rule, which is a very fundamental rewrite rule describing the associativity of the Join query plan, and can be found in most extensible query optimizer implementations, such as Apache Calcite [12].

The Join Associate permutes the order of two consecutive Joins in a query plan. The SQL pattern corresponding to the query plans where this rewrite rule applies is illustrated in Listing 1.

```
SELECT * FROM (
    SELECT * FROM Q0 INNER JOIN Q1
        ON P0(Q1) AND P1(Q0, Q1)
) INNER JOIN Q2 ON P2(Q1, Q2) AND P3(Q0, Q1, Q2);
```

Listing 1: The match pattern for the Join Associate rule

For the SQL pattern in Listing 1, `Q0, Q1, Q2` are query plans used as the inputs to the pattern, and `P0, P1, P2, P3` are RuleScriptJoin predicates where their arguments specify which input query plans they depend on. In words, this pattern matches any two consecutive Inner Joins where the innermost Join predicate is logically equivalent to the conjunction of `P0, P1` for some `P0, P1`, and the outermost Join predicate is logically equivalent to the conjunction of `P2, P3` for some `P2, P3`. If a query plan is matched by this pattern, the Join Associate rule will transform it according to the SQL pattern shown in Listing 2, where `Q0, Q1, Q2` and `P0, P1, P2, P3` are the same as those in the matched query plan.

```
SELECT * FROM Q0 INNER JOIN (
    SELECT * FROM Q1 INNER JOIN Q2
        ON P0(Q1) AND P2(Q1, Q2)
) ON P1(Q0, Q1) AND P3(Q0, Q1, Q2);
```

Listing 2: The transform pattern for the Join Associate rule

While the rewrite in the other direction also works, for simplicity we restrict our discussion to the current scenario. Notice that the pattern requires both Joins involved to be Inner Joins. Apache Calcite developers express concerns about whether this limitation is too strict in their implementation for this rule [12]. Specifically, they wonder if they can rewrite any two consecutive Joins similarly, even if they are not both Inner Joins. We can solve this problem with RuleScript, and we will show that the Join Associate rule is indeed applicable to other types of Join, such as the case where both Joins are Outer Joins.

This brings a few challenges that could not be addressed with prior work such as HoTTSQL and Cosette. First, the only type of Join supported by HoTTSQL is Cross Join, which can only be used to model Inner Join with an additional Filter operator. Thus HoTTSQL cannot help us determine whether the rewrite is correct if Left Joins, Right Joins, or Outer Joins are involved.

If it turns out that a similar rule works for other Join types, developers can manually modify their existing implementations for the Join Associate rule, or even add a new rewrite rule for this case. However, both options are inefficient since developers have to either maintain the correctness of the original rewrite rule during modification or duplicate existing codes for a new rule. RuleScript streamlines this process by automatically generating the implementation based on the verified rewrite rule. In addition, since the Join Associate rule is very generic and does not depend on any specific query engine, RuleScript has a general way of performing code generation such that the implementation for this rule can be derived for different query engines, as long as we have sufficient information about internal query optimizer. Specifically, RuleScript needs its users to provide an adapter to the target query engine, which defines the procedure to generate code for each type of symbol used in the rewrite rules.

RuleScript solves the challenges mentioned above with its framework for rewrite rule verification and code generation, as illustrated in Figure 1. To showcase how this works on the motivation example, we first define the Join Associate rule in RuleScript. Here we only provide the high-level overview of what needs to be specified with pseudo-code in Listing 3, and we explain it line by line below.

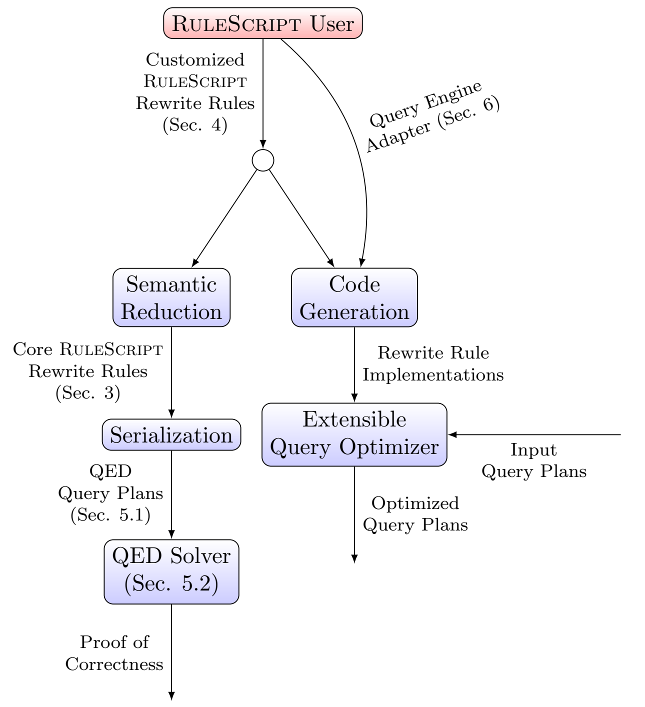
RuleScript User
Customized RuleScript Rewrite Rules (Sec. 4)
Query Engine Adapter (Sec. 6)
Semantic Reduction
Code Generation
Core RuleScript Rewrite Rules (Sec. 3)
Rewrite Rule Implementations
Serialization
Extensible Query Optimizer
Input Query Plans
QED Query Plans (Sec. 5.1)
Optimized Query Plans
QED Solver (Sec. 5.2)
Proof of Correctness

Figure 1: Components of the RuleScript framework. The left branch handles the verification of rewrite rules in RuleScript, while the right branch handles the automatic generation of their implementations.

```
01 Q0       =   Plan(R0, [Field(x, T0)])
02 Q1       =   Plan(R1, [Field(y, T1)])
03 Q2       =   Plan(R2, [Field(z, T2)])
04 F0       =   P0(x, y) AND y IS NOT NULL
05 F1       =   P1(y, z) AND y IS NOT NULL
06 before   =   Join(Outer, F1, Join(Outer, F0, Q0, Q1), Q2)
07 after    =   Join(Outer, F0, Q0, Join(Outer, F1, Q1, Q2))
08 rewrite  =   Rule(before, after)
```

Listing 3: The Join Associate rule in RuleScript

Line 1-3 define `Q0, Q1, Q2`, which are generic RuleScript query plans named `R0, R1, R2` respectively. `x, y, z` are the attribute names and `T0, T1, T2` are the types for these attributes. Moreover, `T0, T1, T2` are uninterpreted, which means that they can represent any type, such as the output schema of a concrete query plan. Thus although there is only one field in each of `Q0, Q1, Q2`, they can represent any concrete query plans. They are part of the RuleScript core syntax, which we will describe in Sec. 3.

Line 4-5 define `F0, F1`, which are RuleScript predicate patterns depending on attributes `x, y, z` and uninterpreted predicate symbols `P0` and `P1`. The uninterpreted predicate symbols can represent arbitrary predicate functions. Here we define `F0, F1` using conjunctions of uninterpreted predicates and the assertion `y IS NOT NULL`, which suggests that `F0, F1` are arbitrary predicates that return `FALSE` when `y` is `NULL`.

Line 6-7 define `before, after`, which are the match pattern and the transform pattern describing what query plans can be applied with the rewrite rule and how they are transformed. They represent consecutive Outer Joins constructed from input query plans `Q0, Q1, Q2` and predicates `F0, F1`. Finally, line 8 defines a rewrite rule `rewrite` with `before, after`. The rewrite rule `rewrite` can be further used for verification or code generation.

Notice that the Join predicates `F0, F1` in Listing 3 are different from those in Listing 1 and Listing 2. We cannot simply create a new rewrite rule by replacing the Inner Joins in Listing 1 and Listing 2 with Outer Joins because such a rewrite rule can produce invalid rewrites that change the semantics of the query plan. For example, consider the case in Listing 4, where `R, S, T` are tables containing a single integer column, named `x, y, z` respectively. Moreover, `R.x` contains a single value `0`, while both `S.y` and `T.z` contain a single value `1`.

```
-- Before Rewrite
SELECT * FROM (
    SELECT * FROM R FULL OUTER JOIN S ON x = 0
) FULL OUTER JOIN T ON z = 0;
-- After Rewrite
SELECT * FROM R FULL OUTER JOIN (
    SELECT * FROM S FULL OUTER JOIN T ON z = 0
) ON x = 0;
```

Listing 4: An incorrect application of Join Associate rule

If we continues to use the Join predicates in Listing 1 and Listing 2 for the Outer Join case, the rewrite in Listing 4 occurs because `P1` is `x = 0` and `P2` is `z = 0`, while `P0` and `P3` always return `TRUE`. The query plan before the rewrite outputs two rows, while the query plan after the rewrite outputs three rows, which suggests that the rewrite changes the semantics of the query plan. To avoid such a situation, we use more restricted Join predicates in Listing 3 so that the restricted rewrite rule cannot be applied to the example above, as we cannot match `y is NOT NULL` is not implied by `x = 0` or `z = 0`.

To verify the restricted rewrite rule, RuleScript will translate the match pattern and the transform pattern in the rewrite rule `rewrite` into the input format for QED, including the uninterpreted predicates `P0, P1` and the uninterpreted types `T0, T1, T2`. The QED solver will show that the match pattern and the transform pattern in Listing 3 are provably equivalent for any input relations. We describe this process in Sec. 5.

RuleScript then passes the verified rewrite rule to an adapter implementation for the target query engine provided by the user, which recursively traverses the rewrite rule and keeps track of relevant information in a context object. Depending on the target query engine, the context object may contain different things. For example, if the target query engine is Apache Calcite, the context object could contain Java code snippets, as is shown in Sec. 7. The adapter spits out the generated implementation from the final context object. In Sec. 6, we will explain how this can be achieved for RuleScript, and in Sec. 7 we will also present the generated code for the rewrite rule we defined in Listing 3.

3 THE CORE LANGUAGE

We now present the core syntax for RuleScript, which is the basis for all rewrite rules expressed in RuleScript. We will provide an overview of its syntax in Sec. 3.1, and then in Sec. 3.2 we will discuss how to interpret the rewrite rules from RuleScript.

3.1 Syntax

We deliberately choose to design RuleScript to mimic the syntax of a concrete query plan, because such syntax clearly describes the query plans that will be matched and transformed into a rewrite rule. We provide its syntax in Listing 5. To distinguish a query plan in RuleScript from an actual query plan, we refer to the former as a pattern.

A rewrite rule in RuleScript has three parts: a match pattern that describes what query plan the rule should apply, a transform pattern that describes how to transform the matched query plan, and optional extra constraints that guarantee the correctness of the rewrite. Such construction is captured by the `<Rule>` symbol in Listing 5, where the match pattern is represented by the first `<Patn>` and the transform pattern is represented by the second `<Patn>`. The constraint in a `<Rule>` is any first-order logic statement `<FOPred>` where all variables involved are quantified by `FORALL` or `EXISTS`. In this paper we may omit this field when the constraint is not necessary, or equivalently when the constraint is trivially `TRUE`. Constraints may help when users would like to restrict the uninterpreted symbols in the rewrite rule. For example, RuleScript users may want to define an uninterpreted injective projection function, and this can be achieved with a first-order logic statement regarding the uninterpreted function symbol.

In RuleScript, uninterpreted symbols can be viewed as variables. During verification RuleScript checks for the correctness of the rewrite under all possible assignments of these variables, and we know that at runtime the rewrite rule should only apply to query plans derived from assignments of these variables. There are three types of uninterpreted symbols in RuleScript. The first one is uninterpreted `<Type>` symbols, which represent any value type. The second one is uninterpreted `<Op>` symbols, which represent any function mapping. The third one is the `Plan` patterns, which represent any concrete query plan subject to potential constraints like uniqueness. In RuleScript, we have two assumptions for any type. First, the type must satisfy the theory of equality and uninterpreted function, which means we can perform equivalence checking for values in this type and describe an uninterpreted function using this type. Second, there exists a special value named Null inside this type. Types with the same name are considered identical, and this is also true for functions.

```
<Rule>   :=  Rule(<Patn>, <Patn>, <FOPred>)
<Patn>   :=  Plan(<Id>, [<Field>])
          |  Empty([<Field>])
          |  Filter(<Pred>, <Patn>)
          |  Project([<Call>], <Patn>)
          |  Join(<J_ty>, <Pred>, <Patn>, <Patn>)
          |  Union(<Patn>, <Patn>)
          |  Aggregate(<Call>, <Call>, <Patn>)
          |  Distinct(<Patn>)
<FOPred> :=  FORALL [<Id>] <Pred>
          |  EXISTS [<Id>] <Pred>
<Id>     :=  Id(<String>)
<Field>  :=  Field(<Id>, <Type>, <Stat>)
<Pred>   :=  <Call> | TRUE | FALSE | NOT <Pred>
          |  <Pred> AND <Pred> | <Pred> OR <Pred>
          |  <Call> IS NOT NULL | <Call> IS NULL
          |  <Call> = <Call> | <Call> != <Call>
<Call>   :=  <Id> | <Lit> | <Op>([<Call>])
<J_ty>   :=  "Inner" | "Left" | "Right"
          |  "Outer" | "Semi" | "Anti"
```

```
<Stat>    :=  Stat({nullable: <Bool>,
                   unique: <Bool>, ...})
<Op>      :=  Op(<Id>, <Type>, <Type>)
<Alias>   :=  PatnAlias(<Id>, <Patn>)
           |  FOPredAlias(<Id>, <FOPred>)
           |  PredAlias(<Id>, <Pred>)
           |  CallAlias(<Id>, <Call>)
           |  JtyAlias(<Id>, <J_ty>)
```

Listing 5: Syntax for the RuleScript. A pair of curly braces represent a dictionary of values, and a pair of square brackets represent an ordered list of zero or more items.

3.2 Interpretation

With the syntax of RuleScript shown in Sec. 3.1, we now explain how to interpret `<Rule>` as a rewrite rule in an extensible query optimizer. The first-order logic statement `<FOPred>` in `<Rule>` specifies the conditions that the uninterpreted symbols have to satisfy, such as the injectiveness of uninterpreted projection functions. The match pattern and the transformed pattern in are RuleScript query plans (i.e., pattern), which can be viewed as partially filled query plans that can be turned into concrete query plans once there is an assignment for all uninterpreted symbols involved. We can view the application of a rewrite rule as an instantiation of the uninterpreted symbols in the match pattern followed by the evaluation of the transform pattern under the same instantiation. Thus we need to treat match patterns and transform patterns differently, even if they are both syntactically RuleScript query plans. We investigate each variant of the pattern below to illustrate this, using the syntax for concrete query plans to be discussed in Sec. 5.1.

3.2.1 `Plan` and `Empty`.

When we view a pattern as an abstract syntax tree, then `Plan` and `Empty` are the only choices for its leaf nodes, which specify how to match concrete query plans based on their output schema and where the matched query plan should be used during the application of the rewrite rule. In both variants, we need to describe the information about the schema, which is represented by an ordered list of `<Field>` symbols.

Each `<Field>` has three parts, the first is the name of the field, the second is the value type of the field, and the third one is a `<Stat>`, which is an extensible dictionary containing useful information about the data for this field. Each `<Stat>` contains two Booleans describing the nullability and the uniqueness of the data for the field it is attached to, and it can be extended for other data constraints. The additional constraints can be encoded as first-order logic assertions. In this paper, we omit the `<Stat>` in a `<Field>` when no requirement is specified for the data within (i.e., the data is nullable, not unique, and have no additional constraint).

A `Plan` represents any query plan satisfying two conditions. First, there exists an instantiation of the uninterpreted types used in the `Plan` so that the product of all field types in the pattern is equal to the output type of the query plan. Second, the instantiation preserves the properties specified by `<Stat>` for each field in the `Plan`. We define two tuples of the same type to be equal if their attributes are pairwise equal, and we define a tuple to be Null if any of its elements is Null. In this way, we can define the uniqueness and nullability of fields with product types accordingly. In addition, `Plan` is labeled by an identifier `<Id>`.

For example, let R be a physical table containing a single column c that contains unique and nullable integers. Then the query plan Project$((c, c+1)$, Table$(R))$ is a valid interpretation of `Plan(R, [(c, T, <Stat>)])` where `<Stat>` requires the field to be unique because we can instantiate the uninterpreted type `T` to be Integer × Integer.

When a `Plan` appears in a match pattern, it describes the procedure to verify whether the query plan is one of its valid interpretations by finding an appropriate instantiation of the uninterpreted types satisfying the properties in `<Stat>`, and if so we call it a match. When a `Plan` appears in the transform pattern, it refers to the actual query plan matched by the same `Plan` in the match pattern.

An `Empty` is a `Plan` that does not output any tuples, but we still need to specify the type of its fields. In reality, it will match against and transform into empty Value query plans (i.e. query plans without any outputs) with the corresponding schema. Unlike `Plan`, we do not need to label `Empty` because it is always empty.

3.2.2 `Union` and `Distinct`.

Except for `Plan` and `Empty`, all other variants of the pattern are recursively defined. `Union` and `Distinct` are the straightforward ones among these variants, as they can be constructed from input patterns without other kinds of arguments.

If they appear in a match pattern, they represent the procedure to verify if the top-level query plan shares the same kind (e.g. a `Distinct` pattern and a Distinct query plan) and then verify the inputs of the query plan against their input patterns. If they appear in a transform pattern, they represent the procedure to construct the same kind of query plan after recursively constructing the query plans from the input patterns.

3.2.3 `Project` and `Aggregate`.

In addition to patterns in their arguments, `Project` and `Aggregate` contain `<Call>`s which require some extra care.

Each `<Call>` in the pattern can be an identifier `<Id>` referring to one of the columns in the input pattern, a literal value, or a function call whose arguments are an ordered list of `<Call>`. The function call operator `<Op>` is specified by an identifier, the input value type (i.e. the product of all individual argument types), and the output type. `<Call>` can also appear directly in `<Rule>` as part of `<FOPred>`, in which case the identifier must refer to one of the bounded variables.

A `<Call>` in a `Project` can be interpreted as any expression that we can obtain by instantiating the uninterpreted types and then the function operators. Instantiating an uninterpreted function operator requires us to express the output given input arguments using operators from concrete types (e.g. + for numerical types) for an instantiation of uninterpreted types.

A `Project` contains zero or more `<Call>`s in an ordered list beside its input pattern, and it is interpreted as the collection of Project query plans satisfying two conditions. First, the source input of the Project plan is a valid interpretation of the input pattern. Second, there exists a way to instantiate the uninterpreted types and operators so that the product of `<Call>`s in the list (i.e. merge the expressions into a single tuple) is equivalent to the projection function in the Project query plan.

If a `Project` appears in a match pattern, it describes the procedure to recursively verify the input of a Project query plan against the input pattern and then verify if the projection function in the Project query plan can be segmented into a list of valid interpretations for the `<Call>`s. If this is successful, we obtain a collection of valid interpretations for the uninterpreted function operators. If a `Project` appears in a transform pattern, it describes the procedure to construct a Project query plan after recursively constructing its input query plan from the input pattern and then the projection function with the interpretations of the function operators involved.

A `Aggregate` contains an aggregation, which is represented by the first `<Call>`, and a group, which is represented by the second `<Call>`, in addition to the input pattern. We need to be aware that the function signatures of aggregation function operators are different, as the input type is a bag of values. An uninterpreted aggregation function operator can be instantiated by expressing the output using aggregation functions from concrete types (e.g. SUM for numerical types), while an uninterpreted group function operator is just a normal function operator. Then we can define the rule semantics of `Aggregate` similar to `Project`.

3.2.4 `Filter` *and* `Join`.

Similar to `Project` and `Aggregate`, `Filter` and `Join` have additional arguments. However, `<Pred>` is slightly different from `<Call>`, as `<Pred>` can only be interpreted as a scalar boolean expression obtained by instantiating the uninterpreted types and then the function operators.

A `Filter` contains one `<Pred>` besides an input pattern. It can be interpreted as any Filter query plan where its input query plan is a valid interpretation of the input pattern, while the predicate expression is equivalent to a valid interpretation of the `<Pred>`. Notice that a valid interpretation of `<Pred>` may not look similar syntactically. For example, `x+y=3 AND x-y=1` is a valid interpretation of `P0(x) AND P1(y)`, as we can instantiate `P0(x)` to `x=2` and `P1(y)` to `y=1`.

In comparison to a `Filter`, a Join has an additional type parameter and an additional input pattern, and we can define its rule semantics similarly.

3.2.5 `<Alias>`.

An `<Alias>` can be used to define custom RuleScript symbols, which can be identified by its identifier `<Id>`. The `<Alias>` symbol will register the corresponding identifier as a valid variant of the input symbol, which can be used in the same way as other variants. When we would like to verify a rule involving an `<Alias>`, we will replace it with its semantics, which is represented by the input symbol. This is important for the extension of RuleScript, which we will discuss in Sec. 4.

3.3 Example

The motivation example, as is shown in Listing 3, provides an example of how to construct a rewrite rule in RuleScript. We first construct the `Plan` patterns as the source inputs. For the Join Associate rule, we move Join predicates around based on their dependencies of the Join inputs. Thus we do not need to further segregate the input schema of the input query plans, and we only need one `<Field>` for each of the `Plan` patterns. Then we construct the two Join predicates, and in this case, we are using custom predicates using `PredAlias`, although we hide the declaration here for simplicity. Then we build the match pattern and the transform pattern from the `Plan`s and custom predicates. Finally, we construct a rule from the match pattern and the transform pattern.

4 EXTENSION

Without the `<Alias>` symbol in Sec. 3, rewrite rules in RuleScript cannot support custom query plans that can be found in most extensible query optimizer implementations. Consequently, it limits the way we can perform code generation, which we will discuss in Sec. 6. Moreover, it is not easy to use to describe families of rules that share similar structures, particularly those involving `Join` s. Thus in this section, we aim to address these issues by extending RuleScript with extra features and explain how they are achieved in our reference implementation.

We choose to implement RuleScript as a library in Java, instead of a custom DSL with its parser and compiler for a few reasons. As a library in Java, RuleScript can be used with an integrated development environment, which can greatly facilitate the usage of RuleScript. Meanwhile, this makes it relatively easy to test our implementation against Apache Calcite, which is also implemented in Java, and we will discuss this in Sec. 7. Moreover, object-oriented programming allowed us to easily extend our codes via class inheritance and interface implementation while implementing a parser and compiler to achieve the same effect requires substantial extra work. We can achieve the same effect of `<Alias>` by defining the corresponding interfaces for symbols in RuleScript. The identifier `<Id>` of the `<Alias>` corresponds to the class name, and the input symbol corresponds to the interface method named `semantics()`, which is a zero-argument method returning a symbol in RuleScript without `<Alias>`. We will discuss these interfaces in Sec. 4.1, after which we introduce additional constructs above these interfaces to facilitate the construction of rewrite rules in Sec. 4.2.

4.1 Alias Interface

If we would like to allow the usage of custom query plans in RuleScript, we need a way to argue for the correctness of rewrite rules where they are involved. Thus we need a way to describe the semantics of these custom patterns. A simple idea is to express the semantics of these custom patterns using the semantics of existing patterns in RuleScript.

In our reference implementation, we define several interfaces for this purpose. The first one is an expression interface, which is used to model scalar symbols in RuleScript such as `<Call>` and `<Pred>`. The second one is a pattern interface, which is used to model custom RuleScript query plans (i.e. `<Patn>`). Both of these interfaces only require one method named `semantics()` to be implemented, which should return the corresponding symbols for the corresponding object without further using custom symbols. We provide the default implementation of `semantics()` for the RuleScript symbols in Sec. 3 except for `<Alias>`, which can be used for any class that intends to implement these interfaces. The constructors of the RuleScript symbol classes can take any object implementing these interfaces as the inputs, which allows users to plug in any custom symbol.

Based on these two interfaces, we define a rule interface that requires two mandatory methods, `match()` and `transform()`, and both of them should return an object implementing the pattern interface. In this interface, there is an optional `constraint()` method that can be used to specify the additional constraints on this rule, which returns an object implementing the expression interface, and by default, it returns the `TRUE` literal. We also provide a default method `explain()` that returns a pretty string for the rule semantics.

We will illustrate how the pattern interface works with two examples below.

4.1.1 Calculate. In Apache Calcite, a Calculate query plan combines the functionality of Filter and Project [14]. Given a source query plan, a Calculate query plan produces projections from input tuples that satisfy the predicate. It should be semantically equivalent to a Project query plan above a Filter query plan for the given input query plan, predicate, and projection. For example, the query plans below should produce the same output in all possible scenarios.

$$\mathsf{Calculate}(x+1, x>0, \mathsf{Table}(x:\mathsf{Integer}))$$
$$\mathsf{Project}(x+1, \mathsf{Filter}(x>0, \mathsf{Table}(x:\mathsf{Integer})))$$

Although we hope that RuleScript can directly support a pattern for Calculate query plans, the underlying solver (i.e. QED) does not have built-in support for them. Thus we need to explain its semantics to the solver when it shows up. With our reference implementation, we can create a new `Calculate` class that implements the pattern interface. The constructor for this class admits any object implementing the pattern interface as its input, along with a predicate and a projection implementing the expression interface, and stores them in the fields of this class. Then we implement `semantics()` by constructing a `Project` above a `Filter` with the class fields and returning it when `semantics()` is invoked. This concludes our implementation of the pattern interface. Finally, this new class can fit nicely into the rest of RuleScript, and when we need to verify a rewrite rule involving `Calculate`, our pipeline automatically invokes the `semantics()` function to obtain its equivalent semantics that is understood by the solver.

4.1.2 Index Join. An Index Join is a special kind of Inner Join that is backed by certain efficient algorithms at runtime. Semantically, it requires that the Join predicate is a conjunction of equality checks between a pair of fields from both input query plans, where the fields from one side are primary keys. For example, the query plans below should produce the same output in all possible scenarios, where all fields in the tables are primary keys.

$$\mathsf{Index\text{-}Join}([(a,x),(b,y)], \mathsf{Table}(a,b:\mathsf{String}), \mathsf{Table}(x,y:\mathsf{String}))$$
$$\mathsf{Join}(\mathsf{Inner}, a=x \wedge b=y, \mathsf{Table}(a,b:\mathsf{String}), \mathsf{Table}(x,y:\mathsf{String}))$$

In addition to creating a new `IndexJoin` class implementing the pattern interface, we can create a new `PairwiseEqual` class implementing the expression interface, whose instances will be used as the predicates for the `IndexJoin` objects. A `PairwiseEqual` object can be instantiated by a list of pairs of field references, and we implement its `semantics()` by constructing the conjunction of pairwise equality assertions for these fields. Then for the `IndexJoin` class, its constructor requires a `PairwiseEqual` object along with two input objects implementing the pattern interface, and we implement its `semantics()` by using the semantics of the corresponding `Join`.

Here we can ignore the requirement for primary keys for Index Join because the rewrite from a Index Join to a Inner Join always holds no matter whether the fields can be treated as primary keys or not, and it is a runtime requirement for the implemented algorithm that the fields from one side should be primary keys.

4.2 Meta Variables and Rule Family

So far we have addressed how to extend RuleScript with custom symbols, but it is tedious to use if we have to manually describe a family of rewrite rules that look similar to each other. Thus we introduce the notion of meta-variables and rule families in RuleScript for this need.

In short, meta-variables are placeholders that can appear in the match pattern and the transform pattern whose `semantics()` are temporarily unknown. We refer to the RuleScript query plans involving meta-variables as templates. We can assign actual RuleScript query plans to these placeholders in templates, and we can obtain a rewrite rule after all meta-variables are assigned. A rule family is a set of rewrite rules created by a set of assignments for one template.

In our reference implementation, a rule family should implement an interface with a `family()` method that returns a list of rewrite rules upon invocation. Meta-variables are `<Alias>` which implement the pattern interface or the expression interface, but they will lead to panics if we invoke the corresponding `semantics()` method. Thus we also need to specify the options for each of the meta-variables. Sometimes, we may have multiple meta-variables in the rewrite rule, and only certain combinations of their assignment may lead to correct rewrite rules. Thus we should specify a set of such valid assignments, with which we can replace the meta-variables in the template and generate the corresponding rewrite rules.

We provide a `RuleGenerator` class in our library as a baseline implementation for this concept that streamlines the process. The constructor for this class accepts a rule involving meta-variables and the set of valid assignments for them, and we implement the rule family interface for `RuleGenerator` by mapping each valid assignment to an actual rewrite rule with variable replacement.

Aside from facilitating the process of constructing similar rewrite rules, the meta-variables also allow us to experiment with potential rewrite rules easily via enumeration. We illustrate this by reusing the motivating example in Sec. 2, where we demand the four Join types are all Outer, and it turns out that this variant of Join Associate rule is provably correct, which we will discuss in Sec. 5. We would like to know if any other combinations of Join types could result in valid variants of the Join Associate rule, except for the cases where all Join types are Inner which is already known.

```
// Q0, Q1, Q2, F0, F1 are unchanged
options =   Assignment((t0, t1, t2, t3), [
                (Inner, Inner, Inner, Inner),
                (Inner, Inner, Inner, Left),
                ...,
                (Outer, Outer, Outer, Outer)])
before  =   Join(t1, F1, Join(t0, F0, Q0, Q1), Q2)
after   =   Join(t3, F0, Q0, Join(t2, F1, Q1, Q2))
rewrite =   Rule(before, after)
family  =   RuleGenerator(rewrite, options)
```

Listing 6: The Join Associate rule with meta-variables

We modify the pseudo-code in Listing 3 to achieve this, as is shown in Listing 6. `Q0, Q1, Q2, F0, F1` remains unchanged, while the Join types are meta-variables named `t0, t1, t2, t3`. We list all assignments of the meta-variables we would like to enumerate in `options`, and in this case, we hope each join type to independently vary across `Inner, Left, Right, Outer`. Thus there are $4^4 = 256$ assignments in `options`, and we put `options` together with the rule in a `RuleGenerator`, which can automatically generate all the 256 rules. It is hard for us to manually check every single one of them, so we verify these rules automatically with QED, which will be discussed in Sec. 5.

Table 1: Valid assignments of `t0, t1, t2, t3` in Listing 6

t0	t1	t2	t3
Inner	Inner	Inner	Inner
Inner	Inner	Inner	Right
Inner	Left	Inner	Inner
Inner	Left	Inner	Right
Inner	Right	Right	Inner
Inner	Outer	Right	Inner
Left	Inner	Inner	Left
Left	Inner	Inner	Outer
Left	Left	Left	Left
Left	Left	Left	Outer
Left	Right	Right	Left
Left	Outer	Outer	Left
Right	Right	Right	Right
Right	Outer	Right	Right
Outer	Right	Right	Outer
Outer	Outer	Outer	Outer

16 of the enumerated assignments can lead to provably correct variants of the Join Associate rule, which are listed in Table 1. Except for the first case and the last case, which we already know, the rest of the assignments could significantly expand the domain of query plans where the Join Associate could apply. In Sec. 6 we will attempt to generate the implementation of all these rules so that developers do not need to manually encode the correct cases.

5 VERIFICATION

To verify the rewrite rules we constructed from the RuleScript symbols in Sec. 3.1, as well as those involving custom query plans in Sec. 4, we need to translate them into encode them into input formats of the QED solver. QED currently accepts any query plan involving uninterpreted types and functions, whose syntax is listed in Sec. 5.1, and we will also briefly discuss how QED verifies these query plans in Sec. 5.2.

5.1 Syntax of Query Plans

Here we briefly list the formalism for query plans, which are the inputs to the QED solver. We list the common syntax and the high-level semantics for them in Table 2.

The output of a query plan is a collection of tuples. We define the type of a tuple to be the product type of its element types. For example, the type of $(0, ”s”)$ is $\mathsf{Integer} \times \mathsf{String}$. Following this definition, the type for the product of two tuples equals the product of their types.

Since QED is the only automated query equivalence solver for now that has the capabilities to reason about uninterpreted symbols

Table 2: Syntax for QED Query Plans

Notation	High-Level Semantics
$Q, Q_0, Q_1, \ldots$	Query plans, which output tuples
$v, v_0, v_1, \ldots$	Individual tuples
$P, P_0, P_1, \ldots$	Predicates, which map a tuple to a Bool
$f, f_0, f_1, \ldots$	Functions, which map a tuple to a tuple
$\alpha, \alpha_0, \alpha_1, \ldots$	Aggregations, which map a collection of tuples to a tuple
τ	Inner | Left | Right | Outer | Semi | Anti
$\mathsf{Table}(R : S)$	A table named R with schema S
$\mathsf{Value}(v_0, \ldots, v_n)$	A table containing values $v_0, \ldots, v_n$
$\mathsf{Filter}(P, Q)$	Filter Q with P
$\mathsf{Project}(f, Q)$	Map Q with f
$\mathsf{Join}(\tau, P, Q_0, Q_1)$	τ-Join Q_0 and Q_1 on P
$\mathsf{Union}(Q_0, Q_1)$	Union Q_0 and Q_1
$\mathsf{Aggregate}(\alpha, f, Q)$	Aggregate Q with α grouped by f
$\mathsf{Distinct}(Q)$	Deduplicate Q

and types, as well as sufficient coverage of SQL features, it is our only option for the underlying solver to verify RuleScript rewrite rules. Consequently, the exact semantics for the query plans we are discussing here is bag semantics, which is the semantic chosen by the QED solver. In bag semantics, query plans output a multi-set of tuples. This means that we care about the multiplicity of the same tuple in the output, and we do not care about the ordering among the output tuples.

5.2 Semi-Ring Semantics for Query Plans

The QED solver verifies that the pair of RuleScript query plans in rewrite rules are equivalent for all uninterpreted types and function operators that appear in them. Within the QED solver, the query plans are recursively translated into semi-ring expressions. For example, a $\mathsf{Table}(R : S)$ is translated into a finitely supported function whose input is any tuple that has the same type as the table schema S and it outputs the multiplicity of the provided tuple in the table, while a $\mathsf{Filter}(P, Q)$ is the product of an indicator function of the predicate and the semi-ring expression for the input query plan.

After the QED solver translates the query plan into semi-ring expressions, it normalizes it with a restricted set of rules into a finite summation of terms, which are unbounded summations of products of indicator functions and finitely supported functions. Given a pair of query plans, QED compares their normalized forms term by term, and if it tries to find equivalent pairs of terms using the help of SMT solvers. QED does not reason about uninterpreted symbols itself, and the support for uninterpreted symbols comes directly from the underlying SMT solvers. A pair of terms are provably equal if the underlying SMT solver cannot find a counter-example that distinguishes them. If QED manages to find matching between the terms from the pair of query plans, then the two query plans are provably equal.

6 CODE GENERATION

In Sec. 3 we discuss how to use RuleScript to express and verify rewrite rules, and in Sec. 4 we propose some additional features for RuleScript to make it more usable in reality. The only remaining problem is how to correctly implement the rule described in RuleScript.

The baseline solution is to ask the developers to implement them manually from scratch, which completely relies on the developers' carefulness and interpretation of the rewrite rule. Moreover, this leads to redundant work since the procedure to match and transform each type of query plan does not vary much, and certain rewrite rules are common to most query optimizer implementations. Given that the efforts have already been made to encode and verify the rewrite rule in RuleScript, we hope to exploit our language to generate functioning implementation based on our representations provided minimal knowledge about the target query optimizer.

From the interpretation of RuleScript in Sec. 3.2, we know that the match pattern in a rewrite rule describes the procedure to verify if the provided query plan is a valid interpretation, while the transform pattern instantiates a new query plan using the information obtained during the verification. This suggests a two-step approach for the implementation. First, we should generate the codes that perform verification based on the match pattern and optionally the constraints and the code should also produce a context that captures relevant information upon successful verification. Second, we should generate the codes that instantiate a new query plan given the context.

Moreover, to maximally reuse code from developers, we should explore the similarity among different rewrite rules. We observe that both the verification procedure and the instantiation procedure for a rewrite rule can proceed recursively, just like the construction of the match pattern and the transform pattern in RuleScript. Thus, for each target query engine, we require users to provide a template implementation for verification and instantiation on all RuleScript symbols, including the custom ones defined by them, and we seek to automatically derive the implementation of all the rules constructed from these patterns for the target engine. Such template implementation is dependent on the implementation of the target query engine, as a result of which it has to be provided by the users.

We refer to such template implementation as a RuleScript adapter. With a RuleScript adapter, RuleScript can recursively compose the implementation for the rewrite rules based on the structure of the match pattern and the transform pattern in a rewrite rule. Such design restricts the danger zone where developers can make mistakes in the implementation of the adapter, which helps to deduplicate the efforts to maintain similar implementations across rewrite rules that come from the code routine to match and transform the same kinds of components in query plans.

To implement an adapter correctly, the rule of thumb is that the generated codes should not apply the rewrite rule on query plans that are not valid interpretations of the match pattern, and the relevant information should be properly maintained in the context and correctly used on the transform pattern. This requires considerable consideration of the design for the context, which we will illustrate with examples later in this section. However, we allow the generated code to be stricter than the corresponding RuleScript rewrite rule, which means that it is fine to not apply the rewrite rule for some interpretations of the match pattern. This could happen a lot due to the limitations of the tools we can use to perform verification or the underlying algorithm, but the correctness of the generated implementation is not violated. For example, in Sec. 4 we define an `IndexJoin` without the use of primary keys, but when we perform generation we can restrict the rule to the cases where primary keys are present and used in pairwise equality checking for the Join predicate.

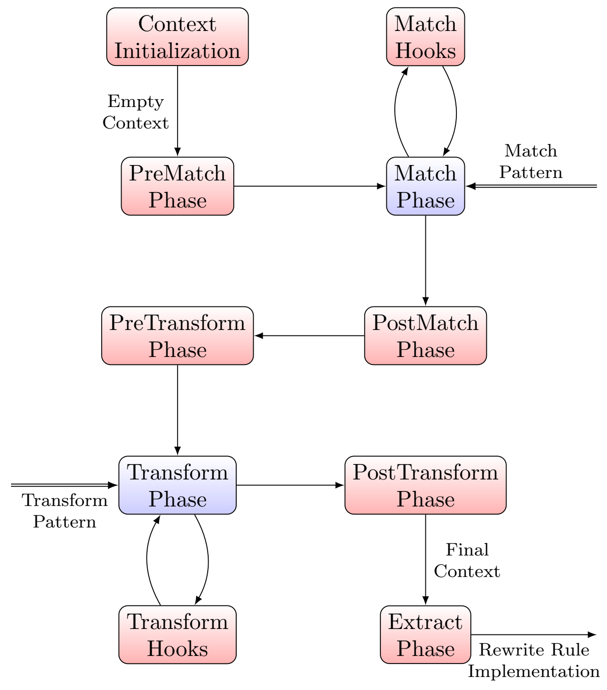

Figure 2: The code generation pipeline within a RuleScript adapter. Red stages require user implementations, while blue stages do not. Arrows without labels represent contexts passed through different stages.

A RuleScript adapter breaks down the code generation routine into two main phases and five auxiliary phases pipelined one after another, as is illustrated in Figure 2. The information that passes through the entire pipeline is the context, and in the two main phases, we additionally take in the match pattern and the transform pattern respectively. The pipeline starts with context initialization, which creates an empty context that propagates through the pipeline. The pipeline ends with the extract phase, which compiles the information in the final context into an implementation of the rewrite rule. The final context is populated with information gathered during the recursive traversal of the match pattern and the transform pattern in the two main phases.

The two main phases are the match phase and the transform phase. During these two phases, RuleScript recursively traverses the pattern using hooks, which are user-provided procedures that specialize in handling certain kinds of RuleScript patterns. For example, when a `Filter` is passed to the match phase, the match phase will look for the match hook defined by users that can handle `Filter`, and forward the match pattern to this hook. When users define such a hook that can handle `Filter`, they can invoke the match phase recursively on the input of the `Filter`. Thus in Figure 2 we can notice a cyclic relation between the match phase and match hooks, and similarly between the transform phase and transform hooks. In this way, we can reuse user-provided hooks whenever we encounter the same kind of pattern. Moreover, the two main phases do not need to be implemented by the user, since the logic to dispatch patterns to corresponding hooks is independent of the target query engine, and they are handled by RuleScript in our reference implementation. The auxiliary phases can be used to perform additional steps before and after the two main phases. We will showcase how such design can help to compile RuleScript rewrite rules to Apache Calcite implementations with an adapter in Sec. 7.

7 CASE STUDIES

With the introduction of RuleScript in previous sections, we can now present the case studies we perform on RuleScript in this section.

7.1 Apache Calcite

We have performed experiments with the rewrite rules from Apache Calcite. Apache Calcite implements 118 rewrite rules in their query optimization pipeline, and we tested the expressiveness of RuleScript by composing these rules in RuleScript. We present the breakdown of rules in Table 3.

Table 3: Breakdown of rewrite rules in Apache Calcite

Category	Total	Supported
Aggregation	21	17
Calculate	6	4
Filter	17	13
Project	23	16
Join	33	20
Sort	8	0
SetOp	7	3
Other	4	0
Total	119	73

The rules are categorized by the relational operators involved in them. For rules that contain multiple kinds of relational operators, we choose the most relevant kinds as their categories. We manually encode 24 of the supported rewrite rules with RuleScript, which expands to 40 cases to be verified by QED. 38 of these cases can be successfully verified within 5 seconds, while the two remaining ones cannot be verified due to the limitations of QED.

Our prototype adapter for Apache Calcite can generate working implementations for simple rewrite rules such as the Filter Merge rule, which transforms two consecutive Filter into one Filter with the conjunction of their predicates. We will present an overview of how this is achieved with the adapter design in Sec. 6 below.

In Apache Calcite, every rewrite rule extends the `RelRule` abstract class. The `RelRule` abstract class consists of a number of methods that each rule should implement. There are two methods that we must implement. The first one is the `onMatch()` method, which checks if the input query plan matches a specific pattern prescribed by the rule, and if so transforms the input query plan accordingly. The second one is the `getOperand()` method, which returns the skeleton of query plans where the rule could apply. The skeleton of a query plan encodes which variants of query plans are involved in the same structure as the original query plan. For example, the Filter Merge rule should apply to Filter query plans where the input query plans are also Filter query plans, so the skeleton of query plans where this rule could apply is Filter(Filter(Any)). For the `onMatch()` method, we must generate Java codes that implement the rewrite rule, while for the `getOperand()`, Apache Calcite provides utility tools with which we can construct the skeleton using a single Java expression.

With the observations above, we need to keep track of a few things in the context for the Apache Calcite adapter. First, we need to create a new Java class for the rewrite rule specified in RuleScript, which includes the import statements in the beginning and the body of the class definition. Then we need to generate Java code for the `onMatch()` method. The body of the `onMatch()` method should consist of a series of variable assignments and return statements . Meanwhile, the Java expression for the skeleton can be constructed recursively without temporary variables. Thus we need to keep track of the list of statements for the `onMatch()` methods, where each statement can use variables defined in previous statements, and we also need to keep track of the skeleton expression that has been constructed so far. Moreover, we need to keep track of the expression for the current query plan where we are focusing, which we will refer to as the focus expression, as well as a mapping from identifiers to expressions that can keep track of how we should instantiate function operators and `Plan` patterns when we see them in the transform pattern. Since a context object keeps track of so many states during code generation, we design it to be immutable so that we do not need to worry about restoring it after we recursively perform operations on it.

With the context class defined above, we can now use them in different phases of code generation. In the match phase, we aim to maintain the recursion convention that the focus expression of the input context is the query plan corresponding to the current pattern we are examining. In the transform phase, we hope to maintain the recursion convention so that the focus expression of the output context is the new query plan we constructed for the current pattern we are examining.

- In the `preMatch()` phase, we modify the empty context by setting the focus expression to the input argument of the `onMatch()` function.
- In the `match()` phase, we need to set the focus expression before we recursively traverse the inputs to the current pattern, and then update the skeleton in the context based on the current pattern and the skeleton we obtained from recursive traversal. We should add `if` statements in the context to verify if the rule is applicable and if not we should return leave the original query plan unmodified. We can add entries in the identifier mapping that could help us generate the code for the transform mappings.
- In the `postMatch()` phase, we may clean up the context and perform additional checks using the constraints on the rewrite rule.

- In the `preTransform()` phase, we initialize a `RelBuilder` object and assign it to a variable in the context, and we set the focus expression to it. `RelBuilder` is the helper class in Apache Calcite that can construct query plans recursively like a stack machine.
- In the `transform()` phase, we recursively generate the code that constructs the query plans for the inputs to the current pattern, and then generate the code that constructs the query plan corresponding to the current pattern using the `RelBuilder` object, and set it to the focus expression for the output context. We can use the identifier mapping in the context to retrieve anything we need for this process.
- In the `postTransform()` phase, we conclude the body of `onMatch()` by adding the statement to return the query plan using the focus expression of the input context.
- In the `extract()` method, we derive the actual implementation from the final context. We first add the import statements in the beginning, then the class definition with the `onMatch()` method, and lastly the skeleton expression that will be used for the constructor of the class.

In addition, we could define utility functions and classes that can be invoked by the generated code, and we can use them by adding the appropriate import statements at the beginning of the generated code. For example, we may need the helper function that composes two function operators, and it is hard to directly generate such a procedure from scratch. Instead, we can write a helper function to achieve this and invoke it during code generation.

We provide the generated Join Associate implementation for Apache Calcite in Listing 7. To facilitate the code generation, we redefine the Join predicates `F0, F1` as aliases whose semantics are the same as before. In this way, we could customize how code generation proceeds for them. Line 4-7 are generated by the match phase, where `matchF0, matchF1` are custom functions that help to verify the Join predicates. `matchF0, matchF1` return the Join predicates unchanged if these predicates can be decomposed as specified by the semantics of the aliases. Otherwise `matchF0, matchF1` return null value, which will terminate the `onMatch()` function and leave the original query plan unchanged. Line 9 is a Java expression generated during the transform phase, which creates a new query plan according to the transform pattern as the optimized query plan. We can reuse the Join predicates using the variables declared in the match phase. Line 14 is the skeleton expression for the rewrite rule, which means that the rewrite rule may be applied to two consecutive Outer Joins. The rest of the code is generated by the auxiliary phases.

```
01 |public class JoinAssociate extends
02 |  RelRule<JoinAssociate.Config> {
03 |  @Override public void onMatch(RelOptRuleCall call) {
04 |    var var_3 = matchF0(((LogicalJoin) call.rel(1)));
05 |    if (var_3 == null) { return; }
06 |    var var_6 = matchF1(((LogicalJoin) call.rel(0)));
07 |    if (var_6 == null) { return; }
08 |    var var_7 = call.builder();
09 |    call.transformTo(var_7.push(call.rel(2)).push(call.
         rel(3)).push(call.rel(4)).join(JoinRelType.FULL,
        var_6).join(JoinRelType.FULL, var_3).build());
10 |  }
11 |
12 |  public interface Config extends EmptyConfig {
13 |    @Override default RelRule.OperandTransform
         operandSupplier() {
14 |      return s_4 -> s_4.operand(LogicalJoin.class).
         predicate(j -> j.getJoinType().equals(JoinRelType.
         FULL)).inputs(s_2 -> s_2.operand(LogicalJoin.class).
         predicate(j -> j.getJoinType().equals(JoinRelType.
         FULL)).inputs(s_0 -> s_0.operand(RelNode.class).
         anyInputs(), s_1 -> s_1.operand(RelNode.class).
         anyInputs()), s_3 -> s_3.operand(RelNode.class).
         anyInputs());
15 |    }
16 |  }
17 |}
```

Listing 7: Generated Join Associate implementation for Apache Calcite. We hide irrelevant Java codes here for simplicity.

7.2 Syntax Guided Synthesis (SyGuS)

Based on our rule semantics, applying a rewrite rule is equivalent to verifying if the provided query plan is a valid interpretation of the match pattern and then instantiating the transform pattern accordingly. In the actual implementation, this is achieved by recursively verifying the query plan against the variants of the match pattern and instantiating the uninterpreted function operators so that expressions in the match pattern are semantically equivalent to those in the given query plan. The recursive verification routine is already handled by the code generation pipeline, so the remaining task is finding the correct instantiation for the expression patterns.

SyGuS seems to be the right toolkit for such a task from our perspective. SyGuS addresses the problem of synthesizing a program that satisfies the specification given the allowed grammar for the program. For example, if we ask the solver to find an unary integer function $f(x)$ such that $f(x) = f(-x)$ using the default integer arithmetic operators, the solver may return $f(x) = x^2$ as the solution.

Such functionality could be very helpful for our pipeline. Take the Join Predicate Push Down rule as an example. This rule aims to push the predicate in an Inner Join down to both of its inputs as much as possible. We define this rule using RuleScript in Listing 8

```
Rule(
    Join(Inner, P0(x, y) AND P1(x) AND P2(y),
        Plan(Q0, [(x, T0)]), Plan(Q1, [(y, T1)]))
    Join(Inner, P0(x, y),
        Filter(P1(x), Plan(Q0, [(x, T0)])),
        Filter(P2(y), Plan(Q1, [(y, T1)])))
```

Listing 8: The Join Predicate Push Down rule

This rule can apply to any Inner Join query plan since we can always set `P0` to be the predicate in the query plan while letting `P1` and `P2` to always return `TRUE`. Although this instantiation is always valid, it is the least interesting one from the perspective of the query optimizer. The goal of this rule is to reduce the size of inputs to the Join operator, but the instantiation above fails to achieve this goal. In reality, the query optimizer will seek to find nontrivial `P1` and `P2` for this rule using custom code routines, such as analyzing the dependencies of the Join predicate, but this is only a best-effort solution.

We know what could be the optimal solution to this problem. Let $P(x, y)$ be the original predicate, we hope the following conditions to hold.

$$\forall x, y.P(x, y) = P_0(x, y) \wedge P_1(x) \wedge P_2(y)$$
$$\forall x.P_1(x) \implies \exists y.P(x, y)$$
$$\forall y.P_2(y) \implies \exists x.P(x, y)$$

The first statement suggests that the instantiated predicates should be semantically equivalent to the original one, while the remaining two statements suggest that `P1` and `P2` should evaluate to `TRUE` only if they have to. This gives us the best predicates we can use for this rewrite rule because they maximally filter away the inputs to the Join while maintaining the original semantics of the query plan. In such a scenario, we have the specification for the program we would like to synthesize on, so a SyGuS solver is supposed to complete the challenge.

However, this approach turns out to be not very practical. We implement a prototype for this using the CVC5 solver, which is a powerful SMT solver with SyGuS capabilities. We create a Inner Join plan on two tables with single integer columns and set the Join predicate to be `x < y AND x + y = 6`. Although we can manually tell that one optimal instantiation is $P_0(x, y) = x + y = 6$, $P_1(x) = x < 3$ and $P_2(y) = y > 3$, the CVC5 solver will timeout on the specification above using the default configuration. If we only tell the first statement to CVC5, it will synthesize near-optimal solutions to the predicates, but even this takes longer than a few seconds, which will not be feasible if we need to perform such tasks more than once for the same user query.

7.3 Limitation

Our project is limited in a few aspects, which will be discussed in this subsection.

List semantics. Our work is based on QED, which interprets the results of query plans as bags of tuples. This prohibits us from encoding rewrite rules where list semantics are involved, such as those designed for Order By and Limit. This can be addressed with an automated query equivalence solver which supports list semantics, but we cannot find a working solution for this. The components of the projects will stay mostly the same except for a few new operators related to list semantics in our core DSL if we eventually find and choose to use a satisfactory solver to replace QED.

Restricted enumeration domain. Currently, it is hard for us to enumerate for symbols other than Join types with meta-variables in RuleScript. For example, we do not have an automated way to come up with the Join predicates for the Join Associate rule in Listing 3. However, this is an independent field of study, and future works can build on top of RuleScript by finding an efficient way to search within the domain of rewrite rules expressible in RuleScript.

Complexity for code generation. Even if we design the code generation pipeline in a modular fashion, it requires considerable engineering efforts to implement an adapter for the query optimizer, and it also requires extra care to maintain the context object during the process, especially when the generation target is at a lower level such as program code. We cannot address the problem without making further assumptions on how the query optimizer is implemented, but from our observation, this varies a lot case by case. We believe that the solution to the problem is a unified query optimizer framework that abstracts away the details of the query engine, and it could be easily shared in codes across different projects. With such a framework, our project can be designed as one of its plugins that handles the verification and execution of individual rewrite rules, and we hope to see such a framework in the future.

8 RELATED WORK

Extensible query optimizer. The idea of extensible query optimization has come a long way since the last century. The Starburst database is among the first RDBMS that bases their query rewrite facilities on a collection of rewrite rules [9], and such design is generalized and standardized by the Volcano Optimization generator [7] and later the Cascade framework [6]. The extensible query optimizer architecture has been improved since then for better extensibility, and it is widely used in modern query processing pipelines, which is the case for Snowflake [5], Apache Calcite [3], and CockroachDB [11]. But still, the growing complexity of rewriting rules today continues to bring development challenges in terms of correctness and development cost. The goal of our work is to address these challenges systematically.

Query equivalence. The problem of verifying the equivalence of queries has attracted serious research interests. The earliest investigation of the problem focuses on its theoretical aspects, and it is proven that the problem in general is undecidable [16]. Later work restricts the problem domain so that the problem can be decided, such as conjunctive queries over bag semantics [8]. The more recent progress on this problem focuses on extending the decidable domain to cover more SQL features, which is made possible by the advancement in automated theorem solvers like CVC5 [2]. Query equivalence verification is reduced to problems in other domains, such as symbolic reasoning in SPES [17] and algebraic reasoning in QED, and query equivalence is considered resolved assuming the existence of powerful solvers that can handle the reduced problems. QED is currently the state-of-the-art query equivalence solver, which supports the features needed for our work, and it is chosen to be our source of truth for rewrite rule verification.

9 CONCLUSION

In this paper, we present RuleScript, an extensible rule language designed to express rewrite rules in any extensible query optimizer. We describe its core syntax, which is designed to resemble the structure of relational query plans, as well as how to extend the core language to incorporate custom patterns with corresponding semantics and how to use meta-variables to generate rewrite rules in batches. We briefly present how rewrite rules in RuleScript are proven by the QED solver. Based on the syntax and semantics of RuleScript, we illustrate how to generate implementations for the rules described in this language provided we have the adapter for the target query engine. We perform case studies for RuleScript in Apache Calcite to test our design in real-world settings.

REFERENCES

[1] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. 1976. System R: Relational approach to database management. *ACM Transactions on Database Systems (TODS)* 1, 2 (1976), 97–137.

[2] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442.

[3] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*. 221–230.

[4] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: Proving query rewrites with univalent SQL semantics. *ACM SIGPLAN Notices* 52, 6 (2017), 510–524.

[5] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.

[6] Goetz Graefe. 1995. The cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[7] Goetz Graefe and William J McKenna. 1993. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th international conference on data engineering*. IEEE, 209–218.

[8] Yannis E Ioannidis and Raghu Ramakrishnan. 1995. Containment of conjunctive queries: Beyond relations as sets. *ACM Transactions on Database Systems (TODS)* 20, 3 (1995), 288–324.

[9] Hamid Pirahesh, Joseph M Hellerstein, and Waqar Hasan. 1992. Extensible/rule based query rewrite optimization in Starburst. *ACM Sigmod Record* 21, 2 (1992), 39–48.

[10] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. 23–34.

[11] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1493–1509.

[12] The Apache Calcite Team. [n.d.]. Apache Calcite: Comments for the Join Associate Rule. https://github.com/apache/calcite/blob/1566663494207c24318d4c1d2c908df4a15c4aa4/core/src/main/java/org/apache/calcite/rel/rules/JoinAssociateRule.java#L103. [Accessed 24-04-2024].

[13] The Apache Calcite Team. [n.d.]. Apache Calcite: Test Cases for Rewrite Rules. https://github.com/apache/calcite/blob/1566663494207c24318d4c1d2c908df4a15c4aa4/core/src/test/java/org/apache/calcite/test/RelOptRulesTest.java. [Accessed 03-05-2024].

[14] The Apache Calcite Team. [n.d.]. Apache Calcite: The Logical Calculate Query Plan. https://github.com/apache/calcite/blob/1566663494207c24318d4c1d2c908df4a15c4aa4/core/src/main/java/org/apache/calcite/rel/logical/LogicalCalc.java. [Accessed 03-05-2024].

[15] The CockroachDB Team. [n.d.]. CockroachDB: Optgen. https://github.com/cockroachdb/cockroach/tree/e3276dc8c5e041aabc787bba43193e638103f31f/pkg/sql/opt/optgen. [Accessed 09-05-2024].

[16] BA Trahtenbrot. 1963. Impossibility of an algorithm for the decision problem in finite classes. *Nine Papers on Logic and Quantum Electrodynamics* (1963), 1–5.

[17] Qi Zhou, Joy Arulraj, Shamkant B Navathe, William Harris, and Jinpeng Wu. 2022. SPES: A symbolic approach to proving query equivalence under bag semantics. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2735–2748.