Quality

A High Performance Spark DQ Library

Chris Twiner

Table of contents

1. Quality - 0.2.0-preview1.1

verage

Statement	88.87 %	Branch	82.32 %

1.1 Run complex data quality and transformation rules using simple SQL in a batch or streaming Spark application at scale.

Write rules using simple SQL or create re-usable functions via SQL Lambdas.

Your rules are just versioned data, store them wherever convenient, use them by simply defining a column.

- 🆕 Spark 4 Connect Support
- 🆕 Folder can use a DefaultProcessor, both Folder and Engine now use the improved collectRunner result processing logic
- 🆕 RuleSuiteGroups, manage a single group of rules by name and use it to access ruleSuites in nested runners and group the results

Rules are evaluated lazily during Spark actions, such as writing a row, with results saved in a single predictable column.

1.2 Enhanced Spark Functionality

- Lambda Functions - user provided re-usable sql functions over late bound columns
- Map lookup expressions for exact lookups and contains tests, using broadcast variables on Classic and Variables on Connect under the hood they are a great fit for small reference data sets
- View loading - manage the use of session views in your application through configuration and a pluggable DataFrameLoader
- Aggregate functions over Maps expandable with simple SQL Lambdas
- Row ID expressions including guaranteed unique row IDs (based on MAC address guarantees)
- Fast PRNG's exposing RandomSource allowing pluggable and stable generation across the cluster
- Support for massive Bloom Filters while retaining FPP (i.e. several billion items at 0.001 would not fit into a normal 2gb byte array) on Spark Classic

Plus a collection of handy functions to integrate it all.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

2. Getting Started

2.1 Building and Setting Up

2.1.1 Migrating from 0.1.x to 0.2.0

Quality, as of 0.2.0, is delivered via 4 jars:

- quality - effectively the same as Quality pre 0.2.0
- quality_api - the basic interface for quality, included by the normal quality library
- quality_testshade - the test shaded uber package for testing and exploration (this shade both the quality and quality_api jars)
- quality_connect_testshade - the connect test shaded uber package for testing and exploration with Spark 4 / DBR 17.3 server extensions. (this only shades the quality_api jar)

Existing users should continue to depend on the quality jar. Connect users on Spark 4 / DBR 17.3 however can also make remote calls by just depending on quality_api.

The following functional areas are only present in the full quality jar:

- Bloom filters, as are
- Processors,
- documentation functions,
- validation functions
- and a number of classic only utility functions.

In order to use the functions and related data types that are not supported in the connect compatible api use classicFunctions:

```
import com.sparkutils.quality.classicFunctions._
```

The classicFunctions rule, engine, folder and expression runner functions will use connect where required and classic where possible. Functions which are only possible to use with classic are annotated with ClassicOnly.

registerQualityFunctions has no params?

com.sparkutils.quality.registerQualityFunctions no longer takes parameters. On classic, non quality_api, it forwards to the com.sparkutils.quality.classicFunction.registerQualityFunctions's default implementation. When using quality_api via connect it's a no-op, as the functions exist on the server.

Breaking Change Spark 4

The SQL function's map_lookup and map_contains take the additional parameter of the MapLookup Spark SQL Variable name:

```
map_lookup('mapid', expr, mapLookupsVar)
map_contains('mapid', expr, mapLookupsVar)
```

Spark 4, Connect and Remote Calls

The quality_api jars can be used as a remote interface to Quality functionality running as a SparkSessionExtension.

The new quality_api jar provides a very thin and stable interface that simply forwards execution to the SparkSessionExtension on the driver and acts as an example of what other language support should provide.

In this pattern the 'client' application only needs to depend on the quality_api jar, allowing the exact Quality implementation 'server' to be upgraded. Under OSS Spark 4:

- spark.sql.artifact.isolation.enabled and
- spark.sql.artifact.isolation.alwaysApplyClassloader

allow multiple client applications to exist and safely share the server, enabled by default when using the Spark Connect server or "spark.api.mode=connect". Databricks Shared Compute clusters with Lakeguard provide the same isolation.

2.1.2 Building The Library

- fork,
- use the Scala dev environment of your choice,
- or build directly using Maven

Building via commandline

For OSS versions (non Databricks runtime - dbr):

```
mvn --batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true -DskipTests install -P Spark321
```

but dbr versions will not be able to run tests from the command line (typically not an issue in intellij):

```
mvn --batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true -DskipTests clean install -P 10.4.dbr
```

You may also build the shaded uber test jar for easy testing in Spark clusters for each profile:

```
mvn -f testShades/pom.xml --batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true -Dmaven.test.skip=true clean install -P
10.4.dbr
```

The uber test jar artefact starts with 'quality_testshade_' instead of just 'quality_' and is located in the testShades/target/ directory of a given build. This is also true for the artefacts of a runtime build job within a full build gitlab pipeline. All of the required jar's are shaded so you can quickly jump into using Quality in notebooks for example.

2.1.3 Running the tests

In order to run the tests you must follow these instructions to create a fake winutils.

Also ensure only the correct target Maven profile and source directories are enabled in your IDE of choice.

The performance tests are not automated and must be manually run when needed.

When running tests on jdk 17/21 you also need to add the following startup parameters:

```
--add-opens=java.base/java.lang=ALL-UNNAMED
--add-opens=java.base/java.lang.invoke=ALL-UNNAMED
--add-opens=java.base/java.lang.reflect=ALL-UNNAMED
--add-opens=java.base/java.io=ALL-UNNAMED
--add-opens=java.base/java.net=ALL-UNNAMED
--add-opens=java.base/java.nio=ALL-UNNAMED
--add-opens=java.base/java.util=ALL-UNNAMED
--add-opens=java.base/java.util.concurrent=ALL-UNNAMED
--add-opens=java.base/java.util.concurrent.atomic=ALL-UNNAMED
--add-opens=java.base/sun.nio.ch=ALL-UNNAMED
--add-opens=java.base/sun.nio.cs=ALL-UNNAMED
--add-opens=java.base/sun.security.action=ALL-UNNAMED
--add-opens=java.base/sun.util.calendar=ALL-UNNAMED
```

Also for Spark 4 builds requiring 17/21 you must use Scala SDK 2.13.12 or similar which supports higher jdk versions.

2.1.4 Build tool dependencies

Quality is cross compiled for different versions of Spark, Scala *and* runtimes such as Databricks. The format for artifact's is:

```
quality_RUNTIME_SPARKCOMPATVERSION_SCALACOMPATVERSION-VERSION.jar
```

e.g.

```
quality_4.0.0.oss_4.0_2.13-0.1.3.1.jar
```

The build poms generate those variables via maven profiles, but you are advised to use properties to configure e.g. for Maven:

```
<dependency>
    <groupId>com.sparkutils</groupId>
    <artifactId>quality_${qualityRuntime}${sparkShortVersion}_${scalaCompatVersion}</artifactId>
    <version>${qualityVersion}</version>
</dependency>
```

The full list of supported runtimes is below:

Spark Version	sparkShortVersion	qualityRuntime	scalaCompatVersion
3.0.3	3.0		2.12
3.1.3	3.1		2.12
3.2.0	3.2		2.12
3.2.1	3.2	3.2.1.oss_	2.12
3.3.2	3.3	3.3.2.oss_	2.12
3.3.2	3.3	12.2.dbr_	2.12
3.4.1	3.4	3.4.1.oss_	2.12
3.4.1	3.4	13.3.dbr_	2.12
3.5.0	3.5	3.5.0.oss_	2.12
3.5.0	3.5	14.3.dbr_	2.12
3.5.0	3.5	15.4.dbr_	2.12
3.5.0	3.5	16.4.dbr_	2.12
4.0.0	4.0	4.0.0.oss_	2.13
4.0.0	4.0	17.3.dbr_	2.13
4.0.0	4.0	api_4.0.0.oss_	2.13
4.0.0	4.0	api_17.3.dbr_	2.13
4.1.0	4.1	api_4.1.0.oss_	2.13
4.1.0	4.1	4.1.0.oss_	2.13

Fabric 1.3 uses the 3.5.0.oss_ runtime, other Fabric runtimes may run on their equivalent OSS version.

Introduced in 0.2.0 is support for Spark Connect driven development, via the quality_api jar (shown above for 4.0.0), this includes Databricks Shared Compute support but requires Session extensions.

0.1.3 Requires com.sparkutils.frameless for newer releases

Quality 0.1.3 uses com.sparkutils.frameless for the 3.5, 13.3 and 14.x releases together with the shim project, allowing quicker releases of Databricks runtime supports going forward. The two frameless code bases are not binary compatible and will require recompilation. This may revert to org.typelevel.frameless in the future.

2.1.5 Sql functions vs column dsl

Similar to normal Spark functions there Quality's functions have sql variants to use with select / sql or expr() and the dsl variants built around Column.

You can use both the sql and dsl functions often without any other Quality runner usage, including lambdas. To use the dsl functions, import quality.functions._, to use the sql functions you can either use the SparkExtension or the regsterXX functions available from the quality package.

Developing for a Databricks Runtime

As there are many compatibility issues that Quality works around between the various Spark runtimes and their Databricks equivalents you will need to use two different runtimes when you do local testing (and of course you *should* do that):

```
<properties>
    <qualityVersion>0.1.3.1</qualityVersion>
    <qualityTestPrefix>4.0.0.oss_</qualityTestPrefix>
    <qualityDatabricksPrefix>17.3.dbr_</qualityDatabricksPrefix>
    <sparkShortVersion>4.0</sparkShortVersion>
    <scalaCompatVersion>2.13</scalaCompatVersion>
</properties>

<dependencies>
    <dependency>
        <groupId>com.sparkutils.</groupId>
        <artifactId>quality_${qualityTestPrefix}${sparkShortVersion}_${scalaCompatVersion}</artifactId>
        <version>${qualityVersion}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>com.sparkutils</groupId>
        <artifactId>quality_${qualityDatabricksPrefix}${sparkShortVersion}_${scalaCompatVersion}</artifactId>
        <version>${qualityVersion}</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

That horrific looking "." on the test groupId is required to get Maven 3 to use different versions many thanks for finding this Zheng.

It's safe to assume better build tools like gradle / sbt do not need such hackery.

The known combinations requiring this approach is below:

Spark Version	sparkShortVersion	qualityTestPrefix	qualityDatabricksPrefix	scalaCompatVersion
3.3.2	3.3	3.3.2.oss_	12.2.dbr_	2.12
3.4.1	3.4	3.4.1.oss_	13.3.dbr_	2.12
3.5.0	3.5	3.5.0.oss_	14.3.dbr_	2.12
3.5.0	3.5	3.5.0.oss_	15.4.dbr_	2.12
3.5.0	3.5	3.5.0.oss_	16.4.dbr_	2.12
4.0.0	4.0	4.0.0.oss_	17.3.dbr_	2.13

See Connect for quality_api based information (Spark 4 onwards).

2.1.6 Using the SQL functions on Spark Thrift (Hive) servers

Using the configuration option:

```
spark.sql.extensions=com.sparkutils.quality.impl.extension.QualitySparkExtension
```

when starting your cluster, with the appropriate compatible Quality runtime jars - the test Shade jar can also be used -, will automatically register the additional SQL functions from Quality.

Pure SQL only

Lambdas, blooms and map's cannot be constructed via pure sql, so the functionality of these on Thrift/Hive servers is limited.

Query Optimisations

The Quality SparkExtension also provides query plan optimisers that re-write as_uuid and id_base64 usage when compared to strings. This allows BI tools to use the results of view containing as_uuid or id_base64 strings in dashboards. When the BI tool filters or selects on these strings passed down to the **same view**, the string is converted back into its underlying parts. This allows for predicate pushdowns and other optimisations against the underlying parts instead of forcing conversions to string.

These two currently existing optimisations are applied to joins and filters against =, <=>, >, >=, <, <= and "in".

In order to use the query optimisations within normal job / calculator writing you must still register via spark.sql.extensions but you'll also be able to continue using the rest of the Quality functionality.

The extension also enables the FunNRewrite optimisation (as of 0.1.3.1 and Spark 3.2 and higher) which expands user functions allowing sub expression elimination.

Optimisations may be disabled through the quality_disable_optimiser_rules spark configuration with comma separated fully qualified class names.

Configuring on Databricks classic runtimes

In order to register the extensions on Databricks runtimes you need to additionally create a cluster init script much like:

```
#!/bin/bash

cp /dbfs/FileStore/XXXX-quality_testshade_12_2_ver.jar /databricks/jars/quality_testshade_12_2_ver.jar
```

where the first path is your uploaded jar location. You can create this script via a notebook on running cluster in the same workspace with throwaway code much like this:

```
val scriptName = "/dbfs/add_quality_plugin.sh"
val script = s"""
#!/bin/bash

cp /dbfs/FileStore/XXXX-quality_testshade_12_2_ver.jar /databricks/jars/quality_testshade_12_2_ver.jar
"""
import java.io._

new File(scriptName).createNewFile
new PrintWriter(scriptName) {write(script); close}
```

You must still register the Spark config extension attribute, but also make sure the Init script has the same path as the file you created in the above snippet.

Dos2Unix

If you are using Windows as your dev env, you will probably have to ensure your line endings are unix, so using git-portable and dos2unix before uploading your file if your are not generating it.

Configuring on Databricks shared runtimes

Supported from DBR 17.3 and Quality 0.2.0 only you must enable init scripts in the UC metastore for a volume. For example:

```
#!/bin/bash

cp /Volumes/databricks_ws/schema/jars/quality_testshade_17.3.dbr_4.0_2.13-0.2.0.jar /databricks/jars/quality_testshade_17.3-0.2.0.jar
```

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

2.2 Defining & Running your first RuleSuite

```
import com.sparkutils.quality._

// setup all the Quality sql functions
registerQualityFunctions()

// define a rule suite
val rules = RuleSuite(rsId, Seq(
  RuleSet(Id(50, 1), Seq(
    Rule(Id(100, 1), ExpressionRule("a % 2 == 0")),
    Rule(Id(100, 2), ExpressionRule("b + 20 < 10")),
    Rule(Id(100, 3), ExpressionRule("(100 * c) + d < e"))
  )),
  RuleSet(Id(50, 2), Seq(
    Rule(Id(100, 5), ExpressionRule("e > 60 or e < 30"))...
  )),
  RuleSet(Id(50, 3), Seq(
    Rule(Id(100, 9),ExpressionRule("i = 5")),
        ...
    ))
  ), Seq(
    LambdaFunction("isReallyNull", "param -> isNull(param)", Id(200,134)),
    LambdaFunction("isGreaterThan", "(a, b) -> a > b", Id(201,131))
  ))

// add the ruleRunner expression to the DataFrame
val withEvaluatedRulesDF = sparkSession.read.parquet(...).
  withColumn("DataQuality", ruleRunner(rules))

withEvaluatedRulesDF.write. ... // or show, or count, or some other action
```

Your expressions used, in dq/triggers, output expressions (for Rules and Folder) and lambda functions can contain any valid SQL that does not include Nondeterministic functions such as rand(), uuid() or indeed the Quality random and unique_id() functions.

3.4 & Sub queries

Prior to 3.4 exists, in, and scalar subqueries (correlated or not) could not be used in any Quality rule SQL snippets.

3.4 has allowed the use of most sub query patterns, such as checking foreign keys via an exists in a dq rule where the data is too large for maps, or selecting the maximum matching value in an output expression. There are some oddities like you must use an alias on the input dataframe if a correlated subquery also has the same field names, not doing so results in either silent failure or at best an 'Expression "XXX" is not an rvalue' compilation error. The ruleEngineWithStruct transformer will automatically add an alias of 'main' to the input dataframe.

Lambdas however introduce some complications, 3.4 quite reasonably had no intention of supporting the kind of thing Quality is doing, so there is code making it work for the obvious use case of DRY using row attributes.

Spark 4.0 / 14.3 LTS introduces SPARK-47509 which limits support by blocking all possible usages. Quality versions after 0.1.3-RC4 work around this by translating all lambda functions at call site to the direct expression. This change has had the added benefit of allowing more complex re-use patterns but may result in more complex errors or the 47509 error.

Per 47509, Quality enables this behaviour only when spark.sql.analyzer.allowSubqueryExpressionsInLambdasOrHigherOrderFunctions is false (the default for Spark 4) or not defined, otherwise the behaviour allows the usage as a higher order function (e.g. in transform etc.) and acts as prior to 0.1.3-RC4.

```
LambdaFunction("genMax", "ii -> select max(i_s.i) from tableName i_s where i_s.i > ii", Id(2404,1)))
```

Calling with genMax(i) or genMax(i * 1) in an Rule or OutputExpression, where i is an column attribute will work and be translated as a join, per 47509 using it within transform will have correctness issues.

2.2.1 withColumn is BAD - how else can I add columns?

I understand repeatedly calling withColumn/withColumnRenamed can cause performance issues due to excessive projections but how else can I add a RuleSuite in Spark?

```
// read a file and apply the rules storing results in the column DataQuality
sparkSession.read.parquet("theFilePath").
  transform(addDataQualityF(rules, "DataQuality"))
```

```
// read a file and apply the rules storing the overall result and details in the columns overallResult, dataQualityResults
sparkSession.read.parquet("theFilePath").
  transform(addOverallResultsAndDetailsF(rules, "overallResult",
    "dataQualityResults"))
```

The transform functions allow easy chaining of operations on DataFrames. However you can equally use the non "xxxxxF" functions such as addOverallResultsAndDetails with the same names to directly add columns and rule processing.

2.2.2 Filtering the Results

The two most common cases for running DQ rules is to report on and filter out bad rows. Filtering can be implemented for a RuleSuiteResult with:

```
withEvaluatedRulesDF.filter("DataQuality.overallResult = passed()")
```

Getting *all* of the rule results can be implemented with the flattenResults function:

```
val exploded = withEvaluatedRulesDF.select(expr("*"),
  expr("explode(flattenResults(DataQuality))").
    as("struct")).select("*","struct.*")
```

Flatten results unpacks the resulting structure, including unpacking all the Id and Versions Ints combined into the single LongType for storage.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

2.3 Those are some Quality flavours

Quality has four main flavours with sprinklings of other Quality ingredients like the sql function suite.

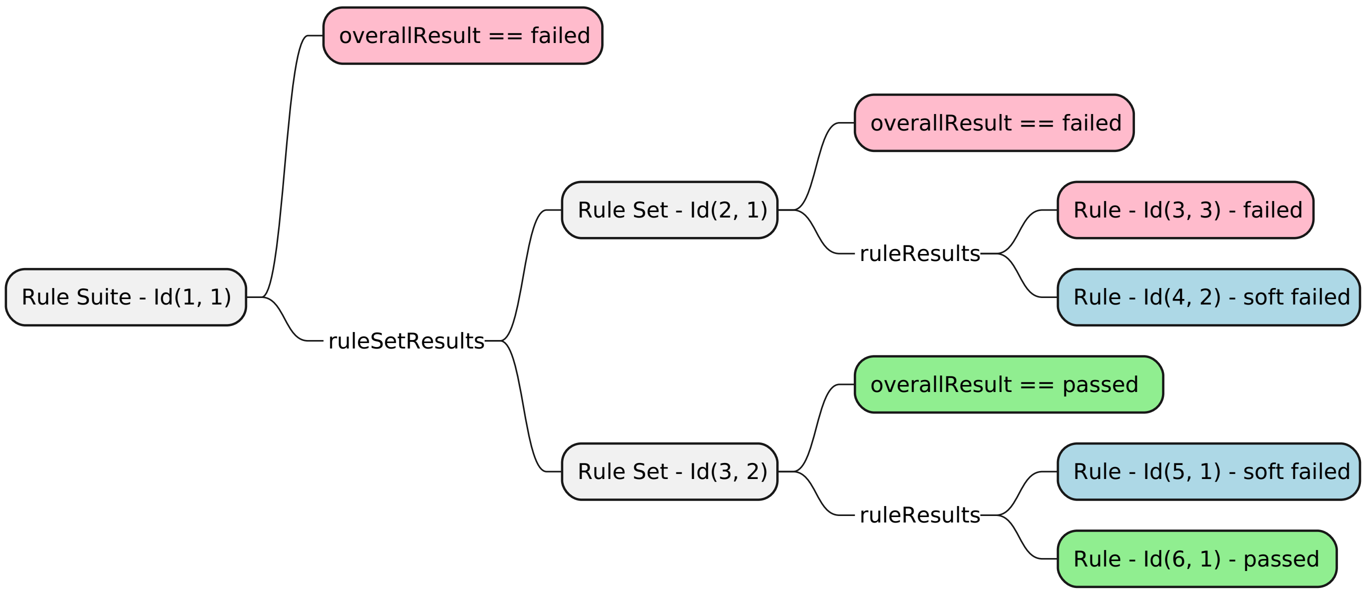
These flavours are provided by four "runners" which add a Column to a Spark Dataset/Dataframe.

2.3.1 Quality / QualityData - ruleRunner

Execute SQL based data validation rules, capture all the results and store them *with* your data for easy and fast access.

Example Usage: Validating in-bound data or the results of a calculation.

What is stored:



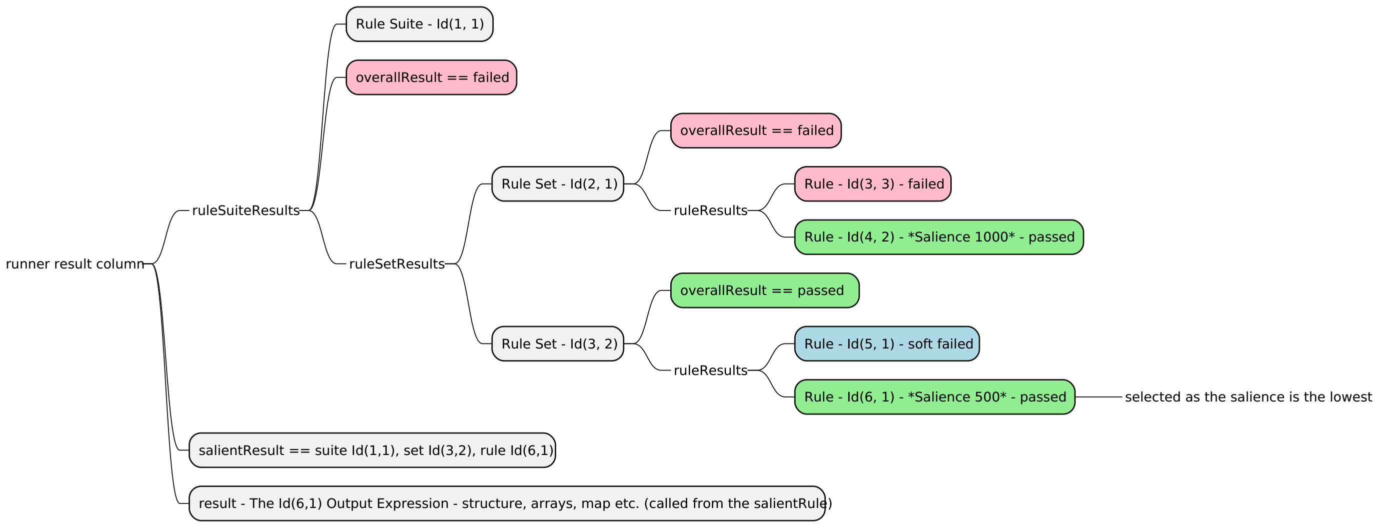
2.3.2 QualityRules - ruleEngineRunner

QualityRules extends the base Quality framework to provide the ability to generate output based on a single SQL rule matching the input data. Effectively an auditable large scale SQL case statement.

Conceptually trigger rules are the *when* and Output rules are the *then* ordered by salience.

Example Usage: Derivation Logic.

What is stored:


runner result column
ruleSuiteResults
Rule Suite - Id(1, 1)
overallResult == failed
ruleSetResults
Rule Set - Id(2, 1)
overallResult == failed
ruleResults
Rule - Id(3, 3) - failed
Rule - Id(4, 2) - *Salience 1000* - passed
Rule Set - Id(3, 2)
overallResult == passed
ruleResults
Rule - Id(5, 1) - soft failed
Rule - Id(6, 1) - *Salience 500* - passed
selected as the salience is the lowest
salientResult == suite Id(1,1), set Id(3,2), rule Id(6,1)
result - The Id(6,1) Output Expression - structure, arrays, map etc. (called from the salientRule)

2.3.3 QualityFolder - ruleFolderRunner

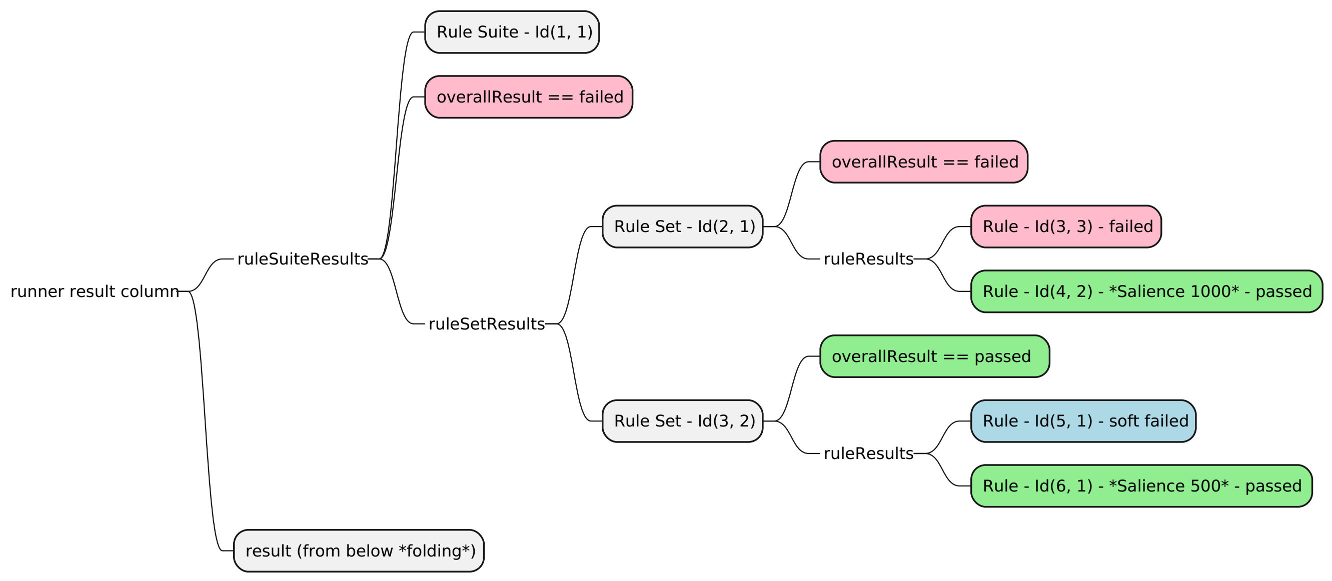
QualityFolder extends QualityRules providing the ability to change values of attributes based on any number of SQL rules matching the input data.

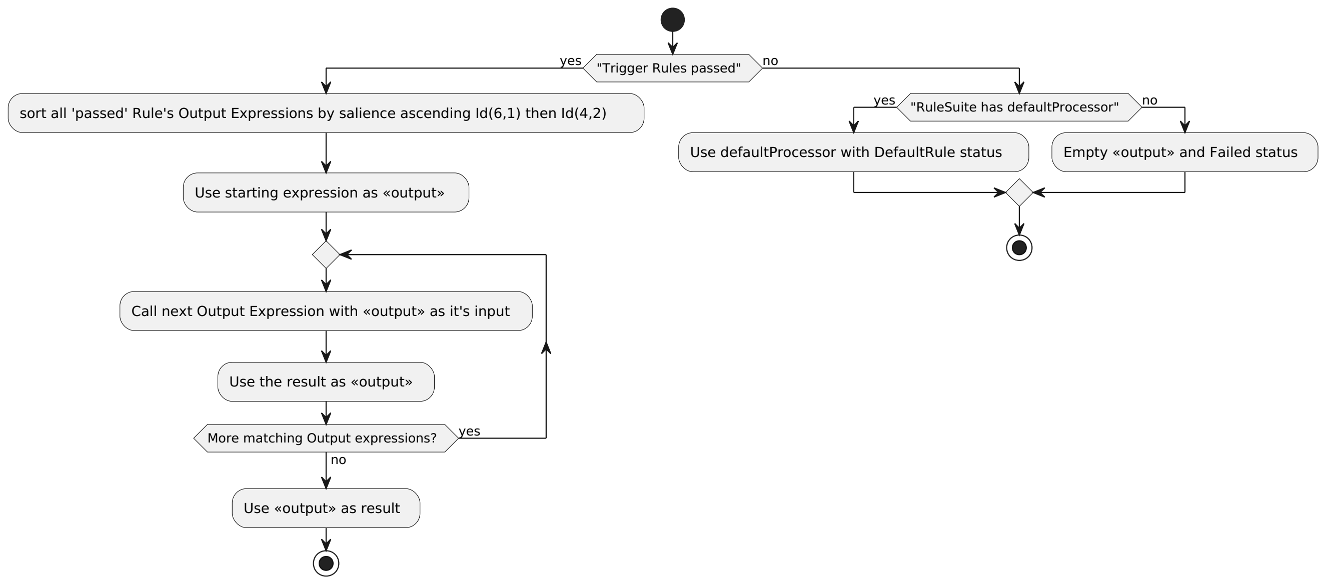
Unlike QualityRules which uses salience to select only one Output expression, Folder uses salience to order the execution of *all* the matching Trigger's paired Output Expressions - folding the results as it goes.

If no triggers match, then an optional defaultProcessor on the RuleSuite can be run.

Example Usage: Correction of in-bound data to enable subsequent calculators to process, defaulting etc.

What is stored:



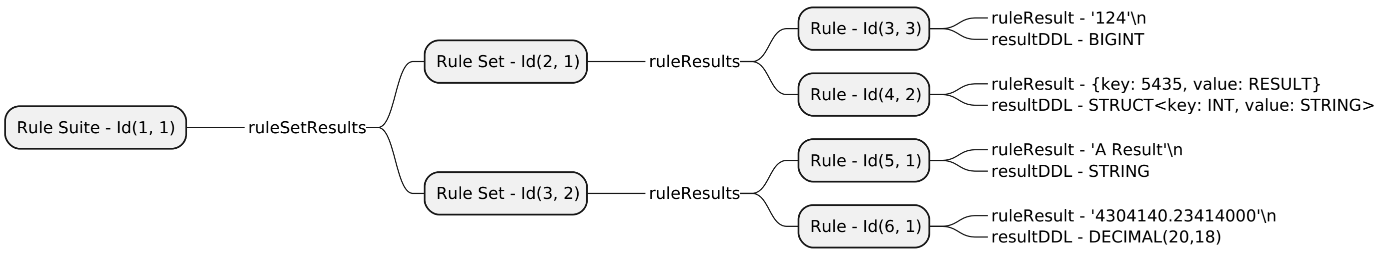

"Trigger Rules passed"
yes
no
sort all 'passed' Rule's Output Expressions by salience ascending Id(6,1) then Id(4,2)
Use starting expression as «output»
Call next Output Expression with «output» as it's input
Use the result as «output»
More matching Output expressions?
yes
no
Use «output» as result
"RuleSuite has defaultProcessor"
yes
no
Use defaultProcessor with DefaultRule status
Empty «output» and Failed status

2.3.4 QualityExpressions - ExpressionRunner

QualityExpressions extends QualityRules providing the raw results as yaml strings (with type) for expressions and allowing aggregate expressions.

Example Usage: Providing totals or other relevant aggregations over datasets or DQ results - e.g. only deem the data load correct when 90% of the rows have good DQ.

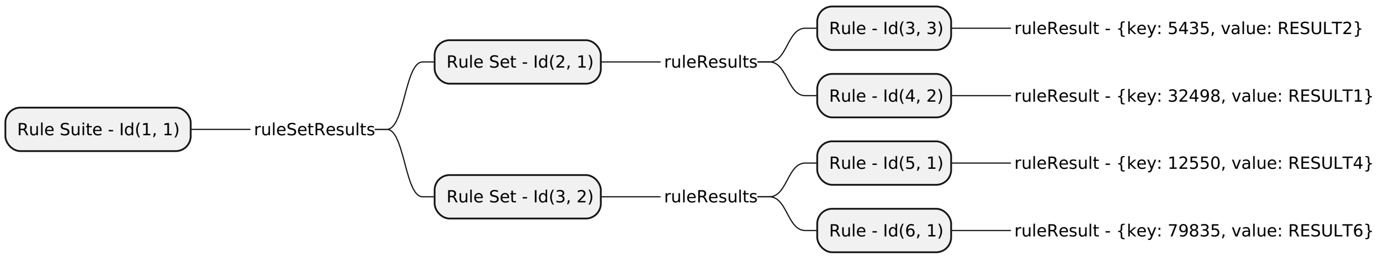
What is stored:



You can also use the typedExpressionRunner, which saves the results of expressions with the same type.

Example Usage: Instead of checking if something exists in a view in a rule, then using the view's value in an Output expression, use typedExpressionRunner to save the lookup value directly. The rule can check if rule_result is null, this can noticeably speed up view heavy queries.

What is stored: For a type of STRUCT


Rule Suite - Id(1, 1)
ruleSetResults
Rule Set - Id(2, 1)
ruleResults
Rule - Id(3, 3)
ruleResult - {key: 5435, value: RESULT2}
Rule - Id(4, 2)
ruleResult - {key: 32498, value: RESULT1}
Rule Set - Id(3, 2)
ruleResults
Rule - Id(5, 1)
ruleResult - {key: 12550, value: RESULT4}
Rule - Id(6, 1)
ruleResult - {key: 79835, value: RESULT6}

2.3.5 QualityCollector - collectRunner

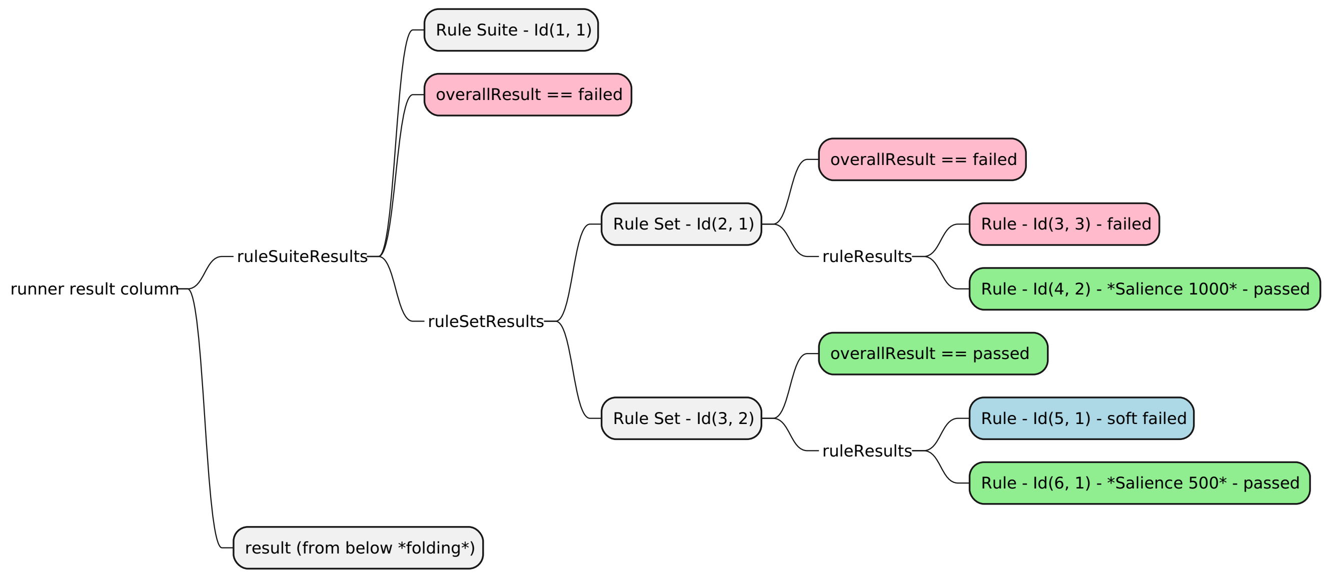
QualityFolder extends QualityRules providing the ability to collect multiple Output Expressions, sorted by salience with a default flatten.

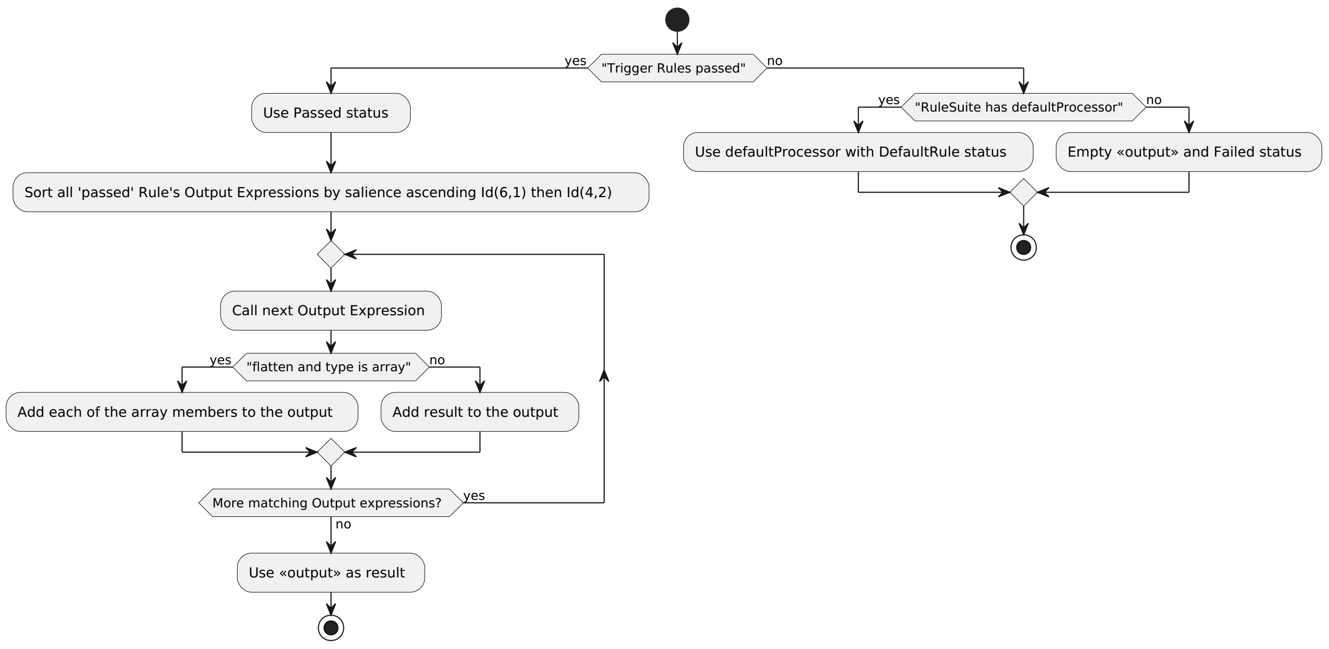
Unlike QualityRules which uses salience to select only one Output expression, Collector uses salience to order the execution of *all* the matching Trigger's paired Output Expressions - collecting the results as it goes.

If no triggers match, then an optional defaultProcessor on the RuleSuite can be run.

Example Usage: Derivation logic

What is stored:




"Trigger Rules passed"
yes
no
Use Passed status
Sort all 'passed' Rule's Output Expressions by salience ascending Id(6,1) then Id(4,2)
Call next Output Expression
"flatten and type is array"
yes
no
Add each of the array members to the output
Add result to the output
More matching Output expressions?
yes
no
Use «output» as result
"RuleSuite has defaultProcessor"
yes
no
Use defaultProcessor with DefaultRule status
Empty «output» and Failed status

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

2.4 Key SQL Functions to use in your Rules

2.4.1 Expressions with constants

- passed() - the value representing a passed rule
- failed() - the value representing a failed rule
- soft_failed() - the value representing a failed rule which doesn't break the bank
- disabled_rule() - the value representing a rule which has been disabled and should be ignored

2.4.2 Expressions which take expression parameters

- probability(x) - returns the probability (between 0.0 for a fail and 1.0 for pass) of a rule result
- pack_ints(lower, higher) - returns a Long with both the lower and higher int's packed in, used for id matching
- soft_fail(x) - if the expression doesn't result in a Passed it returns softFailed() which does not trigger an overall failed() RuleSuite, this is ideal for when you want to flag a rule as passing a test you wish to query on later but do not care if it doesn't pass. It can be treated as a "warn" or passed() expression.
- rule_suite_result_details(ruleSuiteResult) - separates the RuleSuiteResult.overallResult from the rest of the structure should it be needed typically this is done via the addOverallResultsAndDetailsF
- rule_result(ruleSuiteResultColumn, packedRuleSuiteId, packedRuleSetId, packedRuleId) uses the packed long id's to retrieve the integer ruleResult (or ExpressionRunner result) or null if it can't be found.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

2.5 Reading & Writing RuleSuites

2.5.1 Reading & Writing RuleSuites

Typically, you'd save the RuleSuite in configuration tables within a Database or Delta or some other easy to edit store.

Saving:

```
// The lambda functions from the RuleSuite
val lambdaDF = toLambdaDS(rules)
lambdaDF.write .....

// The rest of the rules
val ruleDF = toRuleSuiteDF(rules)
ruleDF.write .....
```

The field names used follow the convention of the default Product Encoder but can be renamed as desired.

Similarly, reading the rules can be as simple as:

```
val rereadWithoutLambdas = readRulesFromDF(ruleDF,
    col("ruleSuiteId"),
    col("ruleSuiteVersion"),
    col("ruleSetId"),
    col("ruleSetVersion"),
    col("ruleId"),
    col("ruleVersion"),
    col("ruleExpr")
  )

val reReadLambdas = readLambdasFromDF(lambdaDF.toDF(),
    col("name"),
    col("ruleExpr"),
    col("functionId"),
    col("functionVersion"),
    col("ruleSuiteId"),
    col("ruleSuiteVersion")
  )

val reReadRuleSuite = integrateLambdas(rereadWithoutLambdas, reReadLambdas)
```

The column names used during reading are not assumed and must be specified.

2.5.2 Versioned rule datasets

The user is completely free to chose their own version management approach, but the design is aimed at immutability and evidencing.

To make things easy a simple scheme with library functions in the simpleVersioning package are provided:

1. Rules can be added to rulesets (or indeed new rulesets) with just a single row within the input DF, this must increase the RuleSet *AND* RuleSuites version:

ruleSuiteId	ruleSuiteVersion	ruleSetId	ruleSetVersion	ruleId	ruleVersion	ruleExpr
1	1	1	1	1	1	`/* existing rule rows */ true()`
1	**2**	1	**2**	**2**	1	`/* new rule */ failed()`

2. Similarly, you can change a rule by adding a new row which increments the Rule Id's, RuleSet *AND* RuleSuites versions:

ruleSuiteId	ruleSuiteVersion	ruleSetId	ruleSetVersion	ruleId	ruleVersion	ruleExpr
1	1	1	1	1	1	`/* existing rule row */ true()`
1	**2**	1	**2**	1	**2**	`/* new version of the above rule */ failed()`

3. To delete a rule you can either use disabled() to flag the rule is inactivated or DELETED to flag the rule to be removed from a RuleSet, as before each version must be incremented:

ruleSuiteId	ruleSuiteVersion	ruleSetId	ruleSetVersion	ruleId	ruleVersion	ruleExpr
1	1	1	1	1	1	`/* existing rule row */ true()`
1	**2**	1	**2**	1	**2**	**DELETED**

4. OutputExpressions may be re-used with different versions (be it for QualityRules or QualityFolder), each rule row that needs to use a later OutputExpression must increment all of it's Id versions. You may are advised to use lambdas to soften the impact:

ruleSuiteId	ruleSuiteVersion	ruleSetId	ruleSetVersion	ruleId	ruleVersion	ruleExpr	ruleE
1	1	1	1	1	1	`true()`	60
1	**2**	1	**2**	1	**2**	`true()`	60

5. Lambda Expressions for a RuleSuite simply take the latest version for a given lambda id. If you want to delete a lambda (for example you have used a name that is now an official Spark sql function) you can add a DELETED row for a given RuleSuite with a higher version.

ruleSuiteId	ruleSuiteVersion	name	functionId	functionVersion	ruleExpr
1	1	aToTrue	1	1	`/** oops */ a -> a`
1	1	always1	2	1	`a -> 1`
1	**2**	aToTrue	1	**2**	`/** corrected */ a -> true()`
1	**2**	always1	2	**2**	**DELETED**

To use these you replace the above with:

```
import com.sparkutils.quality._
import simpleVersioning._

val rereadWithoutLambdas = readVersionedRulesFromDF(ruleDF,
  ...
)

val reReadLambdas = readVersionedLambdasFromDF(lambdaDF.toDF(),
  ...
)

val outputExpressions = readVersionedOutputExpressionsFromDF(outputDF,
  ...
)
val rereadWithLambdas = integrateVersionedLambdas(rereadWithoutLambdas, lambdas)
val (reread, missingOutputExpressions) = integrateVersionedOutputExpressions(rereadWithLambdas, outputExpressions)
```

The "readVersioned" functions modify the dataframe per the above logic to create full sets of ruleSuiteId + ruleSuiteVersion pairs.

The "integrateVersioned" functions will first try the same ruleSuiteId + ruleSuiteVersion pairs and were not present will take the next lowest available version. This runs on the assumption you if didn't need to change any OutputExpressions for a new ruleSuite version why should you need to create fake entries.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

2.6 Running Quality on Databricks

LTS' get explicit support, other interim versions may be supported as needed.

2.6.1 Testing out Quality via Notebooks

You can use the appropriate runtime quality_testshade artefact jar (e.g. DBR 17.3) from maven to upload into your workspace / notebook env (or add via maven). When using Databricks make sure to use the appropriate _Version.dbr builds.

Then using:

```
import com.sparkutils.qualityTests.QualityTestRunner
import com.sparkutils.testing.SparkTestUtils

// uncomment to disable connect test usage on runtimes that support it, like DBR 17.3
// System.setProperty("SPARKUTILS_DISABLE_CONNECT_TESTS","true")

// uncomment to disable classic test usage on runtimes that support connect, DBR 17.3
// a good use case is simulating a UC shared cluster (with init script / spark session extensions enabled)
// when running on a classic cluster setup.
// On an actual UC Shared Compute cluster this is true by default (as the connect.SparkSession is provided)
// System.setProperty("SPARKUTILS_DISABLE_CLASSIC_TESTS","true")

// for running on azure set the configuration for both classic and connect client
val keyMap = Map(s"fs.azure.account.key.${srv_path}${dfs}" -> accountKey)
SparkTestUtils.setRuntimeConnectClientConfig(keyMap)
SparkTestUtils.setRuntimeClassicConfig(keyMap)

val root_path = loc
SparkTestUtils.setPath(root_path+"/qualityTests")
QualityTestRunner.test()
```

in your cell will run through all of the test suite used when building Quality.

Ideally at the end of your runs you'll see - after 10 minutes or so and some stdout - for example a run on DBR 17.3 provides:

```
Quality - starting test batch 0
Run starting. Expected test count is: 183
....
Run completed in 2 minutes, 49 seconds.
Total number of tests run: 183
Suites: completed 10, aborted 0
Tests: succeeded 183, failed 0, canceled 0, ignored 0, pending 0
All tests passed.
projectName - gc'ing after finishing test batch 0
Quality - starting test batch 1
....
Run completed in 1 minute.
Total number of tests run: 158
Suites: completed 10, aborted 0
Tests: succeeded 158, failed 0, canceled 0, ignored 1, pending 0
All tests passed.
....
Run completed in 1 minute, 56 seconds.
Total number of tests run: 106
Suites: completed 10, aborted 0
Tests: succeeded 106, failed 0, canceled 0, ignored 0, pending 0
All tests passed.
Quality - gc'ing after finishing test batch 2
all Quality test batches completed
```

2.6.2 Running on Databricks Runtime 17.3 LTS

Supported as of 0.1.3.1.

17.3, in addition to Spark 4 usage, introduced a binary incompatible change to NamedExpressions not present in the OSS codebase.

The following test combinations are supported as of 0.2.0:

Compute Type	Cluster Library	Extension	Connect Via quality_api	Full Pre 0.2.0 Functionality	QualityTe Test Coun
Non Shared	quality_testshade_17.3			✓	> 500
Non Shared	quality_testshade_17.3	quality_testshade_17.3	✓	✓	> 500 test < 180 test Connect
Shared Compute	quality_testshade_17.3	quality_testshade_17.3	✓		< 180
Shared Compute	quality_connect_testshade_17.3	quality_testshade_17.3	✓		< 180
Shared Compute	quality_connect_testshade_4.0.0.oss	quality_testshade_17.3	✓		< 180
Shared Compute	quality_api_17.3	quality_17.3	✓		⊘
Shared Compute	quality_api_4.0.0.oss	quality_17.3	✓		⊘

Non Shared Connect

Using quality_testshade on both client and extension allows mixing modes. To use connect on the test cases leverage:

```
SparkSession.builder.config("spark.api.mode", "connect").getOrCreate()
```

from the testing 17.3 specific build. For any Fabric (future) / OSS similar approaches using the quality_testshade_4.0.0.oss developers will need to set:

```
System.setProperty("SPARKUTILS_TESTING_USE_LOCAL_CONNECT", "true")
```

before running any tests to stop an attempt to spawned jvm's (which is never needed with Databricks).

Shading/Uber jar

The last two combinations require users to correctly manage the shading and, as such, do not include the test jars or any of the jarjar handling.

See connect for details on how the jars / libraries can be built, correctly shaded and more importantly how the server side can now evolve independently of your client application code.

Using Lakeguard / Shared Compute clusters with 0.2.0

In order to use Shared Compute clusters you must register spark session extensions for the server side quality (to be run on the driver and executor nodes).

As this mode is purely connect, no ClassicOnly functions will be usable.

WHICH SHADE TO USE FOR EXPLORATION OR TESTING?

You can use either the quality_connect_testshade or quality_testshade to test or experiment in workbooks in this setup. The quality_connect_testshade only packages the quality_api so runs fully only via Spark 4 Connect apis, moreover - as it doesn't require server side code you can also use the oss shade.

When using the connect_testshade jar the number of tests run is smaller (only 118 pure connect tests are run) but you should see similar output to:

```
Quality - starting test batch 0
Run starting. Expected test count is: 118
AggregatesTest:
- mapTest (10 seconds, 151 milliseconds)
- multiGroups (6 seconds, 610 milliseconds)
- testFlattenResults (1 second, 809 milliseconds)
- testSalience (1 second, 817 milliseconds)
- testDebug (293 milliseconds)
...
Run completed in 4 minutes, 22 seconds.
Total number of tests run: 118
Suites: completed 10, aborted 0
Tests: succeeded 118, failed 0, canceled 0, ignored 0, pending 0
All tests passed.
Quality - gc'ing after finishing test batch 0
Quality - starting test batch 1
Run starting. Expected test count is: 53
...
Run completed in 1 minute, 28 seconds.
Total number of tests run: 53
Suites: completed 6, aborted 0
Tests: succeeded 53, failed 0, canceled 0, ignored 0, pending 0
All tests passed.
Quality - gc'ing after finishing test batch 1
all Quality test batches completed
```

KNOWN ISSUES

- Logging INFO with map operations, despite these being implemented by Quality, you can ignore these

  INFO Log4jUsageLogger: sparkThrowable=1.0, tags=List(errorClass=UNSUPPORTED_FEATURE.SET_OPERATION_ON_MAP_TYPE), blob=null INFO Log4jUsageLogger: sparkThrowable=1.0, tags=List(errorClass=DATATYPE_MISMATCH.INVALID_ORDERING_TYPE), blob=null
- Any use of Spark Classic / catalyst internals on a shared cluster can trigger very wierd issues such as Verify or ImplementationChanged Errors.
- A number of stacks will seemingly flash in the output window after starting a cell in the notebooks, this seems unrelated to Quality.

2.6.3 Running on Databricks Runtime 12.2 LTS

DBR 12.2 backports at least SPARK-41049 from 3.4 so the base build is closer to 3.4 than the advertised 3.3.2. Building/Testing against 3.3.0 is the preferred approach for maximum compatibility.

2.6.4 Running on Databricks Runtime 13.3 LTS

13.3 backports yet more 3.5 so the 13.3.dbr build must be used.

2.6.5 Running on Databricks Runtime 14.3 LTS

14.3, in addition to the 14.2 StaticInvoke and ResolveReferences changes also implements a new VarianceChecker that requires a new 14.3.dbr runtime.

2.6.6 Running on Databricks Runtime 15.4 LTS

Supported as of 0.1.3.1.

15.4 LTS now requires its own runtime if you are using rng functions as Databricks introduced a breaking change in optimisation of Nondeterministic functions (which relies on a newly introduced Expression.nonVolatile field not present in OSS Spark)

2.6.7 Running on Databricks Runtime 16.4 LTS

Supported as of 0.1.3.1.

16.3 Introduced a number of API changes, Stream is returned in some unexpected forceInterpreted cases, and UnresolvedFunction gets a new param.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

2.7 Running Quality on Fabric

Fabric support has been added since 0.1.3.1 and, at time of the 1.3 runtime, follows the OSS Spark codebase. Other OSS stack to Synapse/Fabric runtimes may similarly "just" work.

2.7.1 Running on Fabric 1.3

Use the OSS 3.5.0 build and testShades.

2.7.2 Testing out Quality via Notebooks

This behaves the same way as per Databricks with one notable exception, System.out is not redirected so you also need:

```scala
// in case it's needed again
val ogSysOut = System.out
System.setOut(Console.out)
```

before running tests to see test progress.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

2.8 Spark Connect Support

Starting with 0.2.0 and Spark 4 Quality leverages the new unified connect and classic APIs to enable connect friendly usage when used with the SparkSessionExtension.

This allows applications to use the quality_api with the stable Spark Connect interface and upgrade their server implementation of Quality without impacting others on the Shared Cluster (other than a restart of course). In order to enable this, from 0.2.0 onwards, Quality's moves to a split jar and implementation model:

- Core Scala types are present in quality_core, each - now source stable for years - are marked with @SerialVersionUID(1L)
- quality_api provides the Connect friendly api, using these core types, with most complex logic taking place on the server
- The quality server jar itself provides the implementation found in the Quality extension and serializes the core jvm types

Breaking binary compatibility of the core types is now identifiable via their @SerialVersionUID(1L) changes. Changes to the core types will include increasing the version id.

Given the stable Connect interface, and a suitable runtime, this *should* allow for client applications to build against versions of Spark Connect for their Scala version only but enjoy running on multiple backends.

Supported combinations:

Spark Runtime	Known Supported Version	Client application Library
Databricks Shared Compute	17.3	quality_api_17.3
Databricks Shared Compute	17.3	quality_api_4.0.0.oss
Databricks Non Shared Compute	17.3	quality_api_17.3
Databricks Non Shared Compute	17.3	quality_api_4.0.0.oss
OSS Spark connect.local	4.0.*	quality_api_17.3

This is also true for remote connect usage, for example running the QualityTestRunner in connect mode from the ide to Databricks with the following environment variables:

```
SPARKUTILS_TEST_OUTPUTDIR abfss://XXX
SPARK_REMOTE sc://adb-XXX.azuredatabricks.net:443/;token=dapiXXX;x-databricks-cluster-id=XXX
SPARKUTILS_DISABLE_CLASSIC_TESTS true
SPARKUTILS_CONNECT_CLIENT.fs.azure.account.key.XXX.dfs.core.windows.net XXXXXXX
```

works without registering jars via Connect client (unless UDF / maps are used) or on the cluster itself (aside from extension). Assuming Databricks 18.x uses 4.1 and the connect code is compatible it *should* be possible to use quality_api_4.0.0.oss against it as well.

This client/server split allows shared servers to upgrade their Quality extension library without forcing each client to upgrade their client version unless the protocol changes (either Spark Connect or quality_api/core).

2.8.1 What is not included in the connect support?

Essentially:

- blooms, these are memory intensive by default but may be targeted for later releases if demand is raised,
- sparkless is distinctly Spark `Classic` in nature
- resolveWith
- validation, documentation
- enableFunRewrites (they are enabled, by default, on the extension side)

Similarly, Databricks serverless is not possible as there is no SparkSessionExtension support for serverless (this is also, of course, true for Classic Quality).

2.8.2 How to build applications against Connect with an Extension?

In order to build your own connect client and server extension jars you should follow:

- the example approach found in the test shade pom to select resources and
- the scripting jarjar approach and shade rules if you wish to re-use the jar for encoding with implicits

The key difference is which jar you build against (the testShades pom illustrates this via the client and server profiles).

- To build your own connect jar, depend on the appropriate quality_api jar only (OSS 4.0.0 and onwards should be sufficient), or
- To build your own server extension jar, depend only on the full quality runtime jar (this will already exclude the api_stub jar)

NB Using the appropriate runtime quality_testshade jar may likely be enough for the server side extension.

Why are there duplicate classes warnings from client shade?

When building a shade you may see "overlapping classes" warnings (this example is for quality_connect_testshade):

```
[WARNING] quality_core_4.0.0.oss_4.0_2.13-0.2.0-SNAPSHOT.jar, quality_api_4.0.0.oss_4.0_2.13-0.2.0-SNAPSHOT.jar, quality_api_stub_4.0.0.oss_4.0_2.13-0.2.0-
SNAPSHOT.jar define 2 overlapping classes:
[WARNING]   - com.sparkutils.shim.EmptyCompilationHelper
[WARNING]   - com.sparkutils.shim.EmptyCompilationHelper$
[WARNING] quality_api_4.0.0.oss_4.0_2.13-0.2.0-SNAPSHOT.jar, quality_api_stub_4.0.0.oss_4.0_2.13-0.2.0-SNAPSHOT.jar define 2 overlapping classes:
[WARNING]   - com.sparkutils.quality.impl.mapLookup.MapLookup$
[WARNING]   - com.sparkutils.quality.impl.mapLookup.MapLookup
[WARNING] maven-shade-plugin has detected that some class files are
[WARNING] present in two or more JARs. When this happens, only one
[WARNING] single version of the class is copied to the uber jar.
[WARNING] Usually this is not harmful and you can skip these warnings,
[WARNING] otherwise try to manually exclude artifacts based on
[WARNING] mvn dependency:tree -Ddetail=true and the above output.
[WARNING] See http://maven.apache.org/plugins/maven-shade-plugin/
```

These can be ignored. The "EmptyCompilationHelper" from shim (and other sparkutils jars) is simply an empty class to force package object scala docs to be built.

The quality MapLookup warnings deserve further explanation, in order to re-use the implementation but provide a consistent interface on both connect and classic quality_api_stub, as the name suggests, provides stub implementations that are then swapped out by quality_api and the classic quality 'server' jar as appropriate.

ep Connect and Server code separated

If your code uses both Classic functions and Connect functions in the same object you may force verify or implementation changed errors as this code is not present in normal Spark connect client runtimes.

2.8.3 Example Java Usage

more to come, including json example

```java
import static com.sparkutils.connect.functions.*;

public void functionUsingSpark(SparkSession session, DataFrame source) {
    DataFrame ruleRows = loadRules(session, columnNames /*...*/);
    DataFrame lambdaFunctionRows = loadLambdaFunctions(session, columnNames /*...*/);
    DataFrame outputExpressionRows = loadOutputExpressions(session, columnNames /*...*/);
    DataFrame combinedRows = combine(ruleRows, lambdaFunctionRows, outputExpressionRows, 0.8d);
    String name = register_rule_suite_variable(combinedRows, VersionedId(1,3), "ruleSuite");
    source.select(col("*"), dq_rule_runner(col(name)));
}
```

2.8.4 Integration Options

Spark Connect, combined with the Quality extension, opens the door for usage from any Spark Connect supporting language as the integration surface is much lower:

```
// versioned reads
def readVersionedRuleRowsFromDF(df: DataFrame, ruleSuiteId: Column,....): DataFrame
def readVersionedLambdaRowsFromDF(lambdaFunctionDF: DataFrame, lambdaFunctionName: Column,....): DataFrame
def readVersionedOutputExpressionRowsFromDF(outputExpressionDF: DataFrame, outputExpression: Column,....): DataFrame
def readVersionedRuleSuitesFromDF(ruleSuitesDF: DataFrame, ruleSuiteId, rulesSuiteVersion,....): DataFrame
// combine and register functions, which can use either simple or versioned reads
def combine(ruleRows: Dataset[RuleRow], lambdaFunctionRows: Dataset[LambdaFunctionRow],
  outputExpressionRows: Dataset[OutputExpressionRow], probablePass: Double,
  globalLambdaSuites: Option[Dataset[Id]] = None, globalOutputExpressionSuites: Option[Dataset[Id]] = None,
  ruleSuites: Option[Dataset[RuleSuiteRow]] = None): Dataset[CombinedRuleSuiteRows]
def register_rule_suite_variable(ds: Dataset[CombinedRuleSuiteRows], id: VersionedId, stableName: String): String
// version specific for lambdas
QualitySparkUtils.registerLambdaFunctions(functions: Seq[LambdaFunction])
```

with each function running on the server and connect using simple commands on local non-global temp views, after any necessary renames etc.:

```
-- versioned reads
QUALITY VERSIONED RULES FROM DF viewName; -- With columns: ruleSuiteId, ruleSuiteVersion, ruleSetId, ruleSetVersion, ruleVersion, ruleExpr,
ruleEngineSalience, ruleEngineId, ruleEngineVersion
QUALITY VERSIONED LAMBDAS FROM DF viewName; -- With columns: name, ruleExpr, functionId, functionVersion, ruleSuiteId, ruleSuiteVersion
QUALITY VERSIONED OUTPUT EXPRESSIONS FROM DF viewName; -- With columns: ruleExpr, functionId, functionVersion, ruleSuiteId, ruleSuiteVersion
QUALITY VERSIONED RULESUITES FROM DF viewName; -- With columns: functionId, functionVersion, probablePass, ruleSuiteId, ruleSuiteVersion
-- combine
QUALITY COMBINE RULESUITES ruleRowsName, lambdaFunctionRowsName | `None`,
  outputExpressionRowsName | `None`, globalLambdaSuitesName | `None`,
  globalOutputExpressionSuitesName | `None`, ruleSuitesName | `None`
QUALITY REGISTER RULE SUITE combinedRowsName, ruleSuiteId Int, ruleSuiteVersion Int, stableName
-- lambdas
CREATE QUALITY FUNCTION simplename _WITH_IMPL_ simpleExpression _END_OF_USER_FUNCTION_
    singleParamName _WITH_IMPL_ p1 -> simpleExpression _END_OF_USER_FUNCTION_
    multiParamsName _WITH_IMPL_ (p1, p2) -> simpleExpression _END_OF_USER_FUNCTION_
```

The loading and serialising functions register ruleSuites as Spark SQL Variables (via DECLARE VARIABLE/SET VARIABLE) with all actual ruleSuite handling taking place on the server.

The other non-loading functionality is represented as Spark Connect compatible sql function calls that require the SparkSessionExtension.

This includes the runners themselves, which also get dsl equivalents:

- dq_rule_runner
- rule_engine_runner
- rule_folder_runner
- typed_expression_runner
- expression_runner

as well as some new utility functions:

- process_if_attribute_missing

All SQL functions are simply forwarders to Spark Connects "call_function" implementation.

What does CombinedRuleSuiteRows look like?

Users load CombinedRuleSuiteRows (or the equivalent DataFrame) representing the following DDL:

```
ruleSuiteId INT NOT NULL,ruleSuiteVersion INT NOT NULL,
 ruleRows ARRAY<
    STRUCT<ruleRow: STRUCT<
            ruleSuiteId: INT NOT NULL, ruleSuiteVersion: INT NOT NULL, ruleSetId: INT NOT NULL,
            ruleSetVersion: INT NOT NULL, ruleId: INT NOT NULL, ruleVersion: INT NOT NULL,
            ruleExpr: STRING, ruleEngineSalience: INT NOT NULL, ruleEngineId: INT NOT NULL,
            ruleEngineVersion: INT NOT NULL
        >,
        outputExpressionRow: STRUCT<
            ruleExpr: STRING, functionId: INT NOT NULL, functionVersion: INT NOT NULL,
            ruleSuiteId: INT NOT NULL, ruleSuiteVersion: INT NOT NULL
        >
    >
>,
lambdaFunctions ARRAY<
    STRUCT<
        name: STRING, ruleExpr: STRING, functionId: INT NOT NULL, functionVersion: INT NOT NULL,
        ruleSuiteId: INT NOT NULL, ruleSuiteVersion: INT NOT NULL
    >
>,
probablePass DOUBLE,
defaultProcessor STRUCT<
  ruleExpr: STRING, functionId: INT NOT NULL, functionVersion: INT NOT NULL,
  ruleSuiteId: INT NOT NULL, ruleSuiteVersion: INT NOT NULL
>
```

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

3. About

3.1 History

3.1.1 Why Quality?

When looking at the Data Quality options for a data mesh standard runtime offering we identified gaps in the available platforms, so we asked:

```
What would our Data Quality library look like?
```

We ended up with a highly peformant and extensible row-level SQL based rule engine with low storage costs and a high degree of optimsation for both Spark and Databricks Runtimes.

3.1.2 Gaps in existing Spark Offerings

Deequ and databricks dq were unsuitable for the meshes requirements, crucially these tools (and others such as OwlDQ) could not run at low cost with tight SLAs, typically requiring processing the data once to get DQ and then once more to save with DQ information or to handle streamed data, not too surprising given their focus on quality across large data sets rather than at a row processing level as a first class citizen. An important use case for DQ rules within this mesh platform is the ability to filter out bad rows but also to allow the consumer of the data to decide what they filter, requiring the producers results to ideally be stored with data rows themselves. Additionally, and perhaps most importantly, they do not support arbitrary user driven rules without recoding.

As such our notional library needs to be:

- fast to integrate into existing Spark action without much overhead
- auditable, it should be clear which rule generated which results
- capable of handling streamed data
- capable of being scripted
- integrate with DataFrames directly, also allowing consumer driven rules in addition to upstream producer DQ
- be able to fit results into a single field (e.g. a map structure of name to results) stored with the row at time of writing the results

3.1.3 Resulting Solution Space

In order to execute efficiently with masses of data the calculation of data quality must scale with Spark, this requires either map functions, UDFs or better still Catalyst Expressions, enabling simple SQL to be used. Storage of results for a row could be json, xml or using nested structures.

The evaluation of these solutions can be found in the next sections.

3.1.4 How did Rules and Folder come about?

Whilst developing a bookkeeping application a need for simple rules that generate an output was raised. The initial approach taken, to effectively generate a case statement, ran into size and scale limitations. The architect of the application asked - can you have an output sql statement for the DQ rules? The result is QualityRules, although it should probably be called QualityCase...

QualityFolder came from a related application which had a need to transform data - providing defaulting in some circumstances - but still had to be auditable and extensible as QualityRules was.

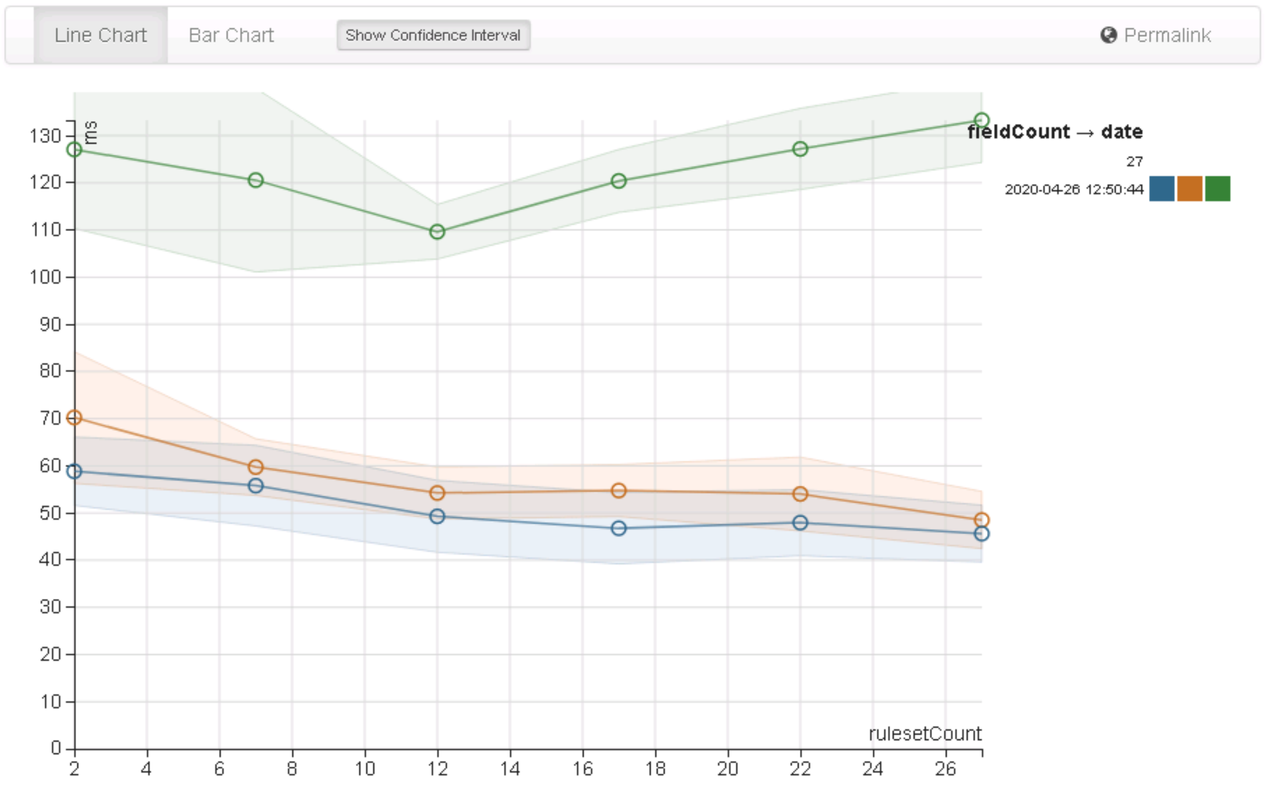
Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

3.2 Performance Choices

3.2.1 How should rules be evaluated?

Performance wise there is a clear winner as to approach for generating results:



The green row is using the map function which is unfortunately the most straightforward to program. The blue is the baseline of processing a row without DQ and the orange is using withColumn.

withColumn can use UDFs or inbuilt Catalyst style functions - the latter giving better performance and ability to more naturally integrate with spark, this review echos the findings and hinting at the effects of catalyst.

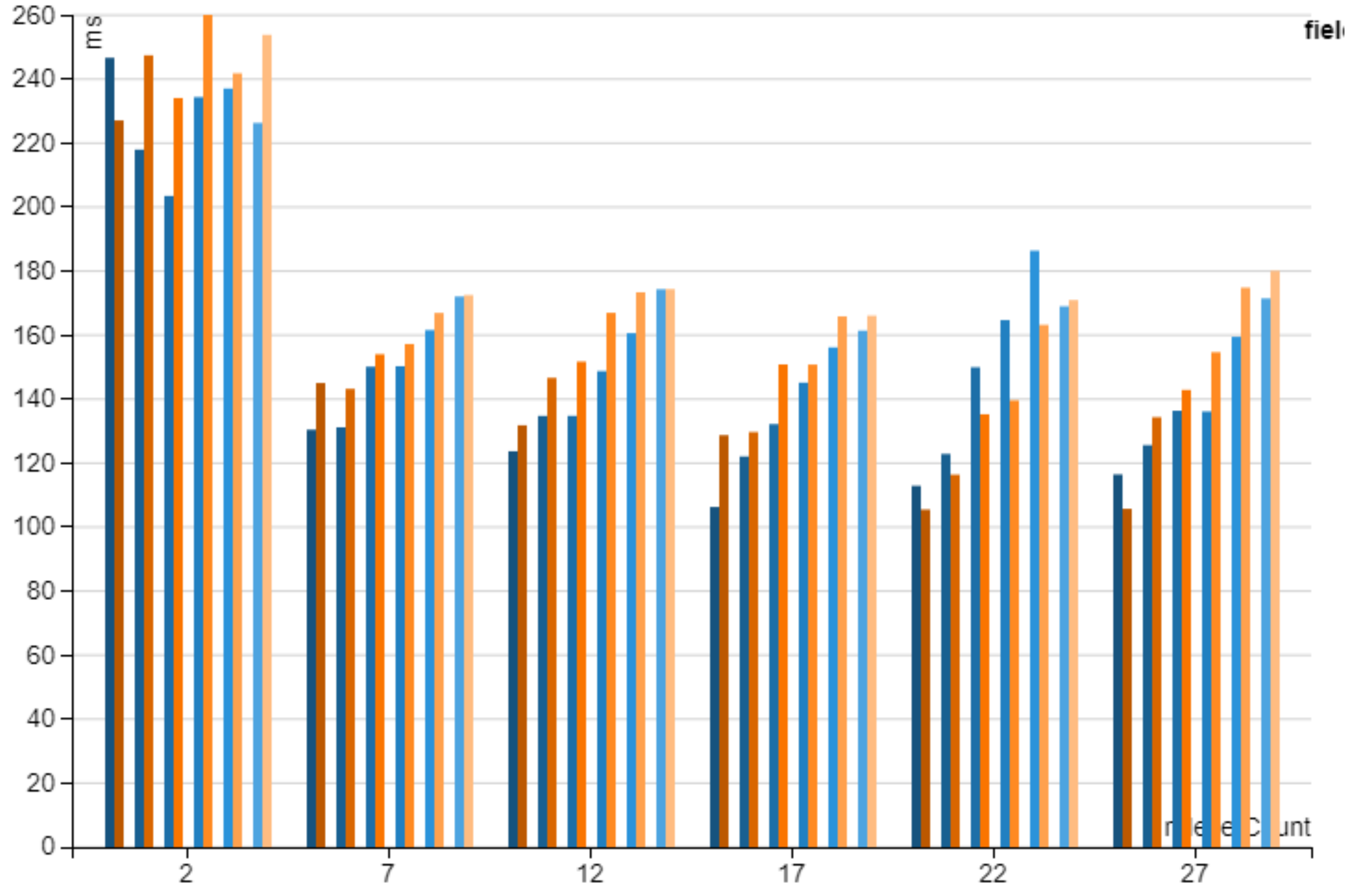
Overall storage winner is nested columns, it has lower storage costs, is as fast as json to serialize (via an Expression) and faster to query with predicate push down support for faster filtering. Details of the analysis are below.

Note

Using withColumn is strongly discouraged, it very quickly introduces performance issues in spark code, prefer to use select and the Quality transform functions. A large part of the performance hit for using UDFs over Expressions is due to the conversion from user types to InternalRow - this cannot be avoided.

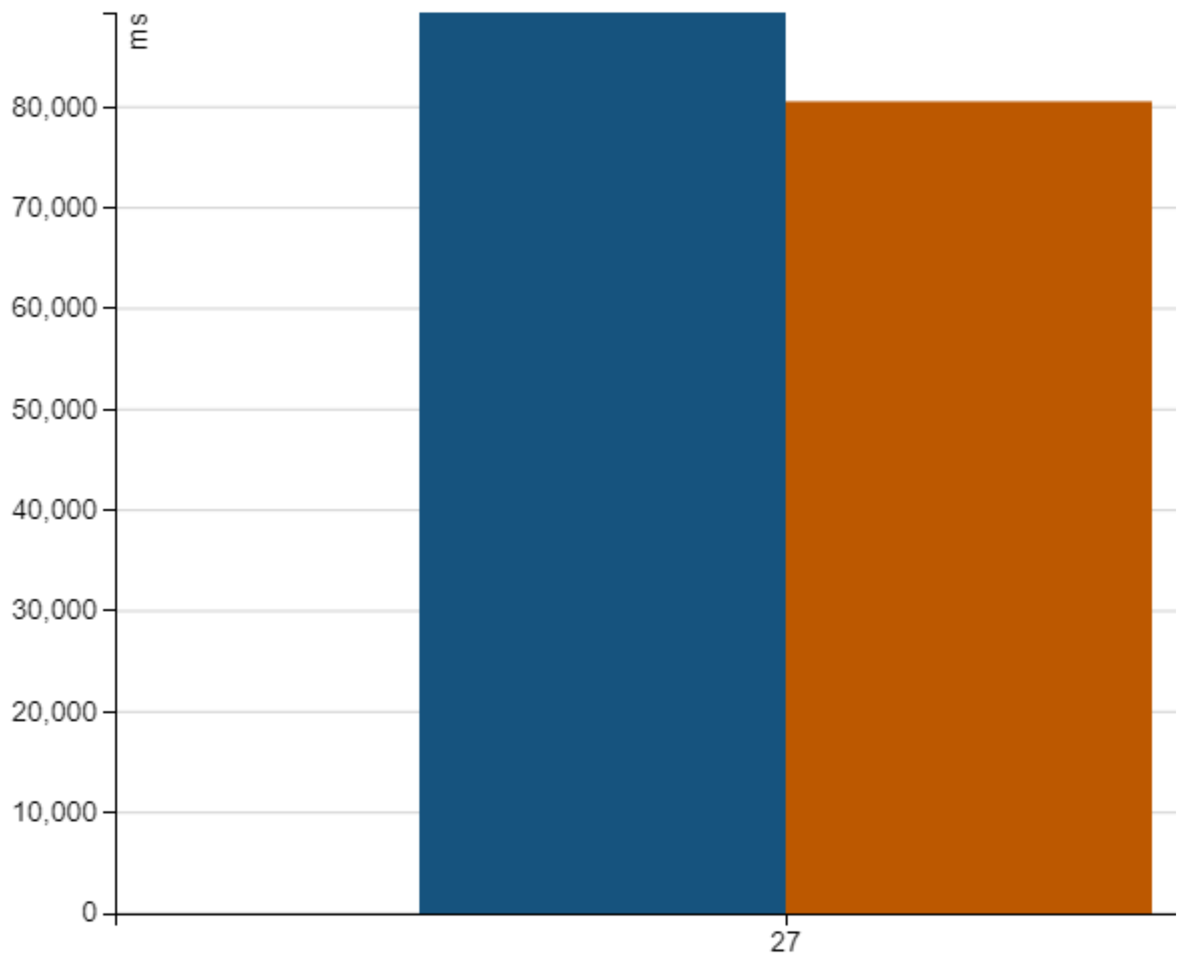
Catalyst Expression Performance

This diagram illustrates the overhead of cost of using Expressions using a simulated complexity of rule suites with increasing number of column checks (c here is the column number, for a simple even check): (`$c` % 2) = 0



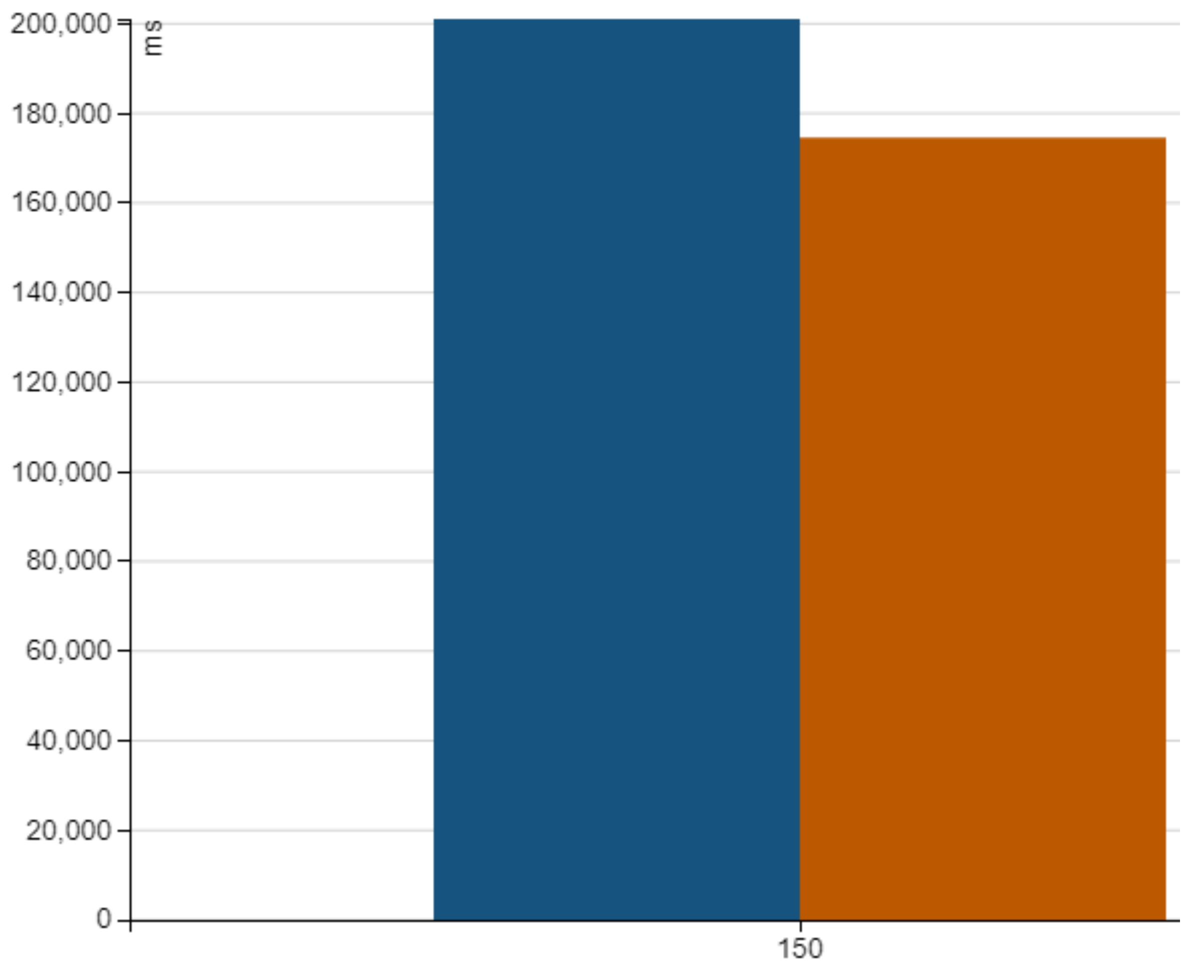
This measurement against 1k rows shows for the last column 230ms for 27 rules each with 27 columns applied, i.e. 0.23 ms per row for 84 rules total (albeit simple rules) on a single 4 core machine (24G heap). Orange representing the default compiled evaluations.

However, this doesn't illustrate very well how things can scale. Running the 27 rules against 1m rows we see:



with a mean time of 80,562ms for 1m rows that's 0.08ms per row for 27 rules, again orange representing the default options for compilation. Conversely, the same test run against 1m rows without rules has a mean of 14,052 - so 66,510ms overhead for processing 27m rules (i.e. 0.0025ms per simple rule).

Stepping the complexity up a bit to 150 columns at 100k (24G ram) with a baseline no rules time of 15,847ms. Running with rules gives:



so for compiled at a mean of 174,583ms we have 15m rules run at 0.011ms per rule. So although increased rule count obviously generates more work the overhead is still low per each rule even with larger counts and the benefit of the default (orange) compilation is visible (see the note at the bottom for when this may not be the case).

When using RuleEngineRunners you should try to re-use output expressions (RunOnPassProcessor) wherever possible to improve performance.

Using custom compilation settings (compileEvals = true, forceTriggerEval = true, forceRunnerEval = true) and chaining runner is not supported

Chaining calls to runners using the result of a runner in another runner can lead to nesting of runners, as if you called runners via sql directly within a trigger or an output expression. This is only supported when using the default compilation options and correctly working as part of wholestage codegen, allowing Spark to "do it's thing" will also allow for higher degrees of sub expression elimination, constant folding etc..

If you must use this, or if compilation - despite Quality's function grouping approach - becomes impossible (due to time or code size) you may use .cache or write out intermediatary results. (see #110)

Sometimes Interpreted Is Better

For very large complex rules (tested sample is 1k rules with over 50k expressions - over 30s compilation for a show and write) compilation can dominate time, as such you can set forceRunnerEval to true on RuleRunner and RuleEngineRunner to skip compilation. While compilation can be slow the execution is heavily optimised with minimal memory allocation, as such you should balance this out when using huge RuleSuites.

Make sure to .cache or write out intermediate results if you are chaining calls and never directly nest calls to runners without the supported defaults. (see #110)

Disabling compilation entirely is not a great idea

Disabled generation, via `ruleRunner(ruleSuite, compileEvals = false, forceRunnerEval = true)`, takes 208,518ms for 150 rules over 100k data - 34s longer than the default, this of course adds up fast over millions of rows.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

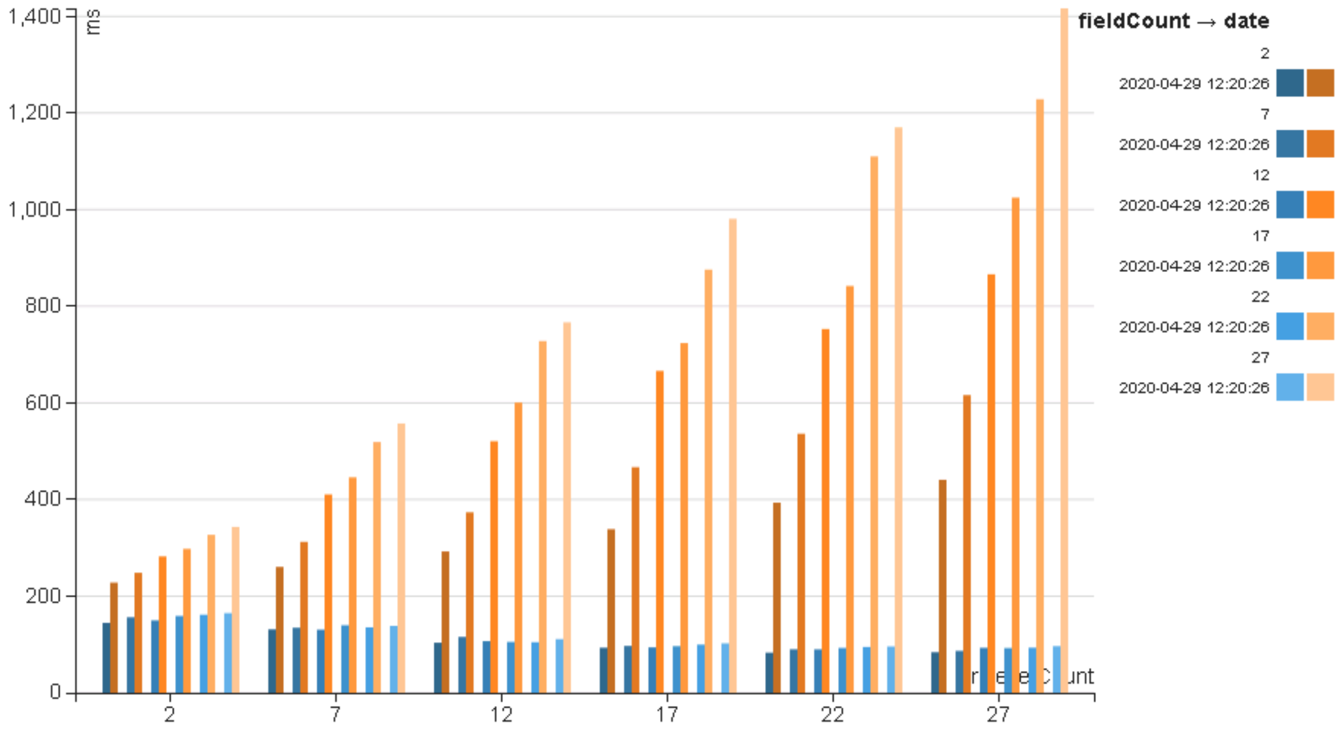
3.2.2 How should rule results be stored? - JSON vs Structures

Note

While Jackson is faster than circe serialization for JSON it doens't serialize easily so only used for comparison as its the fastest possible serialization framework.

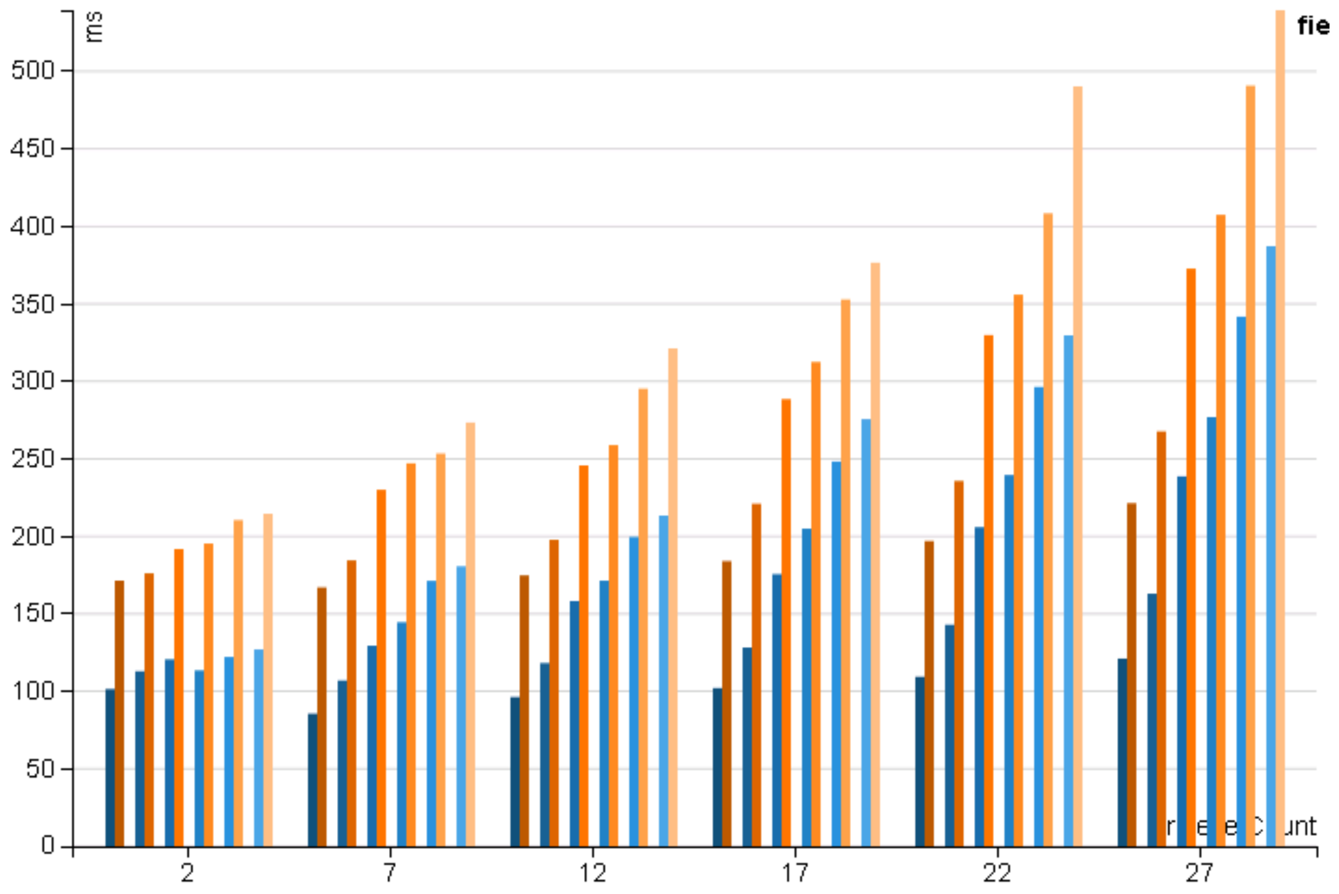
UDF Created Structures

When serializing rule results to Nested Rows via UDF struct creation (shown as Orange) the results are very expensive, the more complex the rule setup the worse the performance. In comparison Jackson (shown as blue) keeps a low cost as it's just a string (the cost instead is in parsing, storage and filtering)



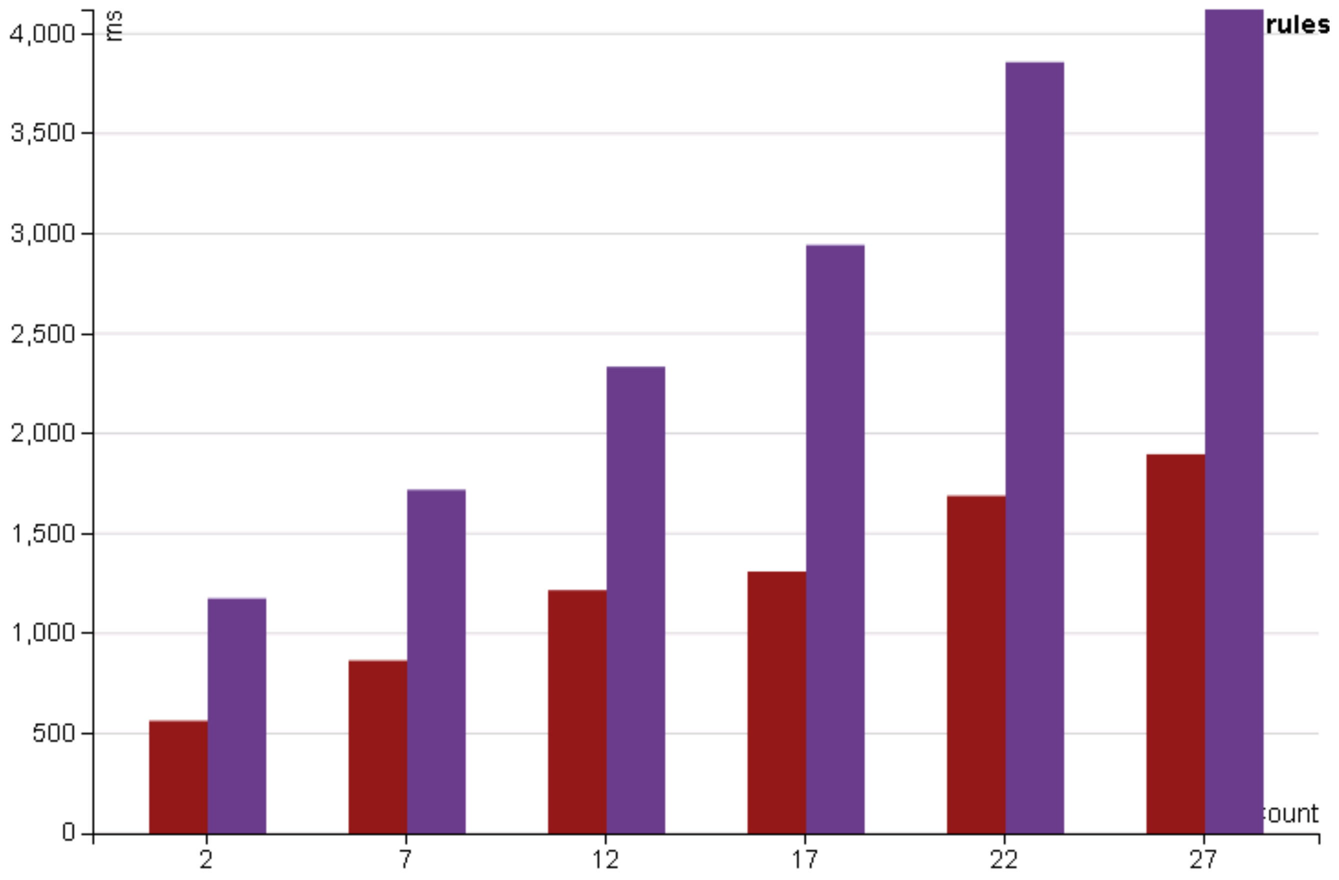
Expression Created Structures

When serializing rule results with a custom Expression (shown as orange, using eval only - without custom compilation), Jackson (shown as blue) based serialisation looses it's clear lead with Expressions closing the gap as complexity increases:



Filtering Costs

Filtering on a nested column with deep queries (shown in red) is as expected faster the same query with a json structure. Nested predicates can be pushed down to the underlying storage for efficient querying.



Note

Depending on the Databricks runtime used the benefit from seperating the overallResult field to a top level field can be 10-20% faster. While each new release of Spark and DBR closes this gap it is recommended to use addOverallResultsAndDetailsF to split the fields.
This not only improves filter speed but also benefits with a simpler filter sql.

Structure Model - storage costs

A naive structure representing RuleSuite, RuleSet and Rule results is actually less efficient than storage of JSON, however the current compressed model used by Quality has low overhead for even complex results.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

3.3 Changelog

0.2.0 24th December, 2025

This release migrates Spark 4 support to use AgnosticEncoders and removes EOL runtimes: 2.4 and DBR's 9.1, 10.4, 11.3, 13.1 and 14.0.
Spark runtimes 3, 3.1.3, 3.2.0, 3.2.1 and 3.3.2 are deprecated as are DBR's 12.2 and 13.3 and will be removed as of Quality version 0.3.0.

#90 - Migrate to Spark 4 sql-api, AgnosticEncoder's and support Connect:

The Sparkutils libraries Shim 0.3.0, Testing 0.1.0 and Frameless 2.0.0 provide support for custom encoding via AgnosticEncoders and a stable API that works with both Spark Classic and Connect.

To use Connect the Quality SparkSessionExtension must be enabled on the "server" Spark Driver side, the connect friendly DSL forwards all the complexity to the Connect Server. The newly released Testing project is used to run the same test cases against both Classic and Connect ensuring the API is stable. This also extends to user functions (LambdaFunctions), on Connect they are sent to the server via a custom command.

The Testless cluster notebook testing experience has been abstracted to the Testing project providing a standard interface Quality Scalatest's on clusters, now shared with all Sparkutils testshade based projects. On Databricks the testing runs will also run in the normal Classic mode and, via the scala.api.mode=connect config parameter, against the provided Spark Connect server.

A number of functions are not possible to run in Connect and are provided with _classic as a suffix, these typically relate to extension points such as monadic add.

Map and Bloom related functions from 0.2.0 Spark 4 onwards allow multiple lookups to be used and leverage a struct Spark Variable. This change swaps the last parameter type of the DSL, and introduces a third breaking change parameter for the SQL interface, to refer to the Spark Variable, with each map being a strongly typed member of the variable available for use with any Spark queries (although probably not all too useful for blooms).

If code was using QualitySparkUtils the import is now ClassicQualitySparkUtils.

Serializing classes are moved to the root package, the impl.util package object provides deprecated forwarders.

#96 - quality_api is introduced, leveraging Spark Connect - allows a client server model and further client language support

#100 - Support for key functions to be run from the SparkSessionExtension when using quality_api, reducing the integration surface area for other client languages and simplifying upgrades

#108 - Support for Spark 4.1 added

#72 - Defaults for sub expression elimination and compilation of triggers are changed to better overall performance with recent Spark versions.

compileEvals and forceTriggerEval now default to false for all runner types and are removed for the connect api, as are resolveWith. This has been found to be the best balance for most rules with large performance gains as of 0.1.3.1 for long-running processes or larger data volumes. These can be set to the previous defaults for the old behaviour if code generation itself dominates your applications time but note that nesting and chaining calls between runners is not supported - use .cache / write interim results if this is needed. ruleEngineRunner's schema parameter when using all parameters is now Option[DataType], wrap in Some if you are using custom parameters for false etc.not relying on the old defaults. Deriving the type for ruleEngineRunner may work but you must use the type if control over nullability is required (for example expressions differ in nullability).

#87 - EOL DBR and Spark runtimes are removed: 9.1, 10.4, 11.3, 13.1, 14.0

#21 - Remove 2.4 Support, Tech Debt removal

#112 - Change overallResult to correctly reflect the ruleEngine processing (if any Passed is present then the overall should be Passed)

#115 - RuleFolder can use the DefaultProcessor from collectRunner, debug will return DefaultRuleSalience (Integer.MIN_VALUE) for the salience

#104 - RuleSuiteGroups allow nesting runners within both trigger and output expressions, a number of helper functions are provided to help you manage audit state

group_results can collect RuleSuiteResults over an array of nested runners (excluding expressionRunner) into RuleSuiteGroupResults, or indeed over RuleSuiteGroupResults and has an optional processing lambda over any results to save projections (e.g. using f -> flatten(f)) unify_result converts all engine runner results (folder, engine and collector) into the RuleFolderResult type (ruleSuiteResults: RuleSuiteResults, result: T) allowing combinations of engine results, however nested. Debug outputs etc. are disguarded.

0.1.4 24th February, 2026

This release provides a new runner type - collectRunner and a new RuleResult type of ignoredRule.

Similar to folder, in that it runs Output Expressions for each matching trigger Rule expression ordered by salience, it collects the result of each Output Expression.
It is optimised around optionally collecting and expanding nested arrays and auto expanding User LambdaFunctions.

If no matching trigger rule is found the new optional RuleSuite.DefaultProcessor can be run and the overallResult will reflect DefaultRule.
In 0.2.0 this default approach will also be extended to engine and folder runners (with an optional fallback configuration for the 0.2.x series).

0.1.4 was released in order to speed up delivery of 0.2.0 functionality for some key users. Please note, at time of publishing, many of the published Databricks and OSS versions are un-supported by their communities, please consider migrating versions to 14.3 LTS and Spark 3.5 at a minimum.

#107 - Introducing collectRunner - an optimised collecting rule engine

#111 - Introducing the notRelevant result - signal that a rule was not relevant for a row but is equivalent to a pass. Allows collecting statistics in three states, rows passed, rows failed and rows not relevant for a given rule.

#93 - Introducing the rule_suite_statistics aggregate function, which collects statistics over a RuleSuiteResult column

#114 - Further improvements to FunNRewrite logic, allowing both more control over the optimisation and more use cases where it can optimise.

0.1.3.1 24th October, 2025

This is the last release of 2.4, 3.0 is deprecated as of this release, similarly, Databricks versions 9.1 through to 11.3 are also now deprecated and unsupported functionality (fixes for #84 will be gladly accepted), 12.2 support will continue.

#95 - DBR 17.3 Support - Databricks introduced a binary change to NamedExpression

#85 - Processor optimisation to provide a passed result

#83 - Processor optimisations for stateless and Higher Order Functions

#82 - Wholestage codegen support for Correlated Subqueries, improved support for pass-through fields from plans

#81 - Enable Quality row level runners to be used outside a spark runtime (still requires spark to build of course)

#78 - Allow extra plans to be added after a rewrite, ConstantFolding as a default given it gives a slight boost.

#76 - DBR 16.3 support - Databricks introduced a number of API changes not found in Spark 4, extra UnresolvedFunction params. (also includes #75)

#75 - DBR 15.4 support - Databricks introduced nonVolatile, a breaking change affecting all StatefulLike/Nondeterministic (rngs, uuids, unique_id), there is also a regression wrt interpreted Spark encoders (returning Stream and incorrect results) - the test cases have moved to Frameless encoders.

#68 - Test setup improvements for running testShades on Fabric (reduced logging and share Databricks behaviour)

#69 - Use different scopes for OSS testShade builds for Fabric testing (bug in snakeyml usage)

#70 - map_with can now be used in groupBy aggregations

#71 - Leverage Spark Sub-expression Elimination:

In order to ensure behavioural compatibility this is not enabled on runners by default in 0.1.3.1.

To enable elimination ensure resolveWith = None (the default and not available in ExpressionRunner), compileEvals = false and forceRunnerEval = false (the default)

As part of this optimisation LambdaFunctions are rewritten and expanded as normal expression trees by a plan re-write. If this causes problems a `/* USED_AS_LAMBDA */` comment may be added to the LambdaFunction definition to disable this expansion for the entire sub-tree.

The entire rewrite plan must be enabled by calling `com.sparkutils.quality.enableFunNRewrites()` within your SparkSession or by default via the Quality extensions.

NB The use of re-writes with 3.2.x has been identified in one test case (testSimpleProductionRules) as problematic for codegen, please use more recent Spark versions.

#73 - Spark 4.0 support (with an upgrade to Shim 0.2.0 using sparkutils.frameless 1.0.0)

0.1.3 4th October, 2024

#53 - Docs parser is now more forgiving, empty descriptions are tolerated and normal scaladoc syntax is allowed

#50 - typedExpressionRunner - audited capture of expressions with the same type

#51 - Spark 3.5.0 support - NOTE ViewLoaderAnalysisException and MissingViewAnalysisException now have Exception causes

#27 - Delta 3.0.0 (Spark 3.5.0) support - compatible version

#55 - DBR 14.0/1 - Snake Yaml 2.0 support

#58 - Migrate custom runtime usage to Shim

#59 - DBR 13.3 LTS support

#57 - DBR 14.3 support

#61 - Use sparkutils frameless for 3.5, 13.3, 14.x builds - Due to encoding and shim changes this frameless fork version is not binary compatible with typelevel frameless proper

#62 - SPARK-47509 workaround for Subqueries in lambdas - most common patterns are supported with 4.0 / 14.3 DBR

#63 - Use actual struct functions where possible for drop_field/update_field functions - required due to 14.3 DBR introduced plan on local relations

#66 - Bug fix - softFail result handling was double encoded - softFail result type is changed to double (breaking)

#65 - Bug fix - Incorrect OverallResult and string result processing

0.1.2.1 4th September, 2023

Maven Central build issues, code wise the same as 0.1.2.

0.1.2 4th September, 2023

#48 - Bug fix - Enable Sub Queries in all runner types

0.1.1 9th July, 2023

#42 - Improve expression runner to store yaml, unlike json, to_yaml and from_yaml allow complete support for roundtripping of Spark data types

0.1.0 10th June, 2023

#29 - Quality OptimzerRule's run with Databricks sql display

#35 - agg_expr and associated lambda support in the functions package

#36 - improved update_field, added drop_field based on the Spark withField (3.4.1 impl)

#34 - simplified quality package usage, column functions are now under the functions package.

#32 - expressionRunner - saves the results of expressions as strings, suitable for aggregate statistics

#28 - rule_result function - retrieves a rule result directly from a dq or expressionRunner result

#15 - Addition of the loadXConfigs and loadX functions for maps and blooms, simplifying configuration management

#24 - Remove saferId / rowid functions - use unique_id where required

#18 - ViewLoader - simple view configuration via DataFrames

#30 - 3.3.2 and 3.4.1 builds - simple version bumps

#20 - 3.5.0 starting support

0.0.3 17th June, 2023

#25 - Use builtIn function registration by default - allows global views to be created using Quality functions

0.0.2 2nd June, 2023

#16 - Remove winutils requirements for testing and usage

#13 - Support 3.4's sub query usage in rules/trigger, output expressions and lambdas

#12 - Introduce the use of underscores instead of relying on camel case for function sql names, inline with Spark built-in functions

#10 - Base64 functions added for RowID encoding and decoding via base64 (more suitable for BI tools)

#9 - Add AsymmetricFilterExpressions with AsUUID and IDBase64 implementation, allows expressions used in field selects to be reversed, support added for optimiser rules through the SparkExtension

#8 - Add set syntax for easier defaulting sql, removing duplicative cruft from intention

#7 - SparkSessionExtension to auto register Quality functions - does not work in 2.4, starting with this release 2.4 support is deprecated

#6 - Simple as_uuid function

#5 - Spark 3.4 and DBR 12.2 LTS support

#4 - comparableMaps / reverseComparableMaps functions, allowing map comparison / set operations (e.g. sort, distinct etc.)

0.0.1 8th March, 2023

Initial OSS version.

(many internal versions in between)

the Quality exploration starts 25th April, 2020

Start of investigations into how to manage DQ more effectively within Spark and the mesh platform.

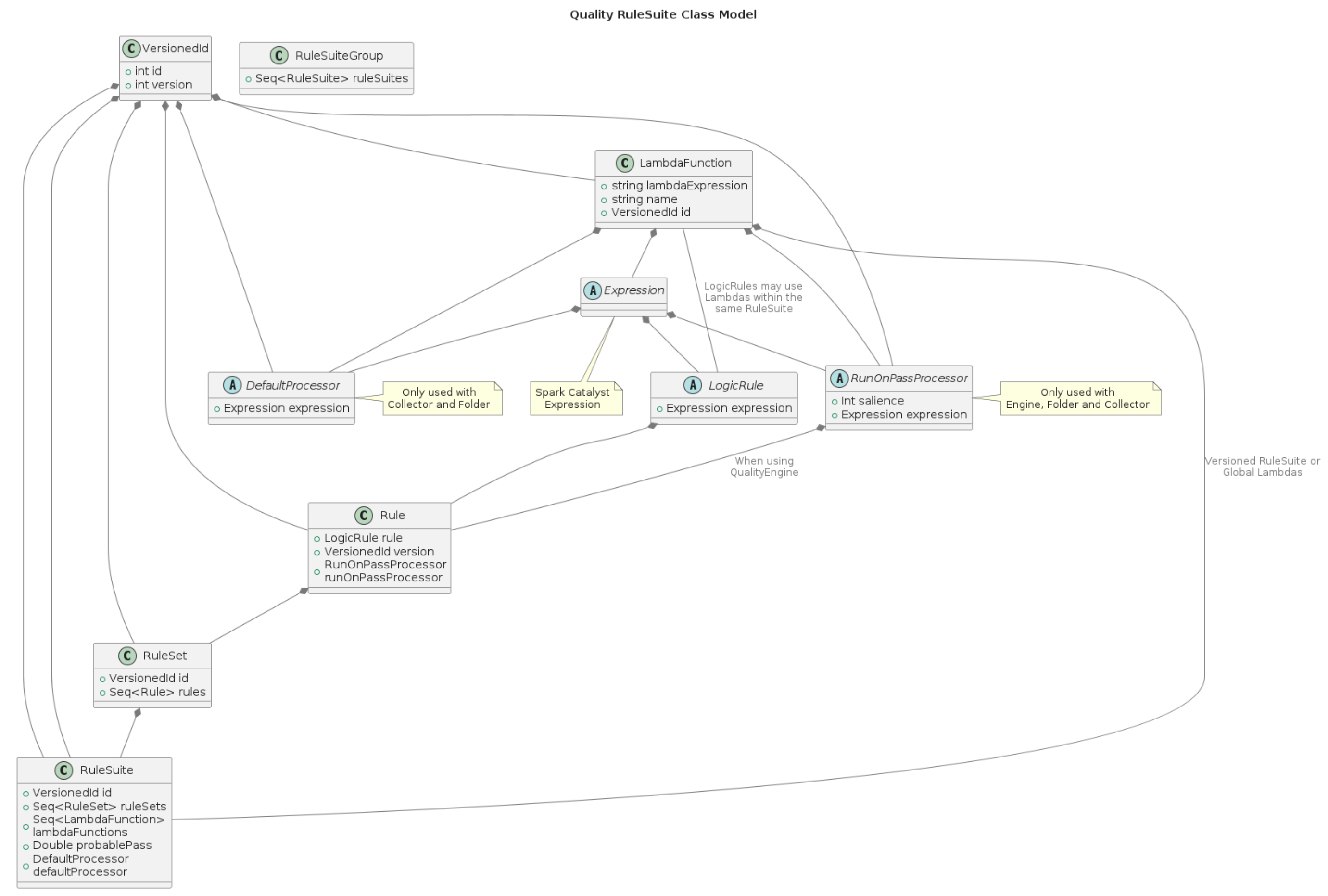
Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

4. Model

4.1 Rule Model

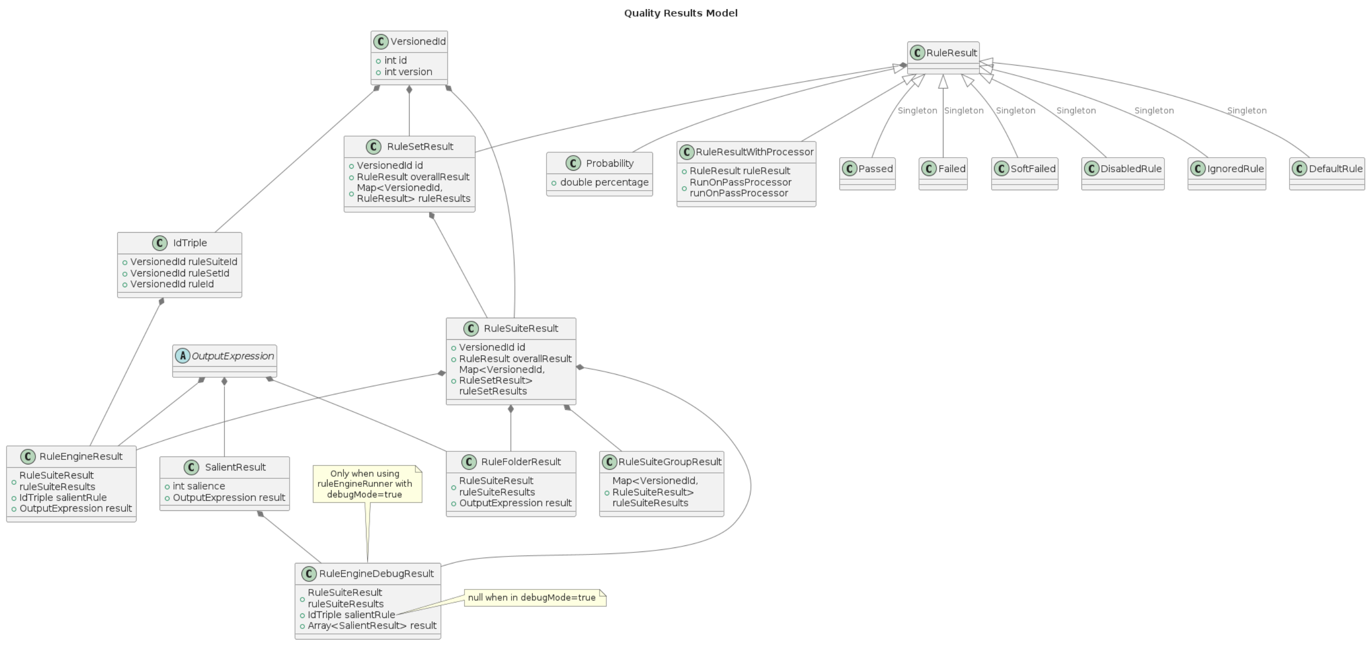
4.1.1 Rules



VersionedIDs are used throughout, changes to a Rule should imply a new Rule version, a new RuleSet version and a new RuleSuite version.

RunOnPassProcessor (output expressions) should only be provided when using the ruleEngineRunner and are treated, like Lambdas, as top level unique concepts. You should organise using output expressions wherever possible as it's not only easier to conceptualise but it's also faster.

4.1.2 Rule Results



- SoftFailed results do not cause the RuleSet or RuleSuite to fail
- DisabledRule results also do not cause the RuleSet or RuleSuite to fail but signal a rule has been disabled upstream
- Probability results with over 80 percent are deemed to have Passed, you may override this with the RuleSuite.withProbablePass function after creating the RuleSuite.
- IgnoredRule results also do not cause the RuleSet or RuleSuite to fail but signal a rule has been ignored upstream, typically to aid in reporting of applicable rules
- DefaultRule is used by Collector to indicate no trigger Rules Passed and defaultProcessor was run. (If no DefaultProcessor was used Failed is returned)

RuleResultWithProcessor is only used when using the ruleEngineRunner and is not returned in the column, rather the result of the expression is - shown above as call to "data".

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

4.2 Storage Model

Nested columns, with nested columns, this lets you use Spark SQL to do filters and have predicate pushdown. Sample filter:

```
df.select(expr("filter(map_values(DataQuality.ruleSetResults),
  ruleSet -> size(filter(map_values(ruleSet.ruleResults),
  result ->  probability(result) > 0.3 )) > 0)").as("filtered"))
```

actual type:

```
struct<id: LongType, overallResult: IntegerType,
  ruleSetResults: map<LongType,
    struct<overallResult: IntegerType,
           ruleResults: map<LongType, IntegerType>>>>
```

Alternatively when creating with addOverallResultsAndDetails you have the

```
overallResult: IntegerType
```

moved to the top level, leaving

```
details: struct<id: LongType,
  ruleSetResults: map<LongType,
    struct<overallResult: IntegerType,
           ruleResults: map<LongType, IntegerType>>>>
```

4.2.1 Where have all the VersionIds and RuleResults gone?

In order to optimise storage and marshalling the VersionId parts are packed into a single LongType. RuleResults are similarly encoded into an IntegerType:

- Failed => FailedInt // 0
- SoftFailed => SoftFailedInt // -1
- Disabled => DisabledInt // -2
- Ignored => IgnoredInt // -3
- DefaultRule => DefaultRuleInt // -4
- Passed => PassedInt // 100000
- Probability(percentage) => (percentage * PassedInt).toInt

When the developer wishes to retrieve the objects they may use the encoders directly:

```
// frameless is used to encode
import frameless._
// imports the encoders for RuleSuiteResult
import com.sparkutils.quality.implicits._
// derive an encoder for the pair with a user type and the RuleSuiteResult for a given row
implicit val enc = TypedExpressionEncoder[(TestIdLeft, RuleSuiteResult)]
// select the fields needed for the user type and the DataQuality result (or details with RuleResult, RuleSuiteResultDetails for separate overall results and
details)
val ds = df.selectExpr("named_struct('left_lower', `1`, 'left_higher', `2`)","DataQuality").as[(TestIdLeft, RuleSuiteResult)]
```

the developer can then integrate the data quality results alongside their relevant data.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

4.3 Meta Rulesets?

Quality introduces a "Meta Ruleset" approach for added automation. Meta Rule sets evaluate each column of a DataFrame to see if a Rule should be generated for that column.

Null checks, type checks etc. may all be applied generically without laboriously copying the rule for each applicable column, just define a single argument lambda expression. In order for this to work and be extensible you require stable ordering for each column used.

```scala
// if you wish to use Meta Rule Sets
val metaRuleSets = readMetaRuleSetsFromDF(metaRuleDF,
// an sql filter of the schema from a provided dataframe - name,
//datatype (as DDL) and nullable can be filtered
    col("columnFilter"),
// single arg lambda to apply to all fields from the column filter
    col("ruleExpr"),
    col("ruleSetId"),
    col("ruleSetVersion"),
    col("ruleSuiteId"),
    col("ruleSuiteVersion")
  )

// make sure we use the correct rule suites for the dataset, e.g.
val filteredRuleSuites: RuleSuiteMap = Map(ruleSuiteId -> rules)

val theDataframe = sparkSession.read.parquet("theFilePath")

// Guarantee each column always returns the same unique position
val stablePositionsFromColumnNames: String => Int = ???

// filter theDataframe columns and generate rules for each Meta
//  RuleSet and re-integrate them
val newRuleSuiteMap = integrateMetaRuleSets(theDataframe, filteredRuleSuites,
  metaRuleSets, stablePositionsFromColumnNames)
```

An optional last paramater for integrateMetaRuleSets allows transformation of a generated column dataframe, allowing joins with other lookup tables for the column definition or applicable rules to generate for the column for example.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

5. Advanced Usage

5.1 Bloom Filters

Bloom Filters are probabalistic data structures that, for a given number of items and a false positive probability (FPP) provides a mightContain function. This function *guarantees* that if an item is not in the bloom filter it will return false, however if it returns true this is to a probability defined by the FPP value.

In contrast to a Set which requires the items (or at least their hash values) to be stored individually blooms make use of multiple blocks and apply bit setting based on hashes of the input value over some function. These resulting blocks and bitsets are far smaller in memory and storage usage than a typical set. For example it's possible to store hundreds of millions of items within a bloom and still keep within a normal Java byte array boundary.

This act of using bit flipping also allows blooms to be or'd for the same size and FPP, which is great for aggregation functions in Spark.

Whilst blooms are great the guarantees break when:

1. The number of items far exceeds the initial size used to create the bloom - false is still guaranteed to not be present but the true value will no longer represent FPP, the bloom has degraded
2. The number of bits required to store the initial number of items at the FPP exceed what can be represented by the bloom algorithm.

If you attempt to store billions of items within a bloom at a high FPP you will quickly fall foul of 2, and this is easily done with both the Spark stats.package and the current bloom filters on Databricks. This makes them next to useless for large dataset lookups on *typical* bloom implementations.

5.1.1 How does Quality change this?

It can't change the fundamental laws of bloom filters, if you use the number of bits up your bloom filter is next to useless. You *can* however add multiple Java byte arrays and bucket the hashes across them. This works great up to about 1.5b items in a typical aggregation function within Spark, however Spark only allows a maximum of 2Gb for an InternalRow - of which aggregates are stored in.

Quality provides three bloom implementations the Spark stats package, small - which buckets within an InternalRow (1.2-1.5b items max whilst maintaining FPP) - and big which doesn't use Spark aggregations to store the results of aggregations but rather a shared file system such as Databricks dbfs.

Both the small and big bloom functions use Parquet's bloom filter implementation which both significantly faster and has better statistical properties than Sparks/Guavas or Breezes.

5.1.2 What are Bloom Maps?

Bloom Maps are identifiers to a bloom filter. The examples below show how to create the key is to use the SparkBloomFilter or bloomFilter functions to provide the value and the FPP is required.

```
registerBloomMapAndFunction(bloomFilterMap)
```

Both registers the Bloom Map, the small_bloom and big_bloom aggregation functions and the probabilityIn function.

5.1.3 Using the Spark stats package

```
// generate a dataframe with an id column
val df = sqlContext.range(1, 20)
// build a bloomfilter over the id's
val bloom = df.stat.bloomFilter("id", 20, 0.01)
// get the fpp and build the map
val fpp = 1.0 - bloom.expectedFpp()
val bloomFilterMap = SparkSession.active.sparkContext.broadcast( Map("ids" -> (SparkBloomFilter(bloom), fpp)) )
```

```
// register the map for this SparkSession
registerBloomMapAndFunction(bloomFilterMap)
// lookup the result of adding column's a and b against that bloom filter for each row
otherSourceDF.withColumn("probabilityInIds", expr("probability_in(a + b, 'ids')"))
```

The stats package bloomFilter function has severe limitations on a single field and does not allow expressions but through the SparkBloomFilter lookup function is integrated with Quality anyway.

5.1.4 Using the Quality bloom filters

The small and big bloom functions take a single expression parameter however it can be built from any number of fields or field types using the hash_with function.

- smallBloom(column, expected number of items, fpp) - an SQL aggregate function which generates a BloomFilter Array[Byte] for use in probabilityIn or rowId:

  ```
  val aggrow = orig.select(expr(s"small_bloom(uuid, $numRows, 0.01)")).head()
  val thebytes = aggrow.getAs[Array[Byte]](0)
  val bf = bloomLookup(thebytes)
  val fpp = 0.99
  val blooms: BloomFilterMap = Map("ids" -> (bf, fpp))
  ```

- bigBloom(column, expected number of items, fpp) - can only be run on large memory sized workers and executors and can cover billions of rows while maintaining the FPP:

  ```
  // via the expression
  val interim = df.selectExpr(s"big_bloom($bloomOn, $expectedSize, $fpp, '$bloomId')").head.getAs[Array[Byte]](0)
  val bloom = com.sparkutils.quality.BloomModel.deserialize(interim)
  bloom.cleanupOthers()

  val blooms: BloomFilterMap = Map("ids" -> (bloomLookup(bloom), fpp))

  // via the utility function, defaults to 0.01 fpp
  val bloom = bloomFrom(df, "id", expectedsize)
  val blooms: BloomFilterMap = Map("ids" -> (bloomLookup(bloom), 1 - bloom.fpp))
  ```

In testing the bigBloom creation over 1.5b rows on a small 4 node cluster took less than 8m to generate, using a resulting bloom however is far easier to load and distribute and constant time for lookups. Whilst the actual big bloom itself cannot be directly broadcast only the file location of the resulting bloom is and each node on the cluster directly loads it from the ADLS (or other hopefully fast store for the multiple GBs).

To change the base location for blooms use the sparkSession.sparkContext.setLocalProperty("sparkutils.quality.bloom.root") to specify the location root.

5.1.5 Bloom Loading

The interface and config row data types is similar to that of View Loader with loadBloomConfigs accepting these additional columns:

```
bigBloom: Boolean, value: String, numberOfElements: BIGINT, expectedFPP: DOUBLE
```

- bigBloom specifies which function should be used, when true the bigBloom algorithm will be used, when false the smallBloom.
- value is an expression string suitable for the bloom filter, the expression will not parse if the type is unsupported, complex types will need special handling but it's typically possible to convert to an array of longs via hash functions such as hash_with.
- numberOfElements is an estimated upper bound for the size of the bloom filter, too low, and many false possible results will be generated
- expectedFPP is the starting percentage of expected percentage of false positives produced, or what can be tolerated, a value of 0.01 implies 99% of the time you get a "should contain" result it will be accurate, and 0.01% of the time it won't be. When using too small an numberOfElements the expected fpp cannot be met. bigBloom will attempt to use both to derive the optimal size with the probability that the resulting fpp is different.

```
import sparkSession.implicits._

val (bloomConfigs, couldNotLoad) = loadBloomConfigs(loader, config.toDF(), expr("id.id"), expr("id.version"), Id(1,1),
  col("name"),col("token"),col("filter"),col("sql"), col("bigBloom"),
```

```
  col("value"), col("numberOfElements"), col("expectedFPP")
)
val blooms = loadBlooms(bloomConfigs)
```

with couldNotLoad holding a set of configuration rows that aren't possible to load (neither a DataFrameLoader token nor an sql).

loadBlooms will process the resulting dataframe using bigBloom, value, numberOfElements and expectedFPP to create the appropriate blooms. Views first loaded via view loader are available when executing the sql column (when token is null).

5.1.6 Expressions which take expression parameters

- probability_in(content to lookup, bloomfilterName) - returns the fpp value of a filter lookup against the bloomFilter with bloomFilterName in the registered BloomFilterMap, which works with the Spark stats package, small and big blooms.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

5.2 Map Functions

A typical use case for processing DQ rules is that of cached value processing, reference data lookups or industry code checks etc.

Quality's map functions reproduce the result of joining datasets but, once they are loaded, guarantees in memory operation only - no merges or joins required. However, for larger data lookups correlated subqueries or Bloom Filters should be preferred, or simply use joins.

Similarly, for cases involving more logic than a simple equality check you must use joins or starting in 3.4 (DBR 12.2) scalar sub queries, see View Loader for a way to manage the loading of views.

Map functions on 0.2.0 Spark 4

Map functions are now able to use stableName variable names and register multiple MapLookups (a simple String for use in SQLs), each function is changed accordingly adding stableName parameters registered as Spark 4 Variables:

```
def loadMaps(configs: Seq[MapConfig]): String
// gains the ability to specify the name to use, which returns stableName
def loadMaps(configs: Seq[MapConfig], stableName: String): String
// and, similarly,
def mapLookupsFromDFs(creators: Map[String, MapCreator]): String
// gains this overloaded version
def mapLookupsFromDFs(creators: Map[String, MapCreator], stableName: String): String
```

finally, the SQL lookup function itself requires the returned stableName:

```
map_lookup('mapid', expr, mapLookupsVar)
map_contains('mapid', expr, mapLookupsVar)
```

with registerMapLookupsAndFunction now becoming a no-op and is retained for compatibilty only.

5.2.1 Map Loading

The interface and config row data types is similar to that of View Loader with loadMapConfigs accepting these additional columns:

```
val (mapConfigs, couldNotLoad) = loadMapConfigs(loader, config.toDF(), expr("id.id"), expr("id.version"), Id(1,1),
  col("name"),col("token"),col("filter"),col("sql"),col("key"),col("value")
)

val maps = loadMaps(mapConfigs)
```

with couldNotLoad holding a set of configuration rows that aren't possible to load (neither a DataFrameLoader token nor an sql).

loadMaps will process the resulting dataframe using key and value as sql expressions in exactly the same way as mapLookupFromDFs, as such they must be valid expressions against the source dataframe. Views first loaded via view loader are available when executing the sql column (when token is null).

5.2.2 Building the Lookup Maps Directly

In order to lookup values in the maps Quality requires a map of map id's to the actual maps.

```
// create a map from ID to a MapCreator type with the dataframe and underlying
// columns, including returning structures / maps etc.
val lookups = mapLookupsFromDFs(Map(
      "countryCode" -> ( () => {
        val df = countryCodeCCY.toDF("country", "funnycheck", "ccy")
        (df, column("country"), functions)("struct(funnycheck, ccy)"))
      } ),
      "ccyRate" -> ( () => {
        val df = ccyRate.toDF("ccy", "rate")
        (df, column("ccy"), column("rate"))
      })
    ))
registerMapLookupsAndFunction(lookups)
```

In the countryCode map lookup case we are creating a map from country to a structure (funnycheck, ccy), whereas the ccyRate is a simple lookup between ccy and it's rate at point of loading.

Map creation is not lazy and is forced at time of calling the registerMap... function, for streaming jobs this may be unacceptable. Prefer to use new map id's and merge old sets if you need to guarantee repeated calls to registerMapLookupsAndFunctions are working with up to date data.

It's possible to have multiple fields used as the key, where all must match, just use struct in the same way as the value example above.

Note

Repeated calls and streaming use cases have not been thoroughly tested, the Spark distribution method guarantees an object can be broadcast but no merging is automatically possible, users would be required to code this by hand.

5.2.3 Expressions which take expression parameters

- map_lookup('map name', x) - looks up x against the map specified in map name, full type transparency from the underlying map values are supported including deeply nested structures

```
// show the map of data 'country' field against country code and get back the currency
df.select(col("*"), expr("map_lookup('countryCode', country).ccy")).show()
```

- map_contains('map name', x) - returns true or false if an item is present as a key in the map

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

5.3 Aggregation Functions

5.3.1 Aggregation Functions

Quality adds a number of aggregation primitives to allow building across dataset functionality similar to Deequ and others but philosophically staying true to the customisation approach used throughout the library.

You can aggregate using any number of agg_expr columns:

- agg_expr(ddl type, filter, lambda sum, lambda result) - allows filter expressions to be used to build up aggregated BIGINT (long) results with lambda functions, leveraging simple lambda functions (note count is currently only BIGINT / LongType):

```
// generates with an long id column from 1 to 20
val df = sparkSession.range(1, 20)
// filter odd numbers, add the them together with sumWith lambda for the
// sum, then using resultsWith lambda variables divide them by the count
// of filtered rows
val res = df.select(expr("agg_expr('BIGINT', id % 2 > 0, sum_with(sum -> sum + id), "+
  "results_with( (sum, count) -> sum / count ) )").as("aggExpr"))
res.show() // will show aggExpr with 10.0 as a result,
  // sum + count would show 110..
```

The filter parameter lets you select rows you care about to aggregate, but does not stop you aggregating different filters in different columns and still process all columns in a single pass. The sum function itself does the aggregation and finally the result function yields the last calculated result. Both of these functions operate on MAPs of any key and value type.

Spark lambda functions are incompatible with aggregation wrt. type inference which requires that the type is specified to agg_expr, it defaults to bigint when not specified.

The ExpressionRunner provides a convenient way to manage multiple agg_expr aggregations in a single pass action via a RuleSuite, just like DQ rules.

5.3.2 Aggregation Lambda Functions

- sum_with(lambda entry -> entry) - processes for each matched row the lambda with the given ddl type which defaults to LongType
- results_with(lambda (sum, count) -> ex) - process results lambda with sum and count types passed in.
- inc([expr]) - increments the current sum either by default 1 or by expr using type LongType
- meanF() - simple mean on the results, expecting sum and count type Long:

```
// generates with an long id column from 1 to 20
val df = sparkSession.range(1, 20)
// filter odd numbers, add the them together with inc lambda for the sum, then using meanF expression to divide them by the count of filtered rows
val res = df.select(expr("agg_expr(id % 2 > 0, inc(id), meanF() )").as("aggExpr"))
res.show() // will show aggExpr with 10.0 as a result, sum + count would show 110..
```

- map_with(keyExpr, x) - uses a map to group via keyExpr and apply x to each element:

```
// a counting example expr - group by and count distinct equivalent
expr("agg_expr('MAP<STRING, LONG>', 1 > 0, map_with(date || ', ' || product, entry -> entry + 1 ), results_with( (sum, count) ->
sum ) )").as("mapCountExpr")
// a summing example expr with embedded if's in the summing lambda for added fun
expr("agg_expr('MAP<STRING, DOUBLE>', 1 > 0, map_with(date || ', ' || product, entry -> entry + IF(ccy='CHF', value, value * ccyrate)  ),
return_sum() )").as("mapSumExpr")
```

- return_sum( ) - just returns the sum and ignores the count param, expands to results_with((sum, count) -> sum)

5.3.3 Column DSL

The same functionality is available in the functions package e.g.:

```
import com.sparkutils.quality.functions._
val df: DataFrame = ...
df.select(agg_expr(DecimalType(38,18), df("dec").isNotNull, sum_with(entry => df("dec") + entry ), return_sum) as "agg")
```

5.3.4 Type Lookup and Monoidal Merging

This section is very advanced but may be needed in a deeply nested type is to be aggregated.

Type Lookup

agg_expr, map_with, sum_with and return_sum all rely on type lookup. The implementation uses sparks in-built DDL parsing to get types, but can be extended by supplying a custom function when registering functions e.g.:

```
registerQualityFunctions(parseTypes = (str: String) => defaultParseTypes(str).orElse( logic goes here ) /* Option[DataType] */)
```

Monoidal Merging

Unlike type lookup custom merging could well be required for special types. Aggregation (as well as MapMerging and MapTransform) require a Zero value the defaultZero function can be extended or overwritten and passed into registerFunctions as per parseTypes. The defaultAdd function uses itself with an extension function parameter in order to supply map value monoidal associative add.

Note

This works great for Maps and default numeric types but it requires custom monoidal 'add' functions to be provided for merging complex types.

Whilst zero returns a value to use as zero you may need to recurse for nested structures of zero, add requires defining Expressions and takes a left and right Expression to perform it:

```
DataType => Option[( Expression, Expression ) => Expression]
```

Warning

This is an area of functionality you should avoid unless needed as it often requires deep knowledge of Spark internals. There be dragons.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

5.4 User Defined Functions

Users may register Lambda Functions using the sql lambda syntax:

```
val rule = LambdaFunction("multValCCY", "(theValue, ccy) -> theValue * ccy", Id(1,2))
registerLambdaFunctions(Seq(rule))
```

they may be then called in rules (or within any SQL expressions), in this case value and ccyrate from the data frame are provided to the function as parameters theValue and ccy:

```
val ndf = df.withColumn("newcalc", expr("multValCCY(value, ccyrate)"))
```

The function parameter and return types are derived during the analysis phase, this may lead to errors if types do not match the expressions upon an action only, such as writing or calling show.

Note

Whilst you are free to add lambdas when not using a RuleSuite the library will not ensure that only functions registered as part of a RuleSuite are used in rules, such hygiene is necessarily left to the user.

LambdaFunctions may have any number of parameters e.g. given a *greaterThan* lambda:

```
 (param1, param2) -> param1 > param2
```

you would be able to call it with two expressions

```
 greaterThan(col1, col2)
```

Single argument lambdas should not use brackets around the parameters and zero argument lambdas use no input or ->. In all cases the lambda can use the attributes from the surrounding dataframe - it's effectively global, you cannot use variables from surrounding / calling lambdas.

Don't use 'current'... as a lambda variable name on 2.4

Bizarrely this causes the parser to fail on 2.4 only, no more recent version suffers this. Same goes for left or right as names.

0.1.3.1 optimisations can be enabled

0.1.3.1 introduces the expansion of Qualitylambda functions, allowing sub expression elimination to take place. The entire rewrite plan must be enabled by calling `com.sparkutils.quality.enableFunNRewrites()` within your SparkSession or by default via the Quality extensions.

You can put the comment `/* USED_AS_LAMBDA */` in an individual rule definition to disable expansion for the entire user function subtree. This is unlikely to be needed, but is provided to allow specific overriding should issues arise. All rewrites can be disabled for a cluster by using quality_disable_optimiser_rules=com.sparkutils.quality.impl.extension.FunNRewrite config.

The use of re-writes with 3.2.x has been identified in one test case (testSimpleProductionRules) as problematic for codegen, please use more recent Spark versions.

5.4.1 Why do these exist when Spark supports SQL functions

In short: 1. Multiple-arity 2. Late type binding (outside of HoFs) 3. Lighter syntax 4. Higher Order Functions, use them in any appropriate Spark function like aggregates 4. Correlated subquery support when used as HoFs

5.4.2 What about default parameter or different length parameter length Lambdas?

To define multiple parameter length lambdas just define new lambdas with the same name but different argument lengths. You can freely call the same lambda name with different parameters e.g.:

```
val rule = LambdaFunction("multValCCY", "multValCCY(value, ccyrate)", Id(1,2))
val rule1 = LambdaFunction("multValCCY", "theValue -> multValCCY(theValue, ccyrate)", Id(2,2))
val rule2 = LambdaFunction("multValCCY", "(theValue, ccy) -> theValue * ccy", Id(3,2))
registerLambdaFunctions(Seq(rule, rule1, rule2))

// all of these should work
df.withColumn("newcalc", expr("multValCCY()"))
df.withColumn("newcalc", expr("multValCCY(value)"))
df.withColumn("newcalc", expr("multValCCY(value, ccyrate)"))
```

5.4.3 Higher Order Functions

As Lambda's in Spark aren't first class citizens you can neither partially apply them (fill in parameters to derive new lambdas) nor pass them into a lambda.

Quality experimentally adds three new concepts to the mix:

1. Placeholders - `_()` - which represents a value which still needs to be filled (partial application)
2. Application - `callFun()` - which, in a lambda, allows you to apply a function parameter
3. Lambda Extraction - `_lambda_()` - which allows Lambdas to be used with existing Spark HigherOrderFunctions (like aggregate)

Unfortunately the last piece of that puzzle of returning a higher order function isn't currently possible.

Putting together 1 and 3 (straight out of the test suite):

```
val plus = LambdaFunction("plus", "(a, b) -> a + b", Id(1,2))
val plus3 = LambdaFunction("plus3", "(a, b, c) -> a + b + c", Id(2,2))
val hof = LambdaFunction("hof", "func -> aggregate(array(1, 2, 3), 0, _lambda_(func))", Id(3,2))
registerLambdaFunctions(Seq(plus, plus3, hof))

import sparkSession.implicits._

// attempt to dropping a reference to a function where simple lambdas are expected.
// control
assert(6 == sparkSession.sql("SELECT aggregate(array(1, 2, 3), 0, (acc, x) -> acc + x) as res").as[Int].head)
// all params would be needed with multiple aritys
assert(6 == sparkSession.sql("SELECT aggregate(array(1, 2, 3), 0, _lambda_(plus(_('int'), _('int')))) as res").as[Int].head)
// can we play with partials?
assert(21 == sparkSession.sql("SELECT aggregate(array(1, 2, 3), 0, _lambda_(plus3(_('int'), _('int'), 5))) as res").as[Int].head)
// hof'd
assert(6 == sparkSession.sql("SELECT hof(plus(_('int'), _('int'))) as res").as[Int].head)
```

In the above example you can see type's being specified to the placeholder function, this is needed because, similar to aggExpr, Spark can't know the types until after they are evaluated and resolved. This does have the benefit of keeping the types at the partial application site. *The default placeholder type is Long / Bigint*.

The *lambda* function extracts a fully *resolved* underlying Spark LambdaFunction, which means the types must be correct as it is provided to the function (use the placeholder function to specify types). Similarly, you use the *lambda* function to extract the Spark LambdaFunction from a user provided parameter (as seen in the hof example).

The aggregate function only accepts two parameters for its accumulator, but in the plus3 example we've 'injected' in a third. Partially applying the plus3 with the value 5 in it's "c" position leaves the two arguments as new function. Quality ensures the necessary transformations are done before it hits the aggregate expression.

Great, but can I use it with aggExpr? Yep:

```
select aggExpr('DECIMAL(38,18)', dec IS NOT NULL, myinc(_()), myretsum(_(), _())) as agg
```

allows you to define the myinc and myretsum elsewhere, you don't need to use the *lambda* function with aggExpr.

What about application? Using callFun:

```
val use = LambdaFunction("use", "(func, b) -> callFun(func, b)", Id(4,2))
```

the first parameter must be the lambda variable referring to your function followed by the necessary parameters to pass in. Func in this case has a single parameter but of course it could have started with 5 and had 4 partially applied. Again you don't need to use *lambda* to pass the functions further down the line:

```
val deep = LambdaFunction("deep", "(func, a, b) -> use(func, a, b)", Id(2,2))
```

Deep takes the function and simply passes it to use where the callFun exists.

Finally, you can also further partially apply your lambda variables:

```
val plus2 = LambdaFunction("plus", "(a, b) -> a + b", Id(3,2))
val plus3 = LambdaFunction("plus", "(a, b, c) -> plus(plus(a, b), c)", Id(3,2))
val papplyt = LambdaFunction("papplyt", "(func, a, b, c) -> callFun(callFun(func, _(), _(), c), a, b)", Id(2,2))
registerLambdaFunctions(Seq(plus2, plus3, papplyt))

import sparkSession.implicits._

assert(6L == sparkSession.sql("select papplyt(plus(_(), _(), _()), 1L, 2L, 3L) as res").as[Long].head)
```

Here the callFun directly applies the function afterwards but you could equally pass it to other functions.

```
callFun(callFun(func, _(), _(), c), a, b)
```

can then be read as partially apply func (plus with 3 arguments) parameter 3 with the lambda variable c, creating a new two argument function. Then call that function with the a and b parameters. Useless in this case perhaps but it should be illustrative.

All that's missing is returning lambdas:

```
val plus2 = LambdaFunction("plus", "(a, b) -> a + b", Id(3,2))
val plus3 = LambdaFunction("plus", "(a, b, c) -> plus(plus(a, b), c)", Id(3,2))
val retLambda = LambdaFunction("retLambda", "(a, b) -> plus(a, b, _())", Id(2,2))
registerLambdaFunctions(Seq(plus2, plus3, retLambda))

import sparkSession.implicits._

assert(6L == { val sql = sparkSession.sql("select callFun(retLambda(1L, 2L), 3L) as res")
  sql.as[Long].head})
```

here the user function retLambda returns the plus with 3 arity applied over a and b, leaving a function of one arity to fill. The top level callFun then applies the last argument (c).

It's sql only

As you can create your own functions based on Column transformations the functionality is not extended to the dsl where there are better host language based solutions.

It is experimental

Although behaviour has been tested with compilation and across the support DBRs it's entirely possible there are gaps in the trickery used.

A good example of the experimental nature is the _() function, it's quite possible that is taken by Spark at a later stage.

***lambda* drop in call arguments to transform_values and transform_keys don't work on 3.0 and 3.1.2/3**

They pattern match on List and not seq, later versions fix this. To work around this you must explicitly use lambdas for these functions.

Do not mix higher order functions with subqueries

Per SPARK-47509 subqueries within a HOF can lead to correctness issues, such usage is not supported

5.4.4 Controlling compilation - Tweaking the Quality Optimisations

Normal Spark LambdaFunctions, NamedLambdaVariable and HigherOrderFunctions aren't compiled, this is - in part - due to the nature of having to thread the lambda variables across the Expression tree and calling bind.

At the time of codegen bind has already been called however so the code is free to create a new tree just for compilation. Quality makes use of this and replaces all NamedLambdaVariables expressions with a simple variable in the generated code.

NamedLambdaVariables also use AtomicReferences, which was introduced to avoid a tree manipulation task - see here for the code introduction. AtomicReferences are slower for both writes and reads of non-contended variables. As such Quality does away with this in its compilation, the exprId is sufficient to track the actual id.

Quality only attempts to replace it's own FunN and reverts to using NamedLambdaVariables if it encounters any other HigherOrderFunction. Where it can replace it uses NamedLambdaVariableCodeGen with an ExprId specific code snippet.

You can customise this logic via implementing:

```
  trait LambdaCompilationHandler {
    /**
     *
     * @param expr
     * @return empty if the expression should be transformed (i.e. there is a custom solution for it).  Otherwise return the full set of NamedLambdaVariables
found
     */
    def shouldTransform(expr: Expression): Seq[NamedLambdaVariable]

    /**
     * Transform the expression using the scope of replaceable named lambda variable expression
     * @param expr
     * @param scope
     * @return
     */
    def transform(expr: Expression, scope: Map[ExprId, NamedLambdaVariableCodeGen]): Expression
  }
```

and supplying it via the environment variable, System.property or via sparkSession.sparkContext.setLocalProperty quality.lambdaHandlers using this format:

```
name=className
```

where name is either a fully qualified class name of a HigherOrderFunction or of a lambda (FunN) function.

The default org.apache.spark.sql.qualityFunctions.DoCodegenFallbackHandler allows you to disable any optimisation for a HigherOrderFunction. It can be used to disable all FunN optimisations with:

```
-Dquality.lambdaHandlers=org.apache.spark.sql.qualityFunctions.FunN=org.apache.spark.sql.qualityFunctions.DoCodegenFallbackHandler
```

Alternatively if you have a hotspot with any inbuilt HoF such as array_transform, filter or transform_values you could replace the implementation for compilation with your own transformation. e.g.:

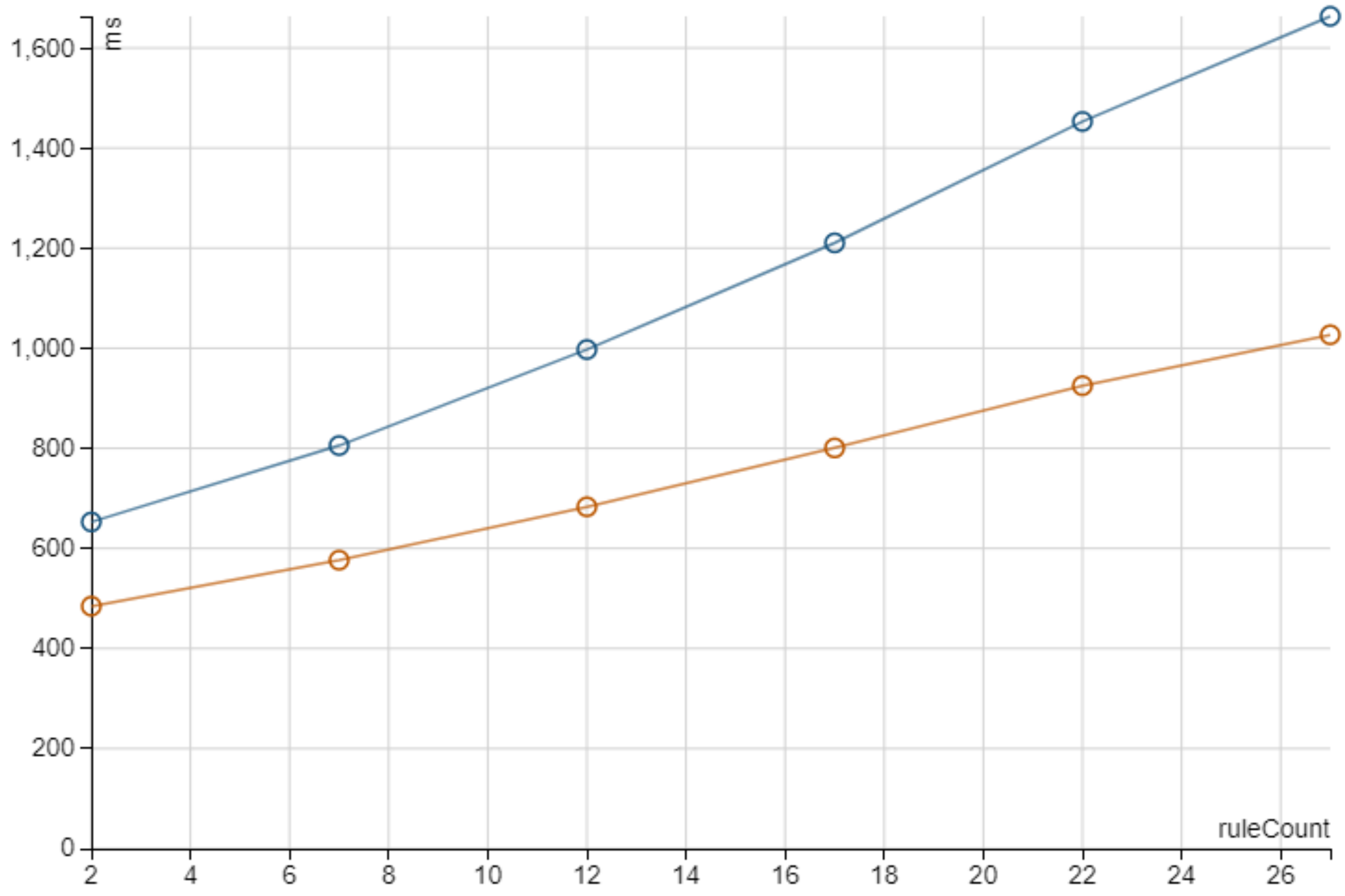
```
-Dquality.lambdaHandlers=org.apache.spark.sql.catalyst.expressions.TransformValues=org.mine.SuperfastTransformValues
```

Handlers disable FunNRewrite

The FunNRewrite optimisation lifts lambdas out to higher level expressions, enabling sub expression elimination. This behaviour can be disabled by using /* USED_AS_LAMBDA */ as a comment within your user function definition, the same occurs when a Spark Higher Order Function, such as transform/ArrayTransform, is used with a handler. This allows the handlers to be run, compiling out the HOF, as a trade-off to possible gains from sub expression elimination. Future versions of Spark may compile out the HigherOrderFunctions removing this limitation.

Why do all this?

Speed, it's up to 40% faster. LambdaRowPerfTest, in the test suite, generates an increasing number of lambdas and only runs over 10k rows but still sees clear benefits e.g. (orange is compiled lambdas):



This difference is already noticeable with a small increment function in a folder:

```
thecurrent -> updateField(thecurrent, 'thecount', thecurrent.thecount + 1)
```

The difference is typically higher with nested lambdas. Should your compilation time exceed the execution time you may wish to disable compilation via the fallback handler.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

5.5 PRNG Functions

The existing Spark rand function has a few of limitations:

- It generates doubles
- Has a fixed implementation
- Only provides reseeding on each new parition ignoring splittable / jumpable algorithms

The Quality psuedorandom generators produce either 128bit values (two longs) or a configurable number of bytes and, as a result, do not suffer precision issues, they also leverage RandomSource implementations allowing users to choose the algorithm used.

In addition, by leveraging .isJumpable and the resulting jump function the Quality prng function can benefit from the implementations own approach to managing overalapping intervals across the cluster.

5.5.1 RNG Expressions

- rng_bytes([number of bytes to fill - defaults to 16], [RandomSource RNG Impl - defaults to 'XO_RO_SHI_RO_128_PP'], [seed - defaults to 0]) - Uses commons rng to create byte arrays, implementations can be plugged in, when seed is 0 the RNG's default seed generator is used. Note when a given RNG `isJumpable` then it will use jumping for each partition where possible both improving speed and statistical results.
- rng([RandomSource RNG Impl - defaults to 'XO_RO_SHI_RO_128_PP'], [seed - defaults to 0]) - Uses commons rng to create byte arrays, implementations can be plugged in, when seed is 0 the RNG's default seed generator is used. Note when a given RNG `isJumpable` then it will use jumping for each partition where possible both improving speed and statistical results.
- rng_uuid(expr) - processes expr with either byte arrays or two longs into a UUID string, it's counterpart long_pair_from_uuid generates two longs

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

5.6 Row ID Functions

Row ID functions are at least 160bit, made of a lower base id and two longs. There are 4 distinct implementations:

1. Random Number, a 128bit payload based on XO_RO_SHI_RO_128_PP
2. Field Based, 128bit MD5 payload based on fields e.g. for DataVault style approaches
3. Provided, an Opaque ID payload, typically 128bit, provided by some upstream system fields (MD5 is not used under the hood)
4. Guaranteed Unique, 160bit ID based on Twitters snowflake IDs at Spark scale - requires MAC addresses to be stable and unique on a driver

These IDs use the "base" field to provide extensibility but comparisons must include all three fields (or more longs should they be added).

From a performance perspective you should transform the column to make the structure into top-level fields via

```
selectExpr("*","myIDField.*").drop("myIDField")
```

- rng_id('prefix') - generates a Random 128bit number with each column name prefixed for easy extraction
- unique_id('prefix') - generates a unique 160bit ID with each column name prefixed for easy extraction
- field_based_id('prefix', 'messagedigest', exp1, exp2, *) - generates a digest based e.g. 'MD5' identifier based on an expression list
- provided_id('prefix', longArrayBasedExpression) - generates a providedID based on supplied array of two longs expression
- murmur3_id('prefix', exp1, exp2, *) - generates and ID using hashes based on a version of murmur3 - not cryptographically secure but fast
- id_equal('left_prefix', 'right_prefix') - (SQL only) tests the two top level field IDs by adding the prefixes, note this does allow predicate push-down / pruning etc. (NB further versions may be added when 160bit is exceeded)

Id's can be 96-bit or larger multiples of 64

The algorithm you chose to use for generating Ids will change the length of underlying longs, idEqual cannot be used on different lengths but you can easily replace this with a lambda of the correct length.

There are many different hash impls

The fieldBasedID functions have a family of alternatives for MessageDigest, ZA based hashes and Guava based Hashers. See SQL Functions and look for the Hash and ID tags.

fieldBasedID with MD5 - Seems far slower than other approaches

It's definitely slower than either uniqueId or rngID. If your use case allows it, consider murmur3ID if this is sufficient, it's slightly faster as is the XXH3 za hash. MD5 was chosen based on the ubiquity of implementations including on backends (e.g. allowing datavault style approaches).

Guaranteed Unique ID - How?

In order to lock down a globally (within a Spark using routable IP address space) ID you need to make sure a given machine, point in time and partition (thread) is unique.

Your networking / vendor setup should guarantee the machines MAC Address is unique for your Spark Driver, Spark guarantees that the partition id, although re-usable, does not get re-used within a Spark cluster and for a given ms since an epoch we can lock down a range of row numbers. This leaves the following storage model:

```
gantt
    dateFormat YYYY-MM-DD
        axisFormat %j
    title       Bit Layout
        todayMarker off

    section First Int
    Unique ID Type and Reserved Space :active, start, 2021-01-01, 8d
    First 3 Bytes of MAC               :  startmac, after start, 24d

    section First Long
    Last 3 Bytes of MAC                :endmac, after startmac, 24d
    Spark Partition      :partition, after endmac, 32d
    First 8 bits of Timestamp    :starttimestamp, after partition, 8d

        section Second Long
        Rest of Timestamp       :done, endtimestamp, after starttimestamp, 33d
        Row number in Partition :rowid, after endtimestamp, 31d
```

When Spark starts a new partition the uniqueID expression resets the timestamp and partition and each row evaluates the rowid. When 32bits of rowid would be hit the timestamp is reset and the count resets to 0 allowing over a billion rows per ms.

This approach is faster than rngID but also means rows written to the same partitions have statistically incrementing id's allowing Parquet statistical ranges to be used for all three values in predicate pushdowns.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

5.7 QualityRules

5.7.1 Engine

Quality provides a basic rule engine for data quality rules the output of each rule however is always translated to RuleResult, encoded and persisted for audit reasons. If *any* rule trigger matches the RuleSuiteResult.overallResult will be Passed otherwise Failed is returned.

The ruleEngineRunner function however allows you to take an action based on the passing of a rule and, via salience, choose the most appropriate output for a given row.

You can understand QualityRules as a large scale auditable SQL case statement with "when" being the trigger rule and the "then" as the output expression.

RuleSuites are built per the normal DQ rules however a RuleResultProcessor is supplied:

```
val ruleResultProcessor =
  RunOnPassProcessor(salience, Id(outputId, outputVersion), RuleLogicUtils)("array(account_row('from', account), account_row('to', 'other_account1'))")))
val rule = Rule(Id(id, version), expressionRule, ruleResultProcessor)
val ruleSuite = RuleSuite(Id(ruleSuiteId, ruleSuiteVersion), Seq(
    RuleSet(Id(ruleSetId, ruleSetVersion), Seq(rule)
    )))

val rer = ruleEngineRunner(ruleSuite,
    DataType.fromDDL("ARRAY<STRUCT<`transfer_type`: STRING, `account`: STRING>>"))

val testDataDF = ...

val outdf = testDataDF.withColumn("together", rer).selectExpr("*", "together.result")
```

The ruleEngineRunner takes a DataType parameter that must describe the type of the result column type. An additional salientRule column is available that packs three the Id's that represent the ruleId chosen by salience. If this is null then *no* rule was triggered and the output column will also be null (verifiable via debug mode), if however there is an entry but the output is null then this signifies that the output expression produced a null.

The salientRule column may be pulled apart down to the id number and versions via the unpack expression or unpackIdTriple to unpack the lot in one go. If you are using frameless encoders these longs can be converted to a triple of Id's.

The salience parameter to the RunOnPassProcessor is used to ensure the lowest value is returned for a ruleSuite. It is the responsibility of the rule configuration to ensure there can only be one output.

All of the existing functionality, lambdas etc. can be used to customise the results and, as per the normal DQ processing, is run in-process across the clusters when the spark action is taken (like writing the dataframe to disk).

Serializing

The serializing approach uses the same functions as normal DQ RuleSuites, the only difference is you should use toDS and provide the two additional ruleEngine parameters when reading from a DF:

```
val withoutLambdasAndOutputExpressions = readRulesFromDF(rulesDF,
    col("ruleSuiteId"),
    col("ruleSuiteVersion"),
    col("ruleSetId"),
    col("ruleSetVersion"),
    col("ruleId"),
    col("ruleVersion"),
    col("ruleExpr"),
    col("ruleEngineSalience"),
    col("ruleEngineId"),
    col("ruleEngineVersion")
  )

val lambdas = ...

val outputExpressions = readOutputExpressionsFromDF(so.toDF(),
  col("ruleExpr"),
  col("functionId"),
  col("functionVersion"),
  col("ruleSuiteId"),
  col("ruleSuiteVersion")
)

val (ruleMap, missing) = integrateOutputExpressions(withoutLambdasAndOutputExpressions, outputExpressions)
```

The ruleExpr is only run for the lowest ruleEngineSalience result of any passing ruleExpr. The missing result will contain any output expressions specified by a rule which do not exist in the output expression dataframe based by rulesuite id, if your rulesuite id is not present in the missing entries your RuleSuite is good to go.

The rest of the serialization functions to combine lambdas etc. work as per normal DQ rules allowing you to use lambda functions in your QualityRules output rules as well.

The result of toDS will contain the three ruleEngine fields, you can simply drop them if they are not needed.

Debugging

The RuleResult's indicate if a rule has not triggered but in the case of multiple matching rules it can be useful to see which rules would have been chosen.

To enable this you can add the debugMode parameter to the ruleEngineRunner:

```
val rer = ruleEngineRunner(ruleSuite,
    DataType.fromDDL("ARRAY<STRUCT<`transfer_type`: STRING, `account`: STRING>>"),
    debugMode = true)
```

This changes the output column 'result' field type to:

```
ARRAY<STRUCT<`salience`: INTEGER, `result`: ARRAY<ORIGINGALRESULTTYPE>>
```

Why do I have a null

There are two cases where you may get a null result:

1. no rules have matched (you can verify this as you'll have no passed() rules).
2. your rule actually returned a null (you can verify this by putting on debug mode, you'll see a salience but no result)

flatten_rule_results

```
val outdf = testDataDF.withColumn("together", rer).selectExpr("explode(flatten_rule_results(together)) as expl").selectExpr("expl.*")
```

This sql function behaves the same way as per flatten_results, however there are now two structures to 'explode'. debugRules works as expected here as well.

resolveWith

Use with care - very experimental

The resolveWith functionality has several issues with Spark compatibility which may lead to code failing when it looks like it should work. Known issues:

1. Using filter then count will stop necessary attributes being produced for resolving, Spark optimises them out as count doesn't need them, however the rules definitely do need some attributes to be useful.
2. You may not select different attributes, remove any, re-order them, or add extra attributes, this is likely to cause failure in show'ing or write'ing
3. Spark is free to optimise other actions than just count, ymmv in which ones work.
4. The 0.1.0 implementation of update_field (based on the Spark impl) does not work in some circumstances (testSimpleProductionRules, testSalience will fail) - see #36

resolveWith attempts to improve performance of planning for general spark operations by first using a reduced plan against the source dataframe. The resulting Expression will have all functions and attributes resolved and is hidden from further processing by Spark until your rules actually run.

```
val testDataDF = ....

val rer = ruleEngineRunner(ruleSuite,
    DataType.fromDDL(DDL), debugMode = debugMode, resolveWith = resolveWith = Some(testDataDF))

val withRules = rer.withColumn("ruleResults", rer)

// ... use the rules
```

WHY IS THIS NEEDED?

For RuleSuites with 1000s of triggers the effort for Spark to prepare the rules is significant. In tests 1k rule with 50 field evalutaions is already sufficient to cause a delay of over 1m for each action (show, write, count etc.) and the size of the data being processed is not relevant.

After building the action QualityRules scale and perform as expected, but that initial costs of 1m per action is significant as it can only be improved by higher spec drivers.

resolveWith, if it works for given use case, drastically reduces this cost, the above 1k example is a 30s evaluation up front and far less cost for each further action.

With the rather horrible 1k rule example the clock time of running 1k rows through 1k rules with a simple show, then count and write for actions was 6m15s on an Azure b4ms, using resolveWith brings this down to 1m30s for the same actions. Still not blazingly fast of course, but far more tolerable and becomes suitable for smaller batch jobs.

ANY REASON WHY I SHOULDN'T TRY IT?

Not really but for production use cases where your trigger and output rules complexity is low you should prefer to not use it, it's likely fast enough and this solution is very much experimental.

You definitely shouldn't use it when using relation or table fields in your expressions e.g. table.field this does not work (verify this by running JoinValidationTest using evalCodeGens instead of evalCodeGensNoResolve). There be dragons. This is known to fail on *all* OSS builds and OSS runtimes (up to and including 3.2.0). 10.2.dbr and 9.1.dbr *actually do work* running the tests in notebooks with resolveWith and relations (the test itself is not built for this however to ensure cross compilation on the OSS base).

forceRunnerEval

By default, QualityRules runs with an optimised wholestage codegen wherever possible. This works by breaking out the nested structure of a RuleSuite into multiple index, salience and id arrays which are fixed for the duration of an action. Whilst this reduces the overhead of array and temporary structure creation the compilation also unrolls the evaluation of trigger rules allowing jit optimisations to kick in.

Using large RuleSuites, however, may cause large compilation times which are unsuitable for smaller batches, as such you can force the interpreted path to be used by setting this parameter to true. Individual trigger and output expressions are still compiled but the evaluation will not be.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

5.7.2 Workflow

Overview and terms

QualityRules is a matching engine which applies match/trigger rules to a Dataframe and, when these rules evaluate to passed (i.e. they match or trigger) output sql is run.

Only one trigger rule may produce output, so salience is used as a tie-breaker, the lowest salience wins.

Aim to have unique salience for tie-breaking

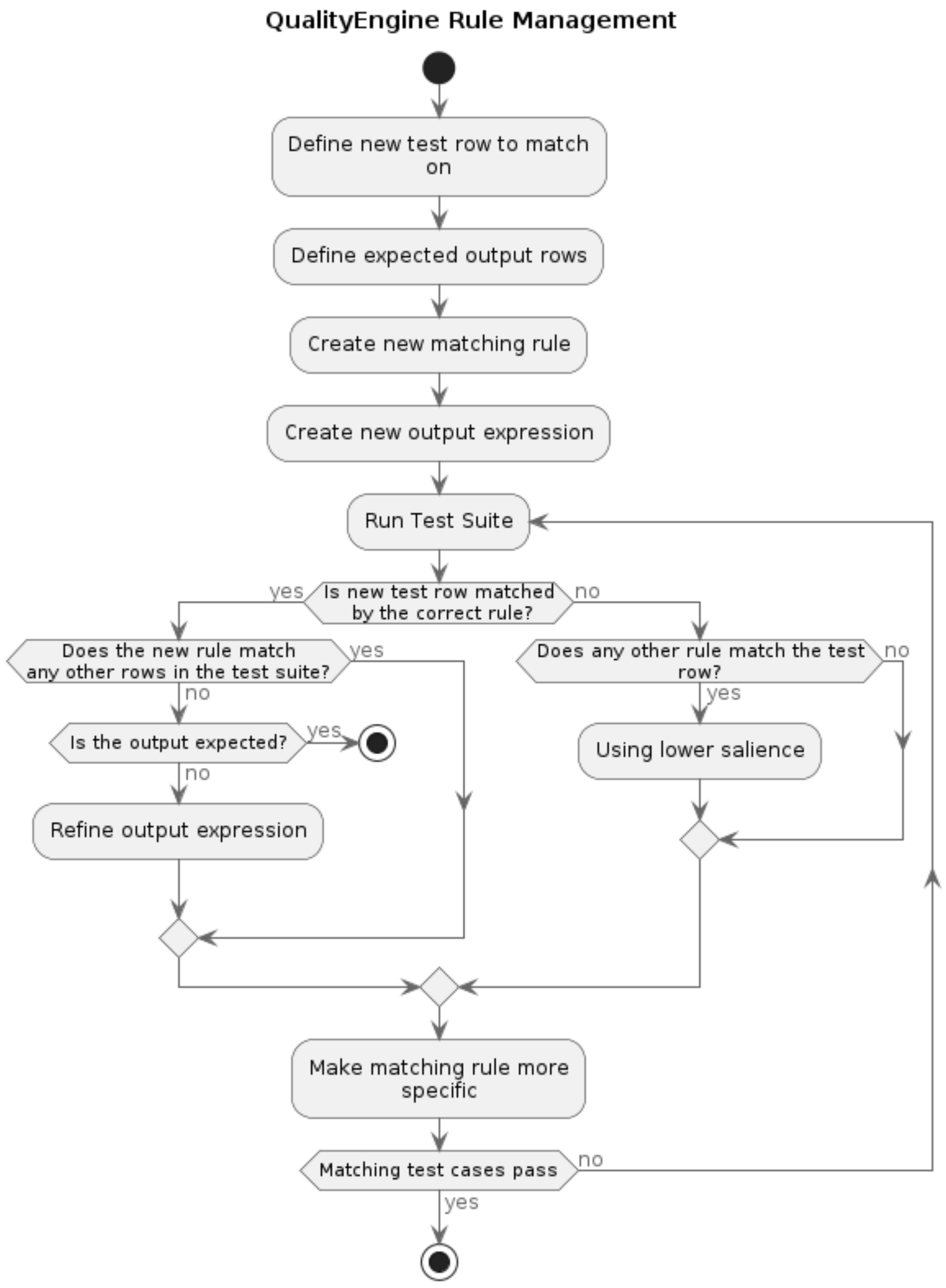
If you have multiple trigger rules with the same salience that both trigger the "winning" output chosen is non-deterministic, chose your salience wisely.

An alternative way to think of this is the trigger rules are your if and the output expressions are the when, from a logic perspective it may be helpful to think of them as output verbs - when this is true do that.

Suggested approach to QualityRules management

- Keep unrelated rules in their own RuleSuites, making things easier to reason about
- Make commonly used lambdas or output expressions global
- Use descriptive verbs for your output expressions
- Keep duplication or complexity in lambdas
- Only use fields that change as parameters to those lambdas
- Always *start* with test data you want to match against *and* your expected output
- Run all test cases for your RuleSuite for any change, don't assume because your rule worked that others won't stop working
- Use the validation and documentation functionality to document your lambdas and verify you've not made simple mistakes - Spark errors aren't always easy to understand

This could be visualised as such:


QualityEngine Rule Management
Define new test row to match on
Define expected output rows
Create new matching rule
Create new output expression
Run Test Suite
Is new test row matched by the correct rule?
yes
no
Does the new rule match any other rows in the test suite?
yes
no
Is the output expected?
yes
no
Refine output expression
Does any other rule match the test row?
no
yes
Using lower salience
Make matching rule more specific
Matching test cases pass
no
yes

Don't repeat yourself

If you are typing the same trigger rule, output expression or even lambda text repeatedly - make another lambda and consider making it global

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

5.8 QualityFolder

The ruleFolderRunner function uses the same data formats and structures as the ruleEngineRunner (with the exception of RuleFolderResult) however it allows you to "fold" results over many matching rules. If no triggers match, then an optional defaultProcessor on the RuleSuite can be run. If *any* rule trigger matches the RuleSuiteResult.overallResult will be Passed, if a DefaultProcessor is present DefaultRule will be returned, otherwise Failed is returned.

In contrast to ruleEngineRunner, which uses salience to select which output expression to run, ruleFolderRunner uses salience to order the execution of each matching output expression. To facilitate this OutputExpressions in the ruleFolderRunner must be lambdas with one parameter.

ruleFolderRunner takes a starter Column, which is evaluated against the row and then is passed as the parameter to the OutputExpression lambdas, in turn the result of these output lambdas is then fed in to the next matching OutputExpression and folded over until the last is run, which is returned.

When using debugMode you get the salience and each output returned in the resulting array, as with ruleEngineRunner the Encoder derivations for RuleFolderResult work with both T and Seq[(Int, T)] where the Int is salience. Where no rules match, unless a DefaultProcessor is provided, the debug output will be null; where a DefaultProcessor is provided (DefaultRuleSalience, default result) is returned in the array.

RuleSuites are built per the normal DQ rules however a RuleResultProcessor is supplied with Lambda OutputExpressions:

```
val ruleResultProcessor =
  RunOnPassProcessor(salience, Id(outputId, outputVersion),
    RuleLogicUtils)("thecurrent -> update_field(thecurrent, 'account', concat(thecurrent.account, '_suffix') )")))
val rule = Rule(Id(id, version), expressionRule, ruleResultProcessor)
val ruleSuite = RuleSuite(Id(ruleSuiteId, ruleSuiteVersion), Seq(
    RuleSet(Id(ruleSetId, ruleSetVersion), Seq(rule)
    )))

val rer = ruleFolderRunner(ruleSuite,
    struct($"transfer_type", $"account"))

val testDataDF = ...

val outdf = testDataDF.withColumn("together", rer).selectExpr("*", "together.result")
```

You may use multiple path and expression combinations in the same call, allowing the change of multiple fields at once - this will be faster than nesting calls to updateField.

Don't use select(*, ruleFolderRunner)

Spark will not NPE using withColumn but will using select(expr("*"), ruleFolderRunner(ruleSuite)). In order to thread the types through the resolving needs an additional projection, if you must avoid withColumn (e.g for performance reasons) then you may specify the DDL via the useType parameter.

5.8.1 Set

Although the use of lambda expressions allows you full control of your output expression it can be a bit verbose. The common use case of defaulting is more easily expressed via the following syntax:

```
set( variable.path = expression to assign, variable2 = other expression, variable3 = expression using currentResult )
```

Only valid variable names and paths, followed by equal and valid expressions (however complex) are allowed.

The following two folder expressions are equivalent, indeed the set call is translated into the lambda:

```
set( account = concat(currentResult.account, '_suffix'), ammount = 5 )

currentResult -> update_field(currentResult, 'account', concat(currentResult.account, '_suffix'), 'ammount', 5 )
```

The set syntax defaults the name of the lambda variable to "currentResult" and removes the odd looking quotes around the variable names.

5.8.2 flatten_folder_results

```
val outdf = testDataDF.withColumn("together", rer).selectExpr("explode(flatten_folder_results(together)) as expl").selectExpr("expl.result")
```

This sql function behaves the same way as per flatten_rule_results with debugRules working as expected.

5.8.3 resolveWith

Use with care - very experimental

The resolveWith functionality has several issues with Spark compatibility which may lead to code failing when it looks like it should work. Known issues:

1. Using filter then count will stop necessary attributes being produced for resolving, Spark optimises them out as count doesn't need them, however the rules definitely do need some attributes to be useful.
2. You may not select different attributes, remove any, re-order them, or add extra attributes, this is likely to cause failure in show'ing or write'ing
3. Spark is free to optimise other actions than just count, ymmv in which ones work.
4. The 0.1.0 implementation of update_field (based on the Spark impl) does not work in some circumstances (testSimpleProductionRules will fail) - see #36

resolveWith attempts to improve performance of planning for general spark operations by first using a reduced plan against the source dataframe. The resulting Expression will have all functions and attributes resolved and is hidden from further processing by Spark until your rules actually run.

```
val testDataDF = ....

val rer = ruleEngineRunner(sparkSession.sparkContext.broadcast(ruleSuite),
    DataType.fromDDL(DDL), debugMode = debugMode, resolveWith = resolveWith = Some(testDataDF))

val withRules = rer.withColumn("ruleResults", rer)

// ... use the rules
```

You definitely shouldn't use it when using relation or table fields in your expressions e.g. table.field this does not work (verify this by running JoinValidationTest using evalCodeGens instead of evalCodeGensNoResolve). There be dragons. This is known to fail on *all* OSS builds and OSS runtimes (up to and including 3.2.0). 10.2.dbr and 9.1.dbr *actually do work* running the tests in notebooks with resolveWith and relations (the test itself is not built for this however to ensure cross compilation on the OSS base).

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

5.9 QualityCollector

collectRunner is expressly designed for the following rule pattern:

Rule ID	Salience	Rule	Output Expression
1	100	a = '5' and b is not null and c > 100	array(outputrow1, outputrow2, outputrow3)
2	200	a = '5' and c > 100	array(outputrow4)

Where the row (a,b,c) (5, 'value', 150) should have the output:

Result
array(outputrow1, outputrow2, outputrow3, outputrow4)

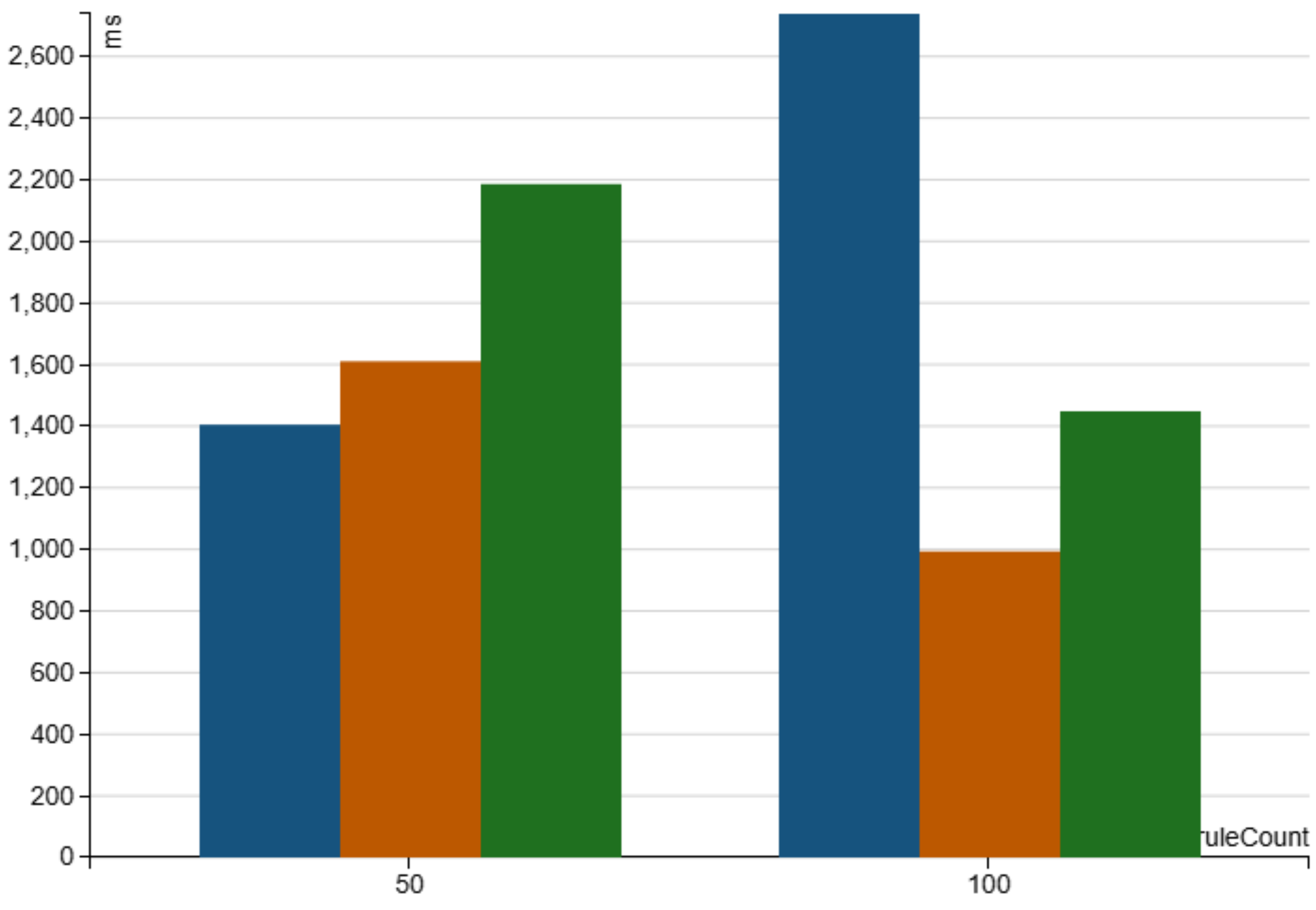
i.e. the Output Expressions of all matching rules should be added and flattened in salience order.

Quality prior to 0.1.4 offered ruleFolder as a general case 'run all the things which match' engine. The above pattern can be represented by:

Rule ID	Salience	Rule	Output Expression
1	100	a = '5' and b is not null and c > 100	set(resultArray = concat(currentResult.resultArray, array(outputrow1, outputrow2, outputrow3)))
2	200	a = '5' and c > 100	set(resultArray = concat(currentResult.resultArray, array(outputrow4)))

This is functionally identical but each rule involves two additional array creations and array copy's.
If this wasn't expensive enough the use of a Spark LambdaFunction disables all subexpression eliminations within those Output Expressions.

The CollectorThroughputBenchmark shows the following indicative results against 1m rows using 50 and 100 rules with Spark 4, the leftmost (blue) is Spark SQL, right (green) is folder and the middle is collectRunner:



The Spark SQL approach is:

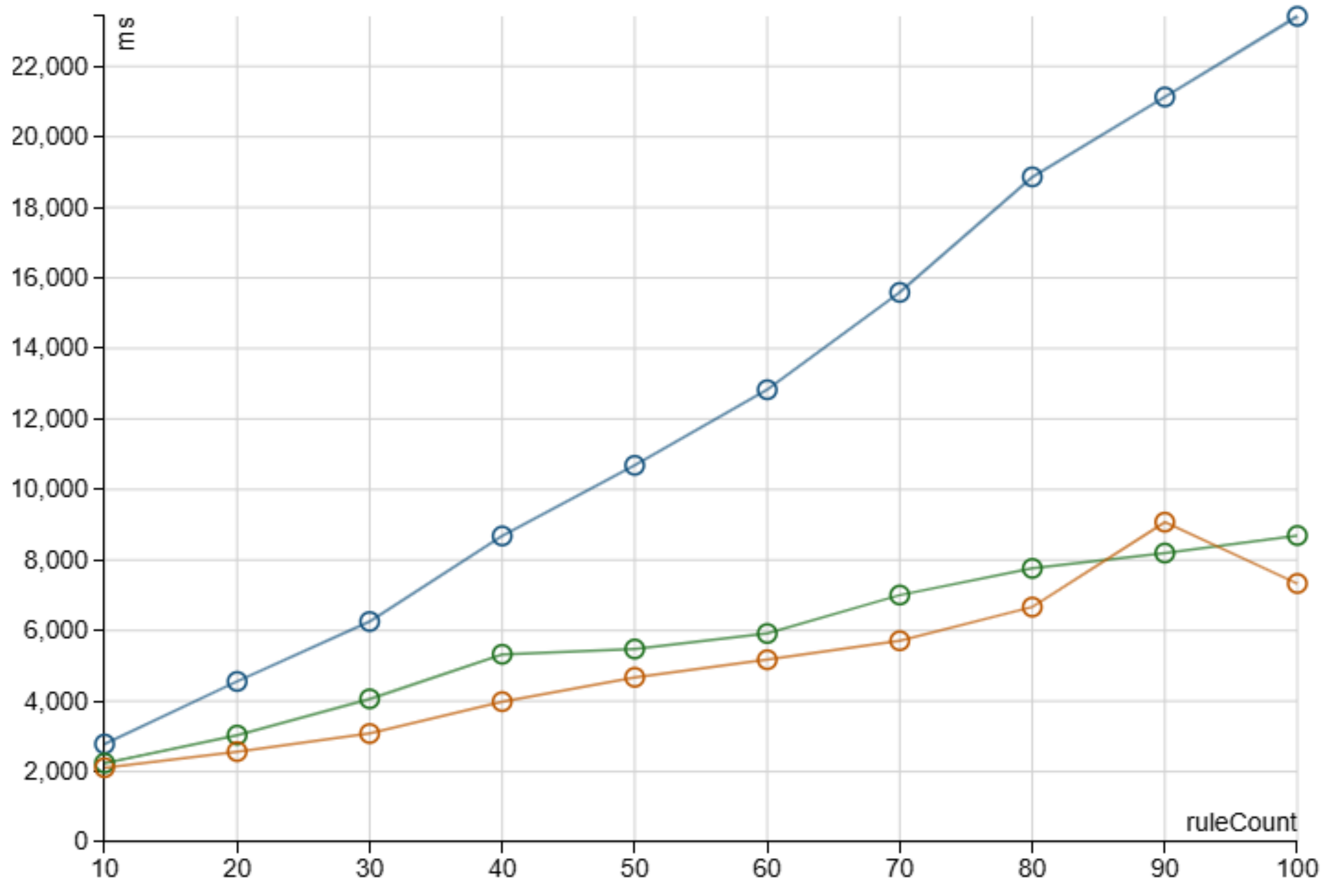
```
flatten(filter(array(if(rule, line1, null), if(rule2, line2, null))..
```

with the filter removing nulls before flattening. Note that this Spark 4 run shows better performance for smaller RuleSuite sizes, where increasing the number of rules causes Quality's runners to take the lead.

Interestingly, the JIT seems to heavily optimise with the more complex 100 rules suite leading to a faster wall clock time for the Quality runs. Less interestingly, the same is true for increases in record counts, when jumping to 10m rows both Quality runners pull ahead with even just 10 rules:

collectRunner leads in these scenarios by efficient array allocations and by default auto flattening nested calls to array and not using filter (this uses a LambdaExpression and cannot be optimised out).

When the rule number falls below 50 Spark is faster, but this only holds on 1m rows of data, at 10m rows the performance of Quality runners pulls ahead for all rule counts e.g.:



with an unfortunate (assumed cpu) interruption on the 90 rule mark the trend is clear (top line is Spark array, middle is folder and bottom is collectRunner).

How can it be faster than normal Spark SQL?

collectRunner filters out for nulls without requiring intermediary array instances, any by direct result array access from the nested expressions.
collectRunner replaces Sparks CreateArray with a version (InPlaceArray) that only allocates one array per partition, Sparks creates a new array per row, it then uses the array directly rather than going through indirect calls with casts etc.

So the equivalent spark of:

```
filter(array(if(a, null, oa), if(b, null, ob)), x -> x IS NOT NULL)
```

which is the same as using array_compact, requires at least three array creations as well as the overhead of the lambda, which, as per folderRunner, cannot take part in sub expression elimination and other optimisation strategies.
Indeed filter currently cannot currently have subexpression elimination applied at all, this also includes the array input.
Of course Quality runners also provide an audit trail missing in the simple Spark SQL approach.

flatten = false, includeNulls = true needs Option

In Spark 3.4 and below not wrapping in Option will cause an NPE. See CollectRunnerTest for example encoding approaches.

5.9.1 defaultProcessor

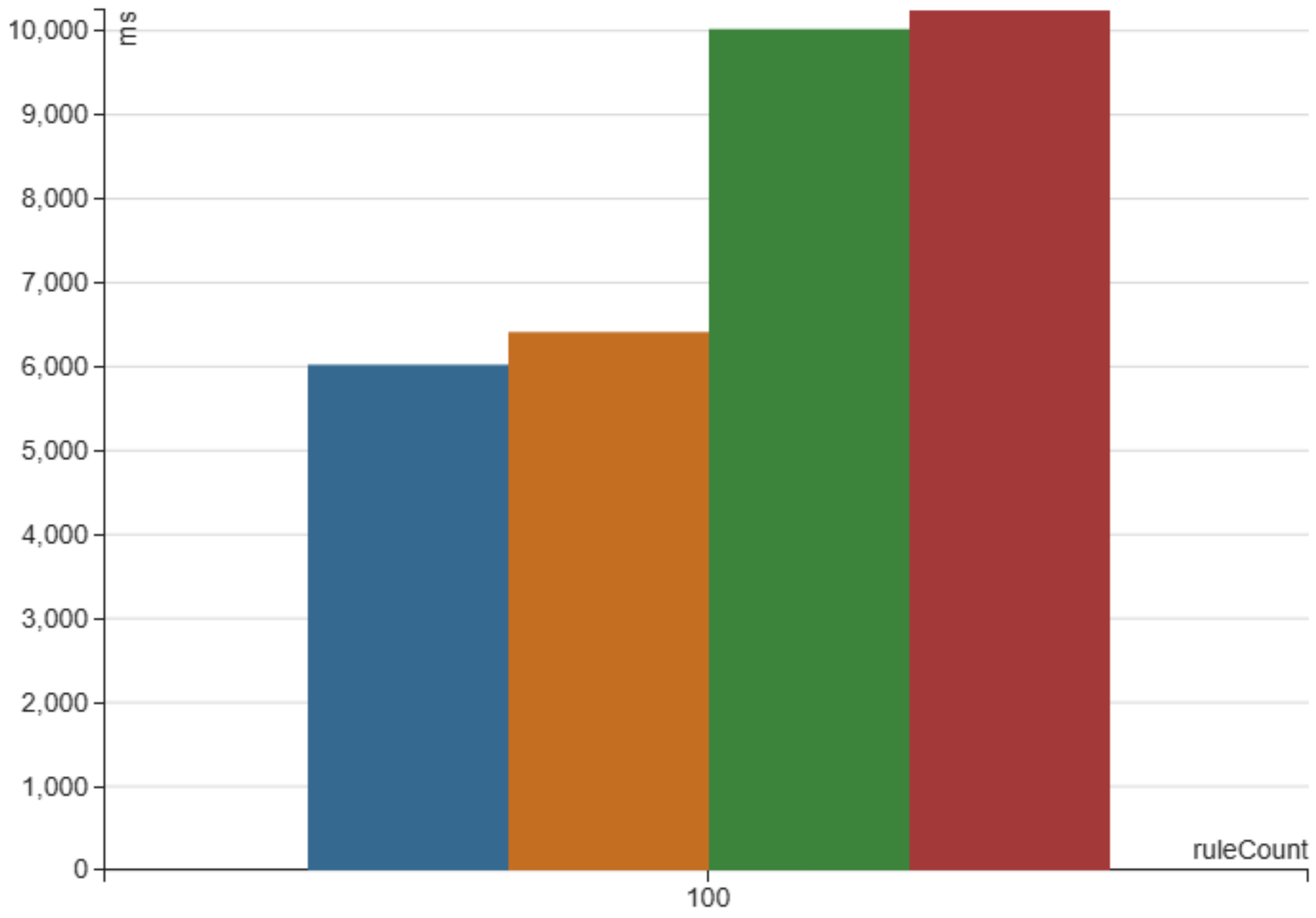
If no trigger rules match the overallStatus for a RuleSuiteResult will be failed() and collectRunner, by default, will return an empty array.

It can be useful however to perform a specific output expression in this case, these can either be directly specified or loaded via with 0.2.0's Connect friendly combine functions or via the serializing integrateRuleSuites functions.

To clearly separate a Passed trigger rule vs. a defaultProcessor result DefaultRule is stored in the RuleSuiteResult.overallResult, although every other RuleSet.overallResult and ruleResult will be Failed.

The alternative from an sql perspective is to use another projection and an 'if' on the resulting array to default, or specify the sql rule twice directly in an if (assuming subexpression elimination will take place).

The benefit of evaluating without a separate projection is clear:



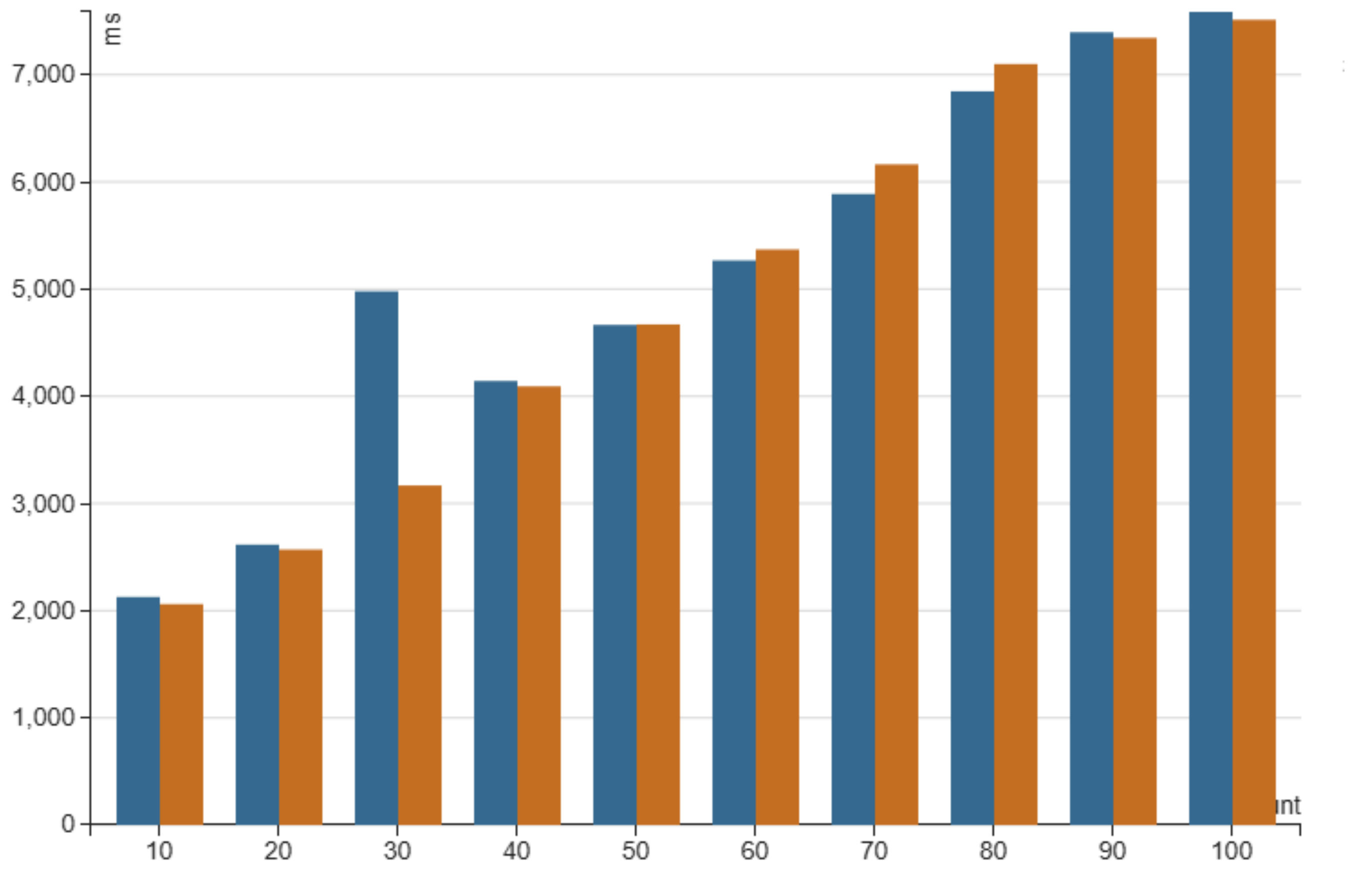
The two left most (blue and orange) are using collect with the defaultProcessor on the left. The two right most are Spark SQL, with the 2nd from the right being a repetition of the SQL rules snippet in the same projection, and the right most being a separate projection. e.g.:

Type	Logic Used	Projection	Mean time taken in ms
collect defaultProcessor	ruleSuite.copy(defaultProcessor = DefaultProcessor(Id(10000,1), OutputExpression("array(...)")	result.result	6022.40
collect default logic in projection	ruleSuite	if(size(result.result) > 0, result.result, array(...))	6412.10
Spark SQL repeated logic	expr(s"if(size($theExpression) > 0, $theExpression, array(...))")	result	10020.00
Spark SQL default logic in projection	expr(s"$theExpression")	if(size(result) > 0, result, array(...))	10242.19

5.9.2 Performance Tweaking Options

InPlaceArray

collectRunner, by default, swaps Spark's CreateArray for InPlaceArray, which only allocates a single array per partition. In general, it performs as well as, if not better, than using CreateArray, but that's not always the case, e.g.:



Here, aside from the measurement blip, the orange bar (right of the bar pairs) are the runs using CreateArray against 10m rows. The difference is performance for using InPlaceArray can be significant, as such it is configurable when calling the ruleSuite and can be overridden by using:

```
com.sparkutils.collect.useInPlaceArray false
```

on your environment or Spark job options.

Loop unrolling

This value can be safely ignored for low numbers of items in the 'array(' Output Expressions, the JIT's optimisation of loops is more than adequate. If, however, there are large numbers of average entry sizes you can experiment with these settings.

With a default of 'false', this applies to InPlaceArray results processing and uses the following parameters:

```
com.sparkutils.collect.unrollOutputArray true
com.sparkutils.collect.unrollOutputArraySize 1
```

The value of 1 forces the compilation to use a for loop and is functionally equivalent to the default unrollOutputArray of false.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

5.10 QualityExpressions

ExpressionRunner applies a RuleSuite over a dataset returning the results of any expression as yaml (json cannot support non string map keys), to return a single real type from all the rules use typedExpressionRunner instead. When used with just aggregate expressions it allows running dataset level checks, run after DQ it also allows statistics on individual rule results.

It is important to note that if you are having multiple runners in the same data pipeline they should each use different RuleSuites, and you should consider .cache'ing the intermediate results.

RuleSuites are built per the normal DQ rules and executed by adding an expressionRunner column:

```
val dqRuleSuite = ...

val aggregateRuleSuite =

val testDataDF = ...

import frameless._
import quality.implicits._

// first add dataQuality, then ExpressionRunner
val processed = taddDataQuality(sparkSession.range(1000).toDF, dqRuleSuite).select(expressionRunner(aggregateRuleSuite)).cache

val res = processed.selectExpr("expressionResults.*").as[GeneralExpressionsResult].head()
assert(res == GeneralExpressionsResult(Id(10, 2), Map(Id(20, 1) -> Map(
  Id(30, 3) -> GeneralExpressionResult("'499500'\n", "BIGINT"),
  Id(31, 3) -> GeneralExpressionResult("'500'\n", "BIGINT")
))))

val gres =
  processed.selectExpr("rule_result(expressionResults, pack_ints(10,2), pack_ints(20,1), pack_ints(31,3)) rr")
    .selectExpr("rr.*").as[GeneralExpressionResult].head

assert(gres == GeneralExpressionResult("'500'\n", "BIGINT"))
```

To retrieve results in the correct type use from_yaml with the correct ddl. As Spark needs an exact type for any expression you can't simply flatten or explode as with the other Quality runner types, each result can have it's own type. As such it's recommended that the expressionRunner result row is cached and extraction is performed with one of the following pattern:

```
import sparkSession.implicits._
val t31_3 = processed.select(from_yaml(rule_result(col("expressionResults"), pack_ints(10,2), pack_ints(20,1), pack_ints(Id(31,3)), 'BIGINT'))).as[Long].head
```

Or parse the GeneralExpressionResult map directly and:

```
val t31_3 = sparkSession.sql(s"from_yaml(${res.ruleSetResults(Id(20,1))(Id(30,3)).ruleResult}, 'BIGINT')").as[Long].head
```

or, finally, and perhaps as a last resort, to use snakeyaml and consume using the resulting java:

```
import org.yaml.snakeyaml.Yaml
val yaml = new Yaml();
val obj = yaml.load[Int](res.ruleSetResults(Id(20,1))(Id(30,3)).ruleResult).toLong;
println(obj);
```

However, as can be seen with the direct use of snakeyaml example the types may not always automatically align and, in this case or Decimal you risk losing precision. Note that the yml spec allows the default implementation as the values have no bit-length for either integer or floats.

To increase precision / accuracy of the yaml types if you are not using from_yaml, you can provide the renderOptions map with useFullScalarType = 'true'. This changes the output considerably:

```
val processed = taddDataQuality(sparkSession.range(1000).toDF, rowrs).select(expressionRunner(rs, renderOptions = Map("useFullScalarType" -> "true")))

val gres =
  processed.selectExpr("rule_result(expressionResults, pack_ints(10,2), pack_ints(20,1), pack_ints(31,3)) rr")
    .selectExpr("rr.*").as[GeneralExpressionResult].head

// NOTE: the extra type information allows snakeyaml to process Long directly without precision loss
assert(gres == GeneralExpressionResult("!!java.lang.Long '500'\n", "BIGINT"))

import org.yaml.snakeyaml.Yaml
val yaml = new Yaml();
val obj = yaml.load[Long](res.ruleSetResults(Id(20,1))(Id(30,3)).ruleResult);
println(obj);
```

NB: Decimal's will be stored with a java.math.BigDecimal type, rather than scala.math.BigDecimal

You can add tags back in if needed

As using useFullScalarType -> true adds yaml type tags on all output scalars it can increase storage costs considerably, as such it's disabled by default.

It can however be retrieved by simply calling from_yaml and to_yaml again with it enabled if the end result should be used outside of Spark.

Don't mix aggregation functions with non-aggregation functions

Spark may complain before running an action, but it's also possible to produce incorrect results.

This is the equivalent of running:

```
select *, sum(id) from table
```

which will not work without group by's.

5.10.1 strip_result_ddl

The resultType string is useful in debugging but may not be for storage, if you wish to trim this information from the results use the strip_result_ddl function.

This turns the result from GeneralExpressionResult into a simple string:

```
val stripped = processed.selectExpr("strip_result_ddl(expressionResults) rr")

val strippedRes = stripped.selectExpr("rr.*").as[GeneralExpressionsResultNoDDL].head()
assert(strippedRes == GeneralExpressionsResultNoDDL(Id(10, 2), Map(Id(20, 1) -> Map(
  Id(30, 3) -> "'499500'\n",
  Id(31, 3) -> "'500'\n")
)))

val strippedGres = {
  import sparkSession.implicits._
  stripped.selectExpr("rule_result(rr, pack_ints(10,2), pack_ints(20,1), pack_ints(31,3))")
    .as[String].head
}

assert(strippedGres == "'500'\n")
```

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

5.11 Validation

Quality provides some validation utilities that can be used as part of your rule design activity to ensure sure you aren't using variables or functions that don't exist, or even possibly having recursive lambda calls.

It comes in two distinct flavours:

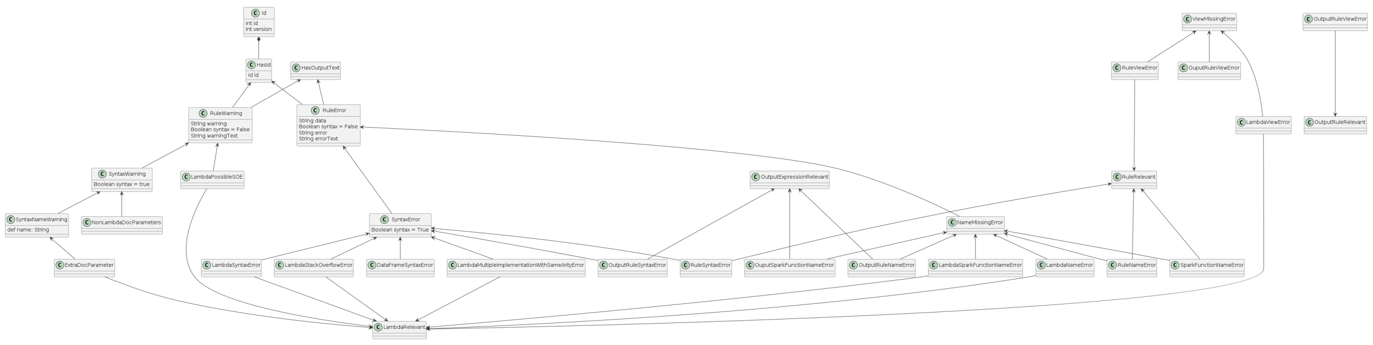
1. Schema Based - The schema representing your dictionary
2. DataFrame Based - Use an actual DataFrame to provide your dictionary

with the option of running the rules against your schema (or DataFrame) via the runnerFunction parameter.

A simpler function for just assessing known Errors against a schema are also provided:

```
def validate(schema: StructType, ruleSuite: RuleSuite): Set[RuleError]
```

The validation result model is as follows:



so the simple version returns any known Errors backed by case classes so you can pattern match as needed or just display as is via the id and errorText functions.

Resolution of function names are run against the functionRegistry, as such you must register any UDF's or database functions *before* calling validate.

5.11.1 What if I want to actually test the ruleSuite runs?

```
def validate(schemaOrFrame: Either[StructType, DataFrame], ruleSuite: RuleSuite, showParams: ShowParams = ShowParams(), runnerFunction: Option[DataFrame =>
Column] = None, qualityName: String = "Quality", recursiveLambdassSOEIsOk: Boolean = false, transformBeforeShow: DataFrame => DataFrame = identity):
(Set[RuleError], Set[RuleWarning], String, RuleSuiteDocs, Map[Id, ExpressionLookup])
```

Given you can either use a ruleRunner or a ruleEngineRunner and set a number of parameters on those Column functions the validate runnerFunction is as simple DataFrame => Column that allows you to tweak the output. In the case of ruleEngineRunner you could use debug mode, try with different DDL output types etc. Use the qualityName parameter if you want to store the output in another column. If you don't provide the runnerFunction the resulting string will be empty.

You don't actually have to provide a DataFrame, instead using just schema will generate an empty dataset to allow Spark to resolve against. Using a DataFrame parameter will allow you to capture the output in the resulting tuples _3 String.

There are a number of overloaded validate arity functions to help solve common cases, they all delegate to the above function, whic also returns the documentation objects for each expression in the RuleSuite via the RuleSuiteDocs object, this provides a base for the documentation of a RuleSuite.

5.11.2 What I want to change the dataframe before I show it?

Using the transformBeforeShow parameter you can enhance, select or filter the DataFrame before showing it.

5.11.3 Why do I get a java.lang.AbstractMethodError when validating?

The validation code also validates the sql documentation, checking documented parameters against lambda parameter names (or indeed that you have any parameters when not a lambda).

You probably have a dependency on the Scala Compiler, due to the scala compiler requiring a different parser combinator library this may occur due to classpath issues.

To remediate please make sure that Quality is higher up on your dependencies than the scala compiler is. If need be manually specify the parser combinator library dependency, making sure to use the same version declared in Qualities pom.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

5.12 Expression Documentation

As Quality is based on sql it can be useful to document it in place, particularly with Lambda and Output expressions, but also applies to rules and trigger rules.

The basic format follows javadocs / scaladocs approach, without *'s on each line, but is possible to define on one line:

```
/** My Description @param name name desc @param othername othername desc @return return val*/
```

This could also be written with newlines including markdown (if the renderer supports it):

```
/**
My Description:

* bullet point
* more points

@param name name desc
@param othername othername:

* more description points

@return return val
*/
```

Param's are optional and will generate a warning if the names don't match in the validate function or if params are used on a non-lambda expression.

The return value is also optional but would apply to all expressions.

Whilst an incorrect parameter name will be flagged and warned against you won't be forced to put a comment for every parameter.

A couple of helpful utility functions:

```
  val (errors, warnings, out, docs, expr) = validate(Left(struct), ruleSuite)

  import com.sparkutils.quality.utils.{RuleSuiteDocs, RelativeWarningsAndErrors}

  val relative = RelativeWarningsAndErrors("../sampleDocsValidation/", errors, warnings)
  val md = RuleSuiteDocs.createMarkdown(docs, ruleSuite, expr, qualityURLGOESHERE+"/sqlfunctions/", Some(relative))

  IOUtils.write(md, new FileOutputStream("./docs/advanced/sampleDocsOutput.md"))

  val emd = RuleSuiteDocs.createErrorAndWarningMarkdown(docs, ruleSuite, relative.copy( relativePath = "../sampleDocsOutput/"))
  IOUtils.write(emd, new FileOutputStream("./docs/advanced/sampleDocsValidation.md"))
```

exist to generate docs of a ruleSuite and validation errors. The validate function returns both of these inputs. You must specify the quality url containing the sqlfunction documentation in order to link, hrefs are not carried across mike links yet.

The sample docs and sample errors/warnings are generated from the DocMarkdownTest.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

5.13 View Loading

As of Spark 3.4 sub queries become a great way to provide lookups and transformation logic in rules. In order to support an easier use of views the following functions have been added in 0.1.0:

```
val (viewConfigs, failed) = loadViewConfigs(loader, config.toDF(), expr("id.id"), expr("id.version"), Id(1,1),
      col("name"),col("token"),col("filter"),col("sql")
    )

val results = loadViews(viewConfigs)
```

loadViewConfigs takes a DataFrameLoader as a parameter allowing Quality to load tables based on your integration logic. There are two flavours, one expecting a table with the following schema:

```
STRUCT< name : STRING, token : STRING nullable, filter : STRING nullable, sql: STRING nullable>
```

and the other tied to a RuleSuite:

```
STRUCT< ruleSuiteId: INT, ruleSuiteVersion: INT, name : STRING, token : STRING nullable, filter : STRING nullable, sql: STRING nullable>
```

both versions return any rows for which token and sql are both null in the failed result and the resulting configuration in viewConfigs.

Where token is present the loader will be called for it and the filter column applied (allowing re-use).

After loading the ViewConfig's the loadViews function can be called, registering all the views via createOrReplaceTempView and returning a set of replaced views, failedToLoadDueToCycles and notLoadedViews, a set of unloaded views. In the event that views refer to other views not present in ViewConfig a MissingViewAnalysisException will be thrown, ViewLoaderAnalysisException for other analysis exceptions, as will parsing exceptions as per normal Spark.

loadViews will attempt to automatically attempt to resolve ViewConfigs that depend on other ViewConfigs, where there is a cycle that is 2x the number of ViewConfigs the call will return with failedToLoadDueToCycles as true.

These calls must be made before running any dq, engine or folder using views.

View names must be quoted if using special characters

A good rule of thumb is minus', dot's etc. that you wouldn't be able to use as a table name in any other sql dialect must be `quoted` in backticks. On Spark versions less than 3.2 any missing views will not contain back ticks, this can lead to situations on earlier Spark versions where views are not loaded and will result in the MissingViewAnalysisException.missingRelationNames also not having backticks returned. Quality will attempt to work around this limitation when resolving dependencies.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

5.14 Processors - Row By Row

Quality Processors allow for Quality rules to be used on a jvm outside of Spark execution. Spark is required for expression resolution and compilation so the pattern of usage is:

```
import com.sparkutils.quality.sparkless.ProcessFunctions._
case class InputData(fields)

val sparkSession = SparkSession.builder().
  config("spark.master", s"local[1]").
  config("spark.ui.enabled", false).getOrCreate()

registerQualityFunctions() // optional

try {
  val ruleSuite = // get rulesuite
  import sparkSession.implicits._
  // thread safe to share
  val processorFactory = dqFactory[InputData](ruleSuite)

  // in other threads an instance is needed
  val threadSpecificProcessor = processorFactory.instance
  try {
    val dqResults: RuleSuiteResult = threadSpecificProcessor(new InputData(...))
  } finally {
    // when your thread is finished doing work close the instance
    threadSpecificProcessor.close()
  }

} finally {
  sparkSession.stop()
}
```

Stateful expressions ruin the fun

Given the comment about "no Spark execution" why is a sparkSession present? The Spark infrastructure is used to compile code, this requires a running spark task or session to obtain configuration and access to the implicits for encoder derivation. **IF** the rules do not include stateful expressions (why would they?) and you use the default compilation, without Spark's higher order functions, this is also possible:

```
import com.sparkutils.quality.sparkless.ProcessFunctions._
case class InputData(fields)

val sparkSession = SparkSession.builder().
  config("spark.master", s"local[1]").
  config("spark.ui.enabled", false).getOrCreate()
val ruleSuite = // get rulesuite
import sparkSession.implicits._
// thread safe to share
val processorFactory = dqFactory[InputData](ruleSuite)

sparkSession.stop()

// in other threads an instance is needed
val threadSpecificProcessor = processorFactory.instance
try {
  threadSpecificProcessor.initialize(partitionId) // Optional, see below Partitions note
  val dqResults: RuleSuiteResult = threadSpecificProcessor(new InputData(...))
} finally {
  // when your thread is finished doing work close the instance
  threadSpecificProcessor.close()
}
```

The above bold IF is ominous, why the caveat? Stateful expressions using compilation are fine, the state handling is moved to the compiled code. If, however, the expressions are "CodegenFallback" and run in interpreted mode then each thread needs its own state. The same is true for using compile = false as a parameter, as such it's recommended to stick with defaults and avoid stateful expressions such as monotonically_incrementing_id, rand or unique_id.

Similarly, Spark's Higher Order Functions such as transform always require re-compilation. Later Spark versions or a custom compilation handlers can remedy this. Using Quality managed user functions is fine as long as they too don't use Spark's Higher Order Functions.

If the rules are free of such stateful expressions then the .instance function is nothing more than a call to a constructor on pre-compiled code.

In short, given stateful expressions can provide different answers for the same inputs it's something to be avoided unless you really need that behaviour.

Thread Safety

In all combinations of ProcessorFactory's the factory itself is thread safe and may be shared, the instances themselves are not and use mutable state to achieve performance.

Partitions / initialize?

Despite all the above commentary on Stateful expressions being awkward to use, if you choose to then you should use the initialize function with a unique integer parameter for each thread.

If you are not using stateful expressions you don't need to call initialize.

5.14.1 Encoders and Input types

The output types of all the runners are well-defined but, like the input types, rely on Spark Encoder's to abstract from the actual types.

For simple beans it's enough to use the Spark Encoders.bean(Class[_]) to derive an encoder or, when using Scala, Frameless encoding derivation.

Java Lists and Maps need special care

Using Java lists or maps with Encoders.bean doesn't work very often, the type information isn't available to the Spark code.

In Spark 3.4 and above you can use AgnosticEncoders instead and specify the types.

Using user Row's with RowEncoders also works, giving you a completely generic ability to push rows through to the processors.

What about something more interesting like an Avro message?

```scala
val testOnAvro = SchemaBuilder.record("testOnAvro")
  .namespace("com.teston")
  .fields()
  .requiredString("product")
  .requiredString("account")
  .requiredInt("subcode")
  .endRecord()
val datumWriter = new GenericDatumWriter[GenericRecord](testOnAvro);

val bos = new ByteArrayOutputStream()
val enc = EncoderFactory.get().binaryEncoder(bos, null)

val avroTestData = testData.map{d =>
  val r = new GenericData.Record(testOnAvro)
  r.put("product", d.product)
  r.put("account", d.account)
  r.put("subcode", d.subcode)
  datumWriter.write(r, enc)
  enc.flush()
  val ba = bos.toByteArray
  bos.reset()
  ba
}

import s.implicits._

val processorFactory = ProcessFunctions.dqFactory[Array[Byte]](rs, inCodegen, extraProjection =
  _.withColumn("vals", org.apache.spark.sql.avro.functions.from_avro(col("value"), testOnAvro.toString)).
    select("vals.*"))
  ...
```

extraProjection allows conversion based on existing Spark conversion functions.

5.14.2 Map Functions

As correlated subqueries cannot be run outside of Spark the Quality Map functions must be used:

```
registerQualityFunctions()

val theMap = Seq((40, true),
  (50, false),
  (60, true)
)
val lookups = mapLookupsFromDFs(Map(
  "subcodes" -> ( () => {
    val df = theMap.toDF("subcode", "isvalid")
    (df, column("subcode"), column("isvalid"))
  } )
))

registerMapLookupsAndFunction(lookups)

val rs = RuleSuite(Id(1,1), Seq(
  RuleSet(Id(50, 1), Seq(
    Rule(Id(100, 1), ExpressionRule("if(product like '%otc%', account = '4201', mapLookup('subcodes', subcode))"))
  ))
))
```

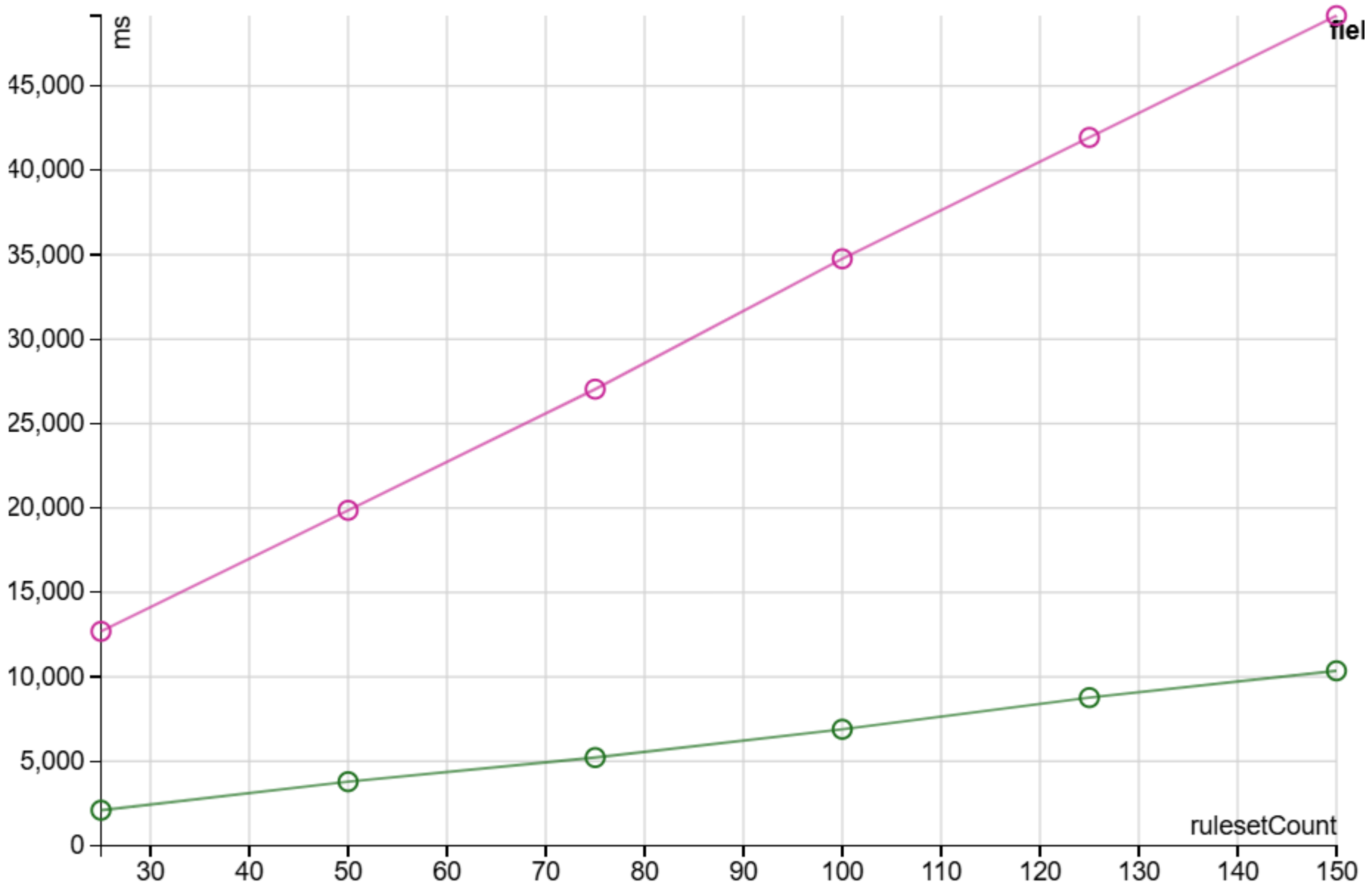
5.14.3 Performance

All the information presented below is captured here in the Processor benchmark.
The run is informative but has some outlier behaviours and should be taken as a guideline only (be warned it takes almost a day to run).
This test evaluates compilation startup time only in the XStartup tests and the time for both startup and running through 100k rows at each fieldCount in a single thread (on a i9-9900K CPU @ 3.60GHz). The inputs for each row are an array of longs, provided by spark's user land Row, with the output a RuleSuiteResult object.

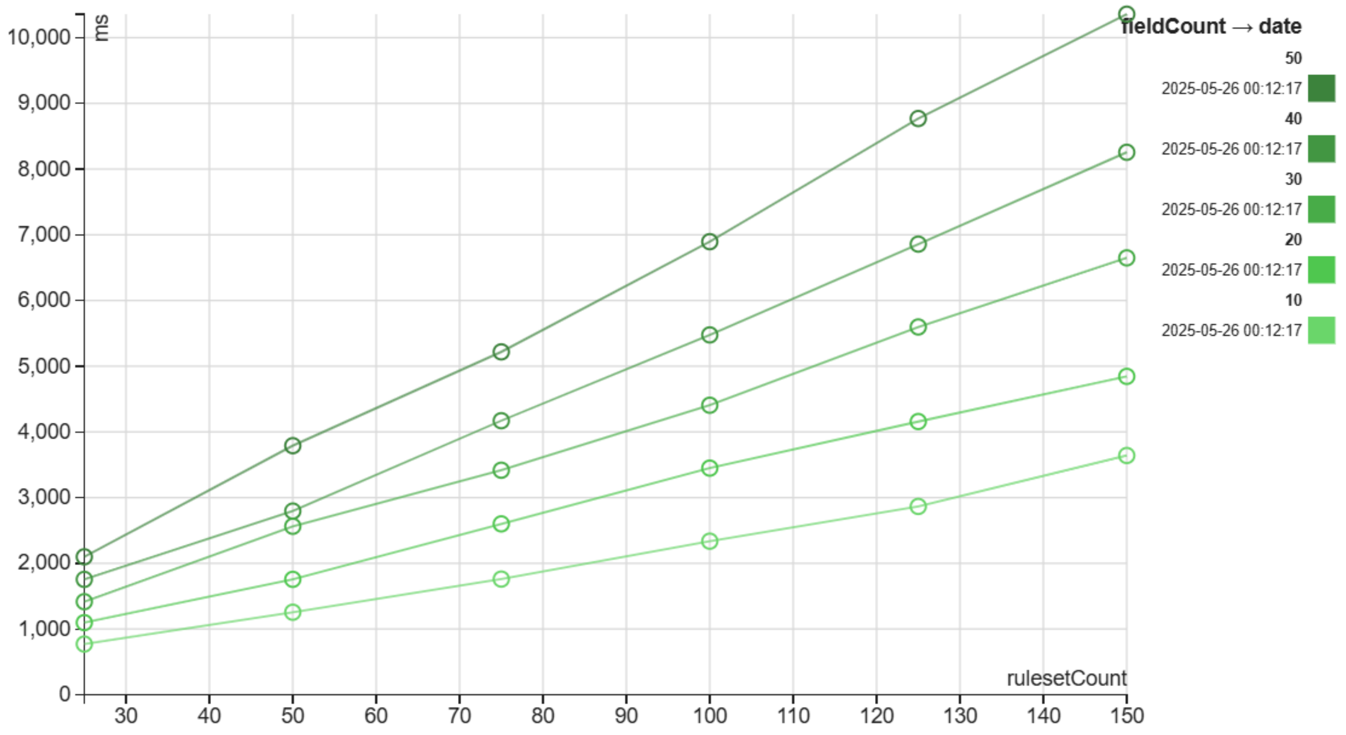
Test combinations to actual rules

rulesetCount	fieldCount	actual number of rules
25	10	30
25	20	55
25	30	80
25	40	105
25	50	130
50	10	60
50	20	110
50	30	160
50	40	210
50	50	260
75	10	90
75	20	165
75	30	240
75	40	315
75	50	390
100	10	120
100	20	220
100	30	320
100	40	420
100	50	520
125	10	150
125	20	275
125	30	400
125	40	525
125	50	650
150	10	180
150	20	330
150	30	480
150	40	630
150	50	780

As noted above the fastest startup time is with `compile = false` as no compilation takes place, this holds true until about the 780 rule mark where compilation catches up with the traversal and new expression tree copying cost. Each subsequent instance call will however pay the same cost again, moreover the actual runtime is by far the worst option:



The lower green line represents the default configuration, which compiles a class and only creates new instances in the general case. The below graph shows the performance trend across multiple rule and field complexities:



In the top right case that's 780 rules total (run across 50 fields) with a cost of about 0.103ms per row (10,300 ms / 100,000 rows) or 0.000132ms/rule/row of simple mod checks.

The performance of the default configuration, leveraging Spark's MutableProjections, is consistently the second best accept for far smaller numbers of rules and field combinations, observable by selecting the 10 fieldCount, every other combination has the default CompiledProjections (GenerateDecoderOpEncoderProjection) in second place by a good enough margin. The first place belongs to the experimental VarCompilation, see the info box below for more details.

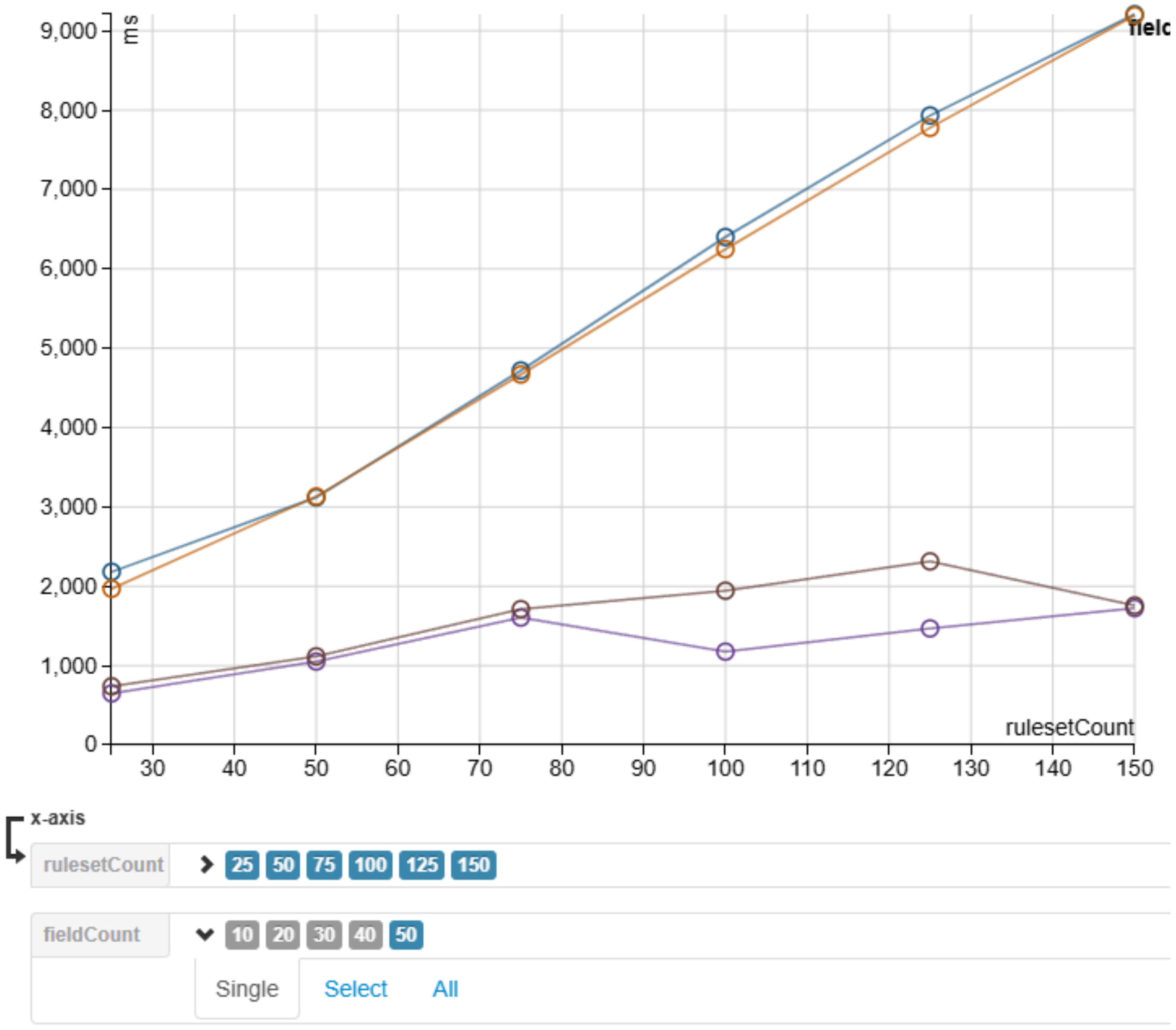
Experimental - VarCompilation

The default of `forceVarCompilation = false` uses a light compilation wrapping around Sparks MutableProjection approach, with the Spark team doing the heavy lifting.

In contrast the `forceVarCompilation = true` option triggers the experimental VarCompilation, mimicing WholeStageCodegen (albeit without severe input size restricitons).
It's additional speed is driven by JIT friendly optimisations and removing all unnecessary work, only encoding from the input data what is needed by the rules. The experimental label is due to the custom code approach, although it can handle thousands of fields actively used in thousands of rules there, and is fully tested it is still custom. This may be changed to the default option in the future.

The majority of cost is the serialisation of the results into the RuleSuiteResult's Scala Maps (via Sparks ArrayBasedMapData.toScalaMap).

If you remove the cost of serialisation, by lazily serializing, things look even faster:



the top two lines are the default and VarCompilation and the bottom two lines their lazy versions, that's 0.0172453 per row and 0.000022109ms per mod check. The dqLazyDetailsFactory function serialises the overall result directly, but only serialises the results on demand, you can choose if you wish to process the details based on the overall result.

Precalculating Passed RuleSuiteResultDetails can be misleading

Although it's possible to use a pre-calculated RuleSuiteResultDetails against all "Passed" results this would not represent any disabled, soft failed or probability results. As such it's not provided by default, if you do have a default RuleSuiteResultDetails you would like to use then you can provide it to the dqLazyDetailsFactory function, using the RuleSuiteResultDetails.ifAllPassed function and the defaultIfPassed parameter.
Using the defaultIfPassed parameter stops actual results from being returned if a row is passed and will only return the default you supplied it.

lazyDQDetailsFactory is also useful if you just want to see if the rules passed and aren't interested in the details.
Similarly the lazyRuleEngineFactory and lazyRuleFolderFactory functions are lazy in their RuleSuiteResult serialisation, which may be appropriate when you are only interested should you not get a result.

You can force expression trees to be copied

quality.forceCopyInProcessorsOverride can be set to override copying of the expression tree, either to force it or stop it from happening. Use this if there are custom expressions that do not behave well in the face of compilation and maintain state but don't use Stateful.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

6. Scala Docs

As of 0.2.0 the main interface is the quality_api docs.

The server side interface can be found here.

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46

7. SQL Functions Documentation

7.1 _

_([ddl type], [nullable]) provides PlaceHolders for lambda functions to allow partial application, use them in place of actual values or expressions to either change arity or allow use in _lambda_.

The default type is Long / Bigint, you will have to provide the types directly when using something else. By default the placeholders are assumed to be nullable (i.e. true), you can use false to state the field should not be null.

Tags:lambda

7.2 _lambda_

lambda(user function) extracts the Spark LambdaFunction from a resolved user function, this must have the correct types expected by the Spark HigherOrderFunction they are parameters for.

This allows using user defined functions and lambdas with in-built Spark HigherOrderFunctions

Tags:lambda

7.3 agg_Expr

agg_Expr([ddl sum type], filter, sum, result) aggregates on rows which match the filter expression using the sum expression to aggregate then processes the results using the result expression.

You can run multiple agg_Expr's in a single pass select, use the first parameter to thread DDL type information through to the sum and result functions.

Tags:aggregate

7.4 as_uuid

as_uuid(lower_long, higher_long) converts two longs into a uuid. Note: this is not functionally equivalent to rng_uuid(longPair(lower, higher)) despite having the same types.

Tags:longs

7.5 big_Bloom

big_Bloom(buildFrom, expectedSize, expectedFPP) creates an aggregated bloom filter using the buildFrom expression.

The blooms are stored on a shared filesystem using a generated uuid, they can scale to high numbers of items whilst keeping the FPP (e.g. millions at 0.01 would imply 99% probability, you may have to cast to double in Spark 3.2).

buildFrom can be driven by digestToLongs or hashWith functions when using multiple fields.

Alternatives:

big_Bloom(buildFrom, expectedSize, expectedFPP, 'bloom_loc') - per above but uses a fixed string bloom_loc instead of a uuid

Tags:bloom

7.6 callFun

callFun(user function lambda variable, param1, param2, ... paramN) used within a lambda function it allows calling a lambda variable that contains a user function.

Used from the top level sql it performs a similar function expecting either a full user function or a partially applied function, typically returned from another lambda user function.

Tags:lambda

7.7 coalesce_If_Attributes_Missing

coalesce_If_Attributes_Missing(expr, replaceWith) substitutes expr with the replaceWith expression when expr has missing attributes in the source dataframe. Your code must call the scala processIfAttributeMissing function before using in validate or ruleEngineRunner/ruleRunner:

```
val missingAttributesAreReplacedRS = processIfAttributeMissing(rs, struct)

val (errors, _) = validate(struct, missingAttributesAreReplacedRS)

// use it missingAttributesAreReplacedRS in your dataframe..
```

NOTE: When using Spark 4 / DBR 17.3 or higher you can use the connect friendly process_if_attribute_missing SQL function instead

Tags:rule

7.8 coalesce_If_Attributes_Missing_Disable

coalesce_If_Attributes_Missing_Disable(expr) substitutes expr with the DisabledRule Integer result (-2) when expr has missing attributes in the source dataframe. Your code must call the scala processIfAttributeMissing function before using in validate or ruleEngineRunner/ruleRunner:

```
val missingAttributesAreReplacedRS = processIfAttributeMissing(rs, struct)

val (errors, _) = validate(struct, missingAttributesAreReplacedRS)

// use it missingAttributesAreReplacedRS in your dataframe..
```

NOTE: When using Spark 4 / DBR 17.3 or higher you can use the connect friendly process_if_attribute_missing SQL function instead

Tags:rule

7.9 collect_runner

collect_runner(ruleSuiteVariable) processes the rule suite stored at ruleSuiteVariable using the DQ runner as if calling collectRunner directly.

The optional ddl type is used to control nullability and the type exactly, defaulting to Spark deriving the type that is collected, the default of flattening flattens any nested arrays returned by the output expressions and the includeNulls, by default, filters out any null values.

ruleSuiteVariables must be registered via the register_rule_suite functions.

Alternatives:

collect_runner(ruleSuiteVariable, ddl) - Spark derives the output expression type from the output expressions, this often does not match nullability expectations, as such specifying the DDL is preferred and more reliable
collect_runner(ruleSuiteVariable, ddl, flatten) - additionally setting flatten to false, returning collections of nested arrays for array type OutputExpressions
collect_runner(ruleSuiteVariable, ddl, flatten, includeNulls) - includeNulls keeps null values, encoding results should use Option with Scala Encoders
collect_runner(ruleSuiteVariable, ddl, flatten, includeNulls, variablesPerFunc, variableFuncGroup) This version provides two additional compilation options, these are typically not needed.
collect_runner(ruleSuiteVariable, ddl, flatten, includeNulls, variablesPerFunc, variableFuncGroup, useInPlaceArray, unrollInPlaceArray, unrollOutputArraySize) This version also allows tweaking further collectRunner specific optimisations, these are typically not needed and use should be carefully evaluated.

Tags:variable, runner, Spark4

7.10 comparable_Maps

comparable_Maps(struct | array | map) converts any maps in the input param into sorted arrays of a key, value struct.

This allows developers to perform sorts, distincts, group bys and union set operations with Maps, currently not supported by Spark sql as of 3.4.

The sorting behaviour uses Sparks existing odering logic but allows for extension during the calls to the registerQualityFunctions via the mapCompare parameter and the defaultMapCompare function.

Tags:map

7.11 default_rule

default_rule() returns the DefaultRule Integer result (-4) for use in filtering and returned by collectRunner defaultProcessor

Tags:rule

7.12 digest_To_Longs

digest_To_Longs('digestImpl', fields*) creates an array of longs based on creating the given MessageDigest impl. A 128-bit impl will generate two longs from it's digest

Tags: Hash

7.13 digest_To_Longs_Struct

digest_To_Longs_Struct('digestImpl', fields*) creates structure of longs with i0 to iN named fields based on creating the given MessageDigest impl.

Tags: Hash

7.14 disabled_rule

disabled_rule() returns the DisabledRule Integer result (-2) for use in filtering and to disable rules (which may not signify a version bump)

Tags:rule

7.15 dq_rule_runner

dq_rule_runner(ruleSuiteVariable) processes the rule suite stored at ruleSuiteVariable using the DQ runner as if calling ruleRunner directly.

All the alternatives correlate to their pre Spark 4 non-sql versions.

ruleSuiteVariables must be registered via the register_rule_suite functions.

Alternatives:

dq_rule_runner(ruleSuiteVariable, variablesPerFunc, variableFuncGroup) - additionally specifies the number of expressions to use per function and the number of functions to call in one group, defaulting to 40 and 20 respectively

Tags:variable, runner, Spark4

7.16 drop_field

drop_field(structure_expr, 'field.subfield'*) removes fields from a structure, but will not remove parent nodes.

This is a wrapped version of 3.4.1's dropField implementation.

Tags:struct

7.17 expression_runner

expression_runner(ruleSuiteVariable) processes the rule suite stored at ruleSuiteVariable using the typed expression runner as if calling expressionRunner directly.

The function always serializes to yaml output and expressions do not need to share the actual real type.

All the alternatives correlate to their pre Spark 4 non-sql versions.

ruleSuiteVariables must be registered via the register_rule_suite functions.

Alternatives:

expression_runner(ruleSuiteVariable, name) - allows naming the column directly
expression_runner(ruleSuiteVariable, name, options) - additionally allows the SnakeYaml output to be configured by the options map
expression_runner(ruleSuiteVariable, name, options, variablesPerFunc, variableFuncGroup) - This version provides two additional compilation options, these are typically not needed.

Tags:variable, runner, Spark4

7.18 failed

failed() returns the Failed Integer result (0) for use in filtering

Tags:rule

7.19 field_Based_ID

field_Based_ID('prefix', 'digestImpl', fields*) creates a variable bit length id by using a given MessageDigest impl over the fields, prefix is used with the _base, _i0 and _iN fields in the resulting structure

Tags:ID, Hash

7.20 flatten_Results

flatten_Results(dataQualityExpr) expands data quality results into a flat array

7.21 flatten_Rule_Results

flatten_Rule_Results(dataQualityExpr) expands data quality results into a structure of flattenedResults, salientRule (the one used to create the output) and the rule result.

salientRule will be null if there was no matching rule

7.22 from_yaml

from_yaml(string, 'ddlType') uses snakeyaml to convert yaml into Spark datatypes

Tags:yaml

7.23 group_results

group_results(array_of_runner_results) processes an array of runner results (excluding expressionRunner), grouping the ruleSuiteResults into a RuleSuiteGroupResults alongside any payload.

This, in combination with rule_suite_from allows nested runners (excluding expressionRunner) to be configured into a RuleSuite and their results grouped into RuleSuiteGroupResults, or indeed run over RuleSuiteGroupResults.

DQ Runner results or simple arrays of RuleSuiteGroupResults do not produce a payload and cannot use the overloaded version. Only arrays of matching structs are accepted, anything else will fail the analysis phase.

Alternatives:

group_results(array_of_runner_results, result processing lambda) - uses a processing lambda for engine payloads, saving a projection e.g.:

```
group_results( array_of_engines_returning_arrays, f -> flatten(f) )
```

using the processing version without an engine result will fail the analysis phase.

Tags:rule

7.24 hash_Field_Based_ID

hash_Field_Based_ID('prefix', 'digestImpl', fields*) creates a variable bit length id by using a given Guava Hasher impl over the fields, prefix is used with the _base, _i0 and _iN fields in the resulting structure

Tags:ID, Hash

7.25 hash_With

hash_With('HASH', fields*) Generates a hash value (array of longs) suitable for using in blooms based on the given Guava hash implementation.

Note based on testing the digestToLongs function for SHA256 and MD5 are faster.

Valid hashes: MURMUR3_32, MURMUR3_128, MD5, SHA-1, SHA-256, SHA-512, ADLER32, CRC32, SIPHASH24. When an invalid HASH name is provided MURMUR3_128 will be chosen.

Open source Spark 3.1.2/3 issues

On Spark 3.1.2/3 open source this may get resolver errors due to a downgrade on guava version - 15.0 is used on Databricks, open source 3.0.3 uses 16.0.1, 3.1.2 drops this to 11 and misses crc32, sipHash24 and adler32.

Tags: Hash

7.26 hash_With_Struct

per hash_With('HASH', fields*) but generates a struct with i0 to ix named longs. This structure is not suitable for blooms

Tags: Hash

7.27 id_base64

id_base64(base, i0, i1, ix) Generates a base64 encoded representation of the id, either the single struct field or the individual parts

Alternatives:

id_base64(id_struct) Uses an id field to generate

Tags:ID

7.28 id_Equal

id_Equal(leftPrefix, rightPrefix) takes two prefixes which will be used to match leftPrefix_base = rightPrefix_base, i0 and i1 fields. It does not currently support more than two i's

Tags:ID

7.29 id_from_base64

id_from_base64(base64) Parses the base64 string with an expected default long size of two i.e. an 160bit ID, any string which is not of the correct size will return null

Alternatives:

id_from_base64(base64f, size) Uses a size, which must be literal, to specify the type

Tags:ID

7.30 id_raw_type

id_raw_type(idstruct) Given a prefixed id returns the fields without their prefix

Tags:ID

7.31 id_size

id_size(base64) Given a base64 from id_base64 returns the number of _i long fields

Tags:ID

7.32 ignored_rule

ignored_rule() returns the IgnoredRule Integer result (-3) for use in filtering and to ignoring rules (e.g. a return value in an active rule)

Tags:rule

7.33 inc

inc() increments the current sum by 1

Alternatives:

inc(x) use an expression of type Long to increment

Tags:aggregate

7.34 long_Pair

long_Pair(lower, higher) creates a structure with these lower and higher longs

Tags:longs

7.35 long_Pair_Equal

long_Pair_Equal(leftPrefix, rightPrefix) takes two prefixes which will be used to match leftPrefix_lower = rightPrefix_lower and leftPrefix_higher = rightPrefix_higher

Tags:longs

7.36 long_Pair_From_UUID

long_Pair_From_UUID(expr) converts a UUID to a structure with lower and higher longs

Tags:longs

7.37 map_Contains

map_Contains('mapid', expr) returns true if there is an item in the map.

On Spark 4 / DBR 17.3 and later this function uses Spark Variables and takes the form:

map_contains('mapid', expr, mapLookupsVar) Where mapLookupsVar is the result of loadMaps

Tags:map, variable, Spark4

7.38 map_Lookup

map_Lookup('mapid', expr) returns either the lookup in map specified by mapid or null.

On Spark 4 / DBR 17.3 and later this function uses Spark Variables and takes the form:

map_lookup('mapid', expr, mapLookupsVar) Where mapLookupsVar is the result of loadMaps

Tags:map, variable, Spark4

7.39 meanF

meanF() simple mean on the results, expecting sum and count type Long

Tags:aggregate

7.40 murmur3_ID

murmur3ID('prefix', fields*) Generates a 160bit id using murmer3 hashing over input fields, prefix is used with the _base, _i0 and _i1 fields in the resulting structure

Tags:ID, Hash

7.41 pack_Ints

pack_Ints(lower, higher) a packaged long from two ints, used within result compression

Tags:ruleid

7.42 passed

passed() returns the Passed Integer for use in filtering: 10000

Tags:rule

7.43 prefixed_To_Long_Pair

prefixed_To_Long_Pair(field, 'prefix') converts a 128bit longpair field with the given prefix into a higher and lower long pair without prefix.

This is suitable for converting provided id's into uuids for example via a further call to rngUUID.

Tags:ID, longs

7.44 print_Code

print_Code([msg], expr) prints the code generated by an expression, the value variable and the isNull variable and forwards eval calls / type etc. to the expression.

The code is printed once per partition on the **executors** std. output. You will have to check each executor to find the used nodes output. To use with unit testing on a single host you may overwrite the writer function in registerQualityFunctions, you should however use a top level object and var to write into (or stream), printCode will not be able to write to std out properly (spark redirects / captures stdout) or non top level objects (due to classloader / function instance issues). Testing on other hosts without using stdout should do so to a shared file location or similar.

It is not compatible with every expression

Aggregate expressions like aggExpr or sum etc. won't generate code so they aren't compatible with printCode.

lambda is also incompatible with printCode both wrapping a user function and the _lambda_ function. Similarly the _() placeholder function cannot be wrapped.

Any function expecting a specific signature like aggExpr or other HigherOrderFunctions like aggregate or filter are unlikely to support wrapped arguments.

Tags:util

7.45 print_Expr

print_Expr([msg], expr) prints the expression tree via toString with an optional msg

The message is printed to the **driver** nodes std. output, often shown in notebooks as well. To use with unit testing you may overwrite the writer function in registerQualityFunctions, you should however use a top level object and var to write into (or stream).

Tags:util

7.46 probability

probability(expr) will translate probability rule results into a double, e.g. 1000 returns 0.01. This is useful for interpreting and filtering on probability based results: 0 -> 10000 non-inclusive

Tags:rule

7.47 probability_In

probability_In(expr, 'bloomid') returns the probability of the expr being in the bloomfilter specified by bloomid.

This function either returns 0.0, where it is definitely not present, or the original FPP where it *may* be present.

You may use digestToLongs or hashWith as appropriate to use multiple columns safely.

Tags:bloom

7.48 process_if_attribute_missing

process_if_attribute_missing(ruleSuiteVariable, ddl, name) processes the rule suite at ruleSuiteVariable for coalesce_If_Attributes_Missing_Disable / coalesce_If_Attributes_Missing and sets the resulting transformed ruleSuite to a variable with the name parameter (this can be the same name as ruleSuiteVariable e.g.:

```
process_if_attribute_missing(myRuleSuite, 'struct<..>', 'myRuleSuite')
```

Calling process_if_attribute_missing inside a ruleSuite is not permitted and will throw exceptions.

ruleSuiteVariables must be registered via the register_rule_suite functions.

Tags:variable, util, Spark4

7.49 provided_ID

provided_ID('prefix', existingLongs) creates an id for an existing array of longs, prefix is used with the _base, _i0 and _iN fields in the resulting structure

Tags:ID

7.50 results_With

results_With(x) process results lambda x (e.g. (sum, count) -> sum) that takes sum from the aggregate, count from the number of rows counted. Defaults both the sumtype and counttype as LongType

Alternatives:

results_With([sum ddl type], x) Use the given ddl type for the sum type e.g. 'MAP<STRING, DOUBLE>'
results_With([sum ddl type], [result ddl type], x) Use the given ddl type for the sum and result types

Tags:aggregate

7.51 return_Sum

return_Sum(sum type ddl) just returns the sum and ignores the count param, expands to resultsWith([sum ddl_type], (sum, count) -> sum)

Tags:aggregate

7.52 reverse_Comparable_Maps

reverses a call to comparableMaps

Tags:map

7.53 rng

rng() Generates a 128bit random id using XO_RO_SHI_RO_128_PP, encoded as a lower and higher long pair

Alternatives:

rng('algorithm') Uses Commons RNG RandomSource to implement the RNG
rng('algorithm', seedL) Uses Commons RNG RandomSource to implement the RNG with a long seed

Tags:longs, RNG

7.54 rng_Bytes

rng_Bytes() Generates a 128bit random id using XO_RO_SHI_RO_128_PP, encoded as a byte array

Alternatives:

rng_Bytes('algorithm') Uses Commons RNG RandomSource to implement the RNG
rng_Bytes('algorithm', seedL) Uses Commons RNG RandomSource to implement the RNG with a long seed
rng_Bytes('algorithm', seedL, byteCount) Uses Commons RNG RandomSource to implement the RNG with a long seed, with a specific byte length integer (e.g. 16 is two longs, 8 is integer)

Tags:RNG

7.55 rng_ID

rng_ID('prefix') Generates a 160bit random id using XO_RO_SHI_RO_128_PP, prefix is used with the _base, _i0 and _i1 fields in the resulting structure

Alternatives:

rng_Id('prefix', 'algorithm') Uses Commons RNG RandomSource to implement the RNG, using other algorithm's may generate more long _iN fields
rng_Id('prefix', 'algorithm', seedL) Uses Commons RNG RandomSource to implement the RNG with a long seed, using other algorithm's may generate more long _iN fields

Tags:ID, RNG

7.56 rng_UUID

rng_UUID(expr) takes either a structure with lower and higher longs or a 128bit binary type and converts to a string uuid - use with, for example, the rng() function.

If a simple conversion from two longs (lower, higher) to a uuid is desired then use as_uuid, rng_uuid applies the same transformations as the Spark uuid to the input higher and lower longs.

Tags:longs

7.57 rule_engine_runner

rule_engine_runner(ruleSuiteVariable, ddl) processes the rule suite stored at ruleSuiteVariable using the DQ runner as if calling ruleEngineRunner directly.

The specified ddl type is used for all output expressions

All the alternatives correlate to their pre Spark 4 non-sql versions.

ruleSuiteVariables must be registered via the register_rule_suite functions.

Alternatives:

rule_engine_runner(ruleSuiteVariable) - Spark derives the output expression type, this often does not match nullability expectations, as such specifying the DDL is preferred and more reliable
rule_engine_runner(ruleSuiteVariable, ddl, debug) - additionally allows entering debug mode and returns each matching rule's output expression results
rule_engine_runner(ruleSuiteVariable, ddl, debug, variablesPerFunc, variableFuncGroup) This version provides two additional compilation options, these are typically not needed.

Tags:variable, runner, Spark4

7.58 rule_folder_runner

rule_folder_runner(ruleSuiteVariable, starter, ddl) processes the rule suite stored at ruleSuiteVariable using the DQ runner as if calling ruleFolderRunner directly.

The starter expression parameter is used to create the initial structure to be folded over, the ddl type is used to control nullability and the type exactly.

All the alternatives correlate to their pre Spark 4 non-sql versions.

ruleSuiteVariables must be registered via the register_rule_suite functions.

Alternatives:

rule_folder_runner(ruleSuiteVariable, starter) - Spark derives the output expression type from starter, this often does not match nullability expectations, as such specifying the DDL is preferred and more reliable
rule_folder_runner(ruleSuiteVariable, starter, ddl, debug) - additionally allows entering debug mode and returns each matching rule's output expression results so you can see changes between folds
rule_folder_runner(ruleSuiteVariable, starter, ddl, debug, variablesPerFunc, variableFuncGroup) This version provides two additional compilation options, these are typically not needed.

Tags:variable, runner, Spark4

7.59 rule_result

rule_result(ruleSuiteResultColumn, packedRuleSuiteId, packedRuleSetId, packedRuleId) uses the packed long id's to retrieve the integer ruleResult (see below for ExpressionRunner) or null if it can't be found.

You can use pack_ints(id, version) to specify each id if you don't already have the packed long version. This is suitable for retrieving individual rule results, for example to aggregate counts of a specific rule result, without having to resort to using filter and map values.

rule_result works with ruleRunner (DQ) results (including details) and ExpressionRunner results. ExpressionRunner results return a tuple of ruleResult and resultDDL, both strings, or if strip_result_ddl is called a string.

Tags:rule

7.60 rule_suite_from

rule_suite_from(ruleSuiteGroupVariable, ruleSuiteId) loads the highest versioned RuleSuite with Id rulesSuiteId from the RuleSuiteGroup stored in the variable.

ruleSuiteGroupVariables must be registered via the register_rule_suite_group functions.

Alternatives:

rule_suite_from(ruleSuiteGroupVariable, ruleSuiteId, ruleSuiteVersion) loads the exact RuleSuite from the RuleSuiteGroup stored in the variable.

Tags:variable, rule, Spark4

7.61 rule_Suite_Result_Details

rule_Suite_Result_Details(dq) strips the overallResult from the dataquality results, suitable for keeping overall result as a top-level field with associated performance improvements

7.62 rule_suite_statistics

The rule_suite_statistics(ruleSuiteResult) aggregate function collects a RuleSuiteGroupStatistics object for a given dataset using the default 0.8 probability. When using a split details and overall pair, recombine them via:

```
rule_suite_statistics(struct(resultDetails.id, overallResult, resultDetails.ruleSetResults))
```

It is not recommended to use with 2.4

The aggregate function is based on Aggregator which, in 2.4, is not possible to apply on specific columns.

Quality provides a backport of Spark 3 functionality to enable this and, in addition to the last Quality 2.4 release, this is not an intended Spark 2.4 pattern, although it works in local testing it has not been tested against clusters.

Tags:rule

7.63 small_Bloom

small_Bloom(buildFrom, expectedSize, expectedFPP) creates a simply bytearray bloom filter using the expected size and fpp - 0.01 is 99%, you may have to cast to double in Spark 3.2. buildFrom can be driven by digestToLongs or hashWith functions when using multiple fields.

Tags:bloom

7.64 soft_Fail

soft_Fail(ruleexpr) will treat any rule failure (e.g. failed()) as returning softFailed()

Tags:rule

7.65 soft_failed

soft_failed() returns the SoftFailed Integer result (-1) for use in filtering

Tags:rule

7.66 strip_result_ddl

strip_result_ddl(expressionsResult) removes the resultDDL field from expressionsRunner results, leaving only the string result itself for more compact storage

Tags:rule

7.67 sum_With

sum_With(x) adds expression x for each row processed in an aggExpr with a default of LongType

Alternatives:

sum_With([ddl type], x) Use the given ddl type e.g. 'MAP<STRING, DOUBLE>'

Tags:aggregate

7.68 to_yaml

to_yaml(expression, [options map]) uses snakeyaml to convert Spark datatypes into yaml.

Passing null into the function returns a null yaml (newline is appended):

```
null

```

All null values will be treated in this fashion. The string "null" will be represented as (again new line is present):

```
'null'
```

The optional "options map" parameter currently supports the following output options:

- useFullScalarType, defaults to false. Instead of using the default yaml tags uses the full classnames for scalars, reducing risk of precision loss if the yaml is to be used outside of the from_yaml function.

sample usage:

```
val df = sparkSession.sql("select array(1,2,3,4,5) og")
    .selectExpr("*", "to_yaml(og, map('useFullScalarType', 'true')) y")
    .selectExpr("*", "from_yaml(y, 'array<int>') f")
    .filter("f == og")
```

snakeyaml is provided scope

Databricks runtimes provide sparkyaml, so whilst Quality builds against the correct versions for Databricks it can onyl use provided scope.

snakeyaml is 1.24 on DBRs below 13.1, but not present on OSS, so you may need to add the dependency yourself, tested compatible versions are 1.24 and 1.33.

Tags:yaml

7.69 typed_expression_runner

typed_expression_runner(ruleSuiteVariable, ddl) processes the rule suite stored at ruleSuiteVariable using the typed expression runner as if calling typedExpressionRunner directly.

The ddl parameter specifies the output expressions type, all expressions must share this type.

All the alternatives correlate to their pre Spark 4 non-sql versions.

ruleSuiteVariables must be registered via the register_rule_suite functions.

Alternatives:

typed_expression_runner(ruleSuiteVariable, ddl, name) - allows naming the column directly
typed_expression_runner(ruleSuiteVariable, ddl, name, variablesPerFunc, variableFuncGroup) - This version provides two additional compilation options, these are typically not needed.

Tags:variable, runner, Spark4

7.70 unify_result

unify_result(rule_engine_results) converts rule engine results into collector / folder results (dropping salient rule), also works with group_results.

It does not process debug, expression or DQ results and will fail these in the analysis phase.

Tags:rule

7.71 unique_ID

uniqueID('prefix') Generates a 160bit guaranteed unique id (requires MAC address uniqueness) with contiguous higher values within a partition and overflow with timestamp ms., prefix is used with the _base, _i0 and _i1 fields in the resulting structure

Tags:ID

7.72 unpack

unpack(expr) takes a packed rule long and unpacks it to a .id and .version structure

Tags:ruleid

7.73 unpack_Id_Triple

unpack_Id_Triple(expr) takes a packed rule triple of longs (ruleSuiteId, ruleSetId and ruleId) and unpacks it to (ruleSuiteId, ruleSuiteVersion, ruleSetId, ruleSetVersion, ruleId, ruleVersion)

Tags:ruleid

7.74 update_field

update_field(structure_expr, 'field.subfield', replaceWith, 'fieldN', replaceWithN) processes structures allowing you to replace sub items (think lens in functional programming) using the structure fields path name.

This is a wrapped version of 3.4.1's withField implementation.

Tags:struct

7.75 za_Field_Based_ID

za_Field_Based_ID('prefix', 'digestImpl', fields*) creates a 64bit id (96bit including header) by using a given Zero Allocation impl over the fields, prefix is used with the _base and _i0 fields in the resulting structure.

Prefer using the zaLongsFieldBasedID for less collisions

Tags:ID, Hash

7.76 za_Hash_Longs_With

za_Hash_Longs_With('HASH', fields*) generates a multi length long array but with a zero allocation implementation. This structure is suitable for blooms, the default XXH3 algorithm is the 128bit version of that used by the internal bigBloom implementation.

Available HASH functions are MURMUR3_128, XXH3

Tags: Hash

7.77 za_Hash_Longs_With_Struct

similar to za_Hash_Longs_With('HASH', fields*) but generates an ID relevant multi length long struct, which is not suitable for blooms

Tags: Hash

7.78 za_Hash_With

za_Hash_With('HASH', fields*) generates a single length long array always with 64 bits but with a zero allocation implementation. This structure is suitable for blooms, the default XX algorithm is used by the internal bigBloom implementation.

Available HASH functions are MURMUR3_64, CITY_1_1, FARMNA, FARMOU, METRO, WY_V3, XX

Tags: Hash

7.79 za_Hash_With_Struct

similar to za_Hash_With('HASH', fields*) but generates an ID relevant multi length long struct (of one long), which is not suitable for blooms.

Prefer zaHashLongsWithStruct for reduced collisions with either the MURMUR3_128 or XXH3 versions of hashes

Tags: Hash

7.80 za_Longs_Field_Based_ID

za_Longs_Field_Based_ID('prefix', 'digestImpl', fields*) creates a variable length id by using a given Zero Allocation impl over the fields, prefix is used with the _base, _i0 and _iN fields in the resulting structure. Murmur3_128 is faster than on the Guava implementation.

Tags:ID, Hash

Last update: March 8, 2026 14:46:46

Created: March 8, 2026 14:46:46