

Quality

None

Chris Twiner

Copyright @ 2022 - UBS AG

Table of contents

1. Quality - 0.0.2-SNAPSHOT	5
1.1 Run complex data quality rules using simple SQL in a batch or streaming Spark application at scale.	5
1.2 Enhanced Spark Functionality	5
2. Getting Started	6
2.1 Building and Setting Up	6
2.2 Defining & Running your first RuleSuite	9
2.3 Those are some Quality flavours	10
2.4 Key SQL Functions to use in your Rules	13
2.5 Reading & Writing RuleSuites	14
2.6 Running Quality on Databricks	18
3. About	20
3.1 History	20
3.2 Performance Choices	22
3.3 Changelog	29
4. Model	30
4.1 Rule Model	30
4.2 Storage Model	32
4.3 Meta Rulesets?	33
5. Advanced Usage	34
5.1 Bloom Filters	34
5.2 Map Functions	36
5.3 Aggregation Functions	37
5.4 User Defined Functions	39
5.5 PRNG Functions	44
5.6 Row ID Functions	45
5.7 QualityRules	47
5.8 QualityFolder	53
5.9 Validation	55
5.10 Expression Documentation	57
6. SQL Functions Documentation	58
6.1 _	58
6.2 _lambda_	58
6.3 aggExpr	58
6.4 as_uuid	58
6.5 bigBloom	58

6.6	callFun	58
6.7	coalesceIfAttributesMissing	58
6.8	coalesceIfAttributesMissingDisable	59
6.9	comparableMaps	59
6.10	digestToLongs	59
6.11	digestToLongsStruct	59
6.12	disabledRule	59
6.13	failed	59
6.14	fieldBasedID	59
6.15	flattenResults	59
6.16	flattenRuleResults	60
6.17	hashFieldBasedID	60
6.18	hashWith	60
6.19	hashWithStruct	60
6.20	idEqual	60
6.21	inc	60
6.22	longPair	60
6.23	longPairEqual	60
6.24	longPairFromUUID	61
6.25	mapContains	61
6.26	mapLookup	61
6.27	meanF	61
6.28	murmur3ID	61
6.29	packInts	61
6.30	passed	61
6.31	prefixedToLongPair	61
6.32	printCode	61
6.33	printExpr	62
6.34	probability	62
6.35	probabilityIn	62
6.36	providedID	62
6.37	resultsWith	62
6.38	returnSum	62
6.39	reverseComparableMaps	62
6.40	rng	62
6.41	rngBytes	63
6.42	rngID	63
6.43	rngUUID	63

6.44	ruleSuiteResultDetails	63
6.45	saferLongPair	63
6.46	smallBloom	63
6.47	softFail	63
6.48	softFailed	63
6.49	sumWith	64
6.50	uniqueID	64
6.51	unpack	64
6.52	unpackIdTriple	64
6.53	updateField	64
6.54	zaFieldBasedID	64
6.55	zaHashLongsWith	64
6.56	zaHashLongsWithStruct	64
6.57	zaHashWith	64
6.58	zaHashWithStruct	65
6.59	zaLongsFieldBasedID	65



1. Quality - 0.0.2-SNAPSHOT

Average			
Statement	86.64 %	Branch	76.76 %

1.1 Run complex data quality 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.

-  - [comparableMaps](#) - allow unions or sorting with map columns without json serialising and parsing overhead
-  - [set syntax](#) - simplified syntax for updating and defaulting

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

1.2 Enhanced Spark Functionality

Lookup Functions are distributed across the Spark cluster and held in memory, as such no shuffling is required where the shuffling introduced by joins may be too expensive:

- 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)
- [Map lookup](#) expressions for exact lookups and contains tests, using broadcast variables under the hood they are a great fit for small reference data sets
- [Lambda Functions](#) - user provided re-usable sql functions over late binded columns
- Fast PRNG's exposing [RandomSource](#) allowing plugable and stable generation across the cluster
- [Aggregate functions](#) over Maps expandable with simple SQL Lambdas
- [Row ID](#) expressions including guaranteed unique row IDs (based on MAC address guarantees)

Plus a collection of handy [functions](#) to integrate it all.

Last update: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

2. Getting Started

2.1 Building and Setting Up

2.1.1 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](#).

2.1.2 Running the tests

As with any local Spark development, in order to run the tests you must have the vcaredist 2010 and winutils packages installed, for Spark 2.4.6 and 3.0 it can be downloaded from [here](#).

If you are using 3.1.3 or 3.2 download both the dll and exe from [here](#) and ensure that not only is the HADOOP_HOME defined but that the bin directory within it is on the PATH, you may need to restart IntelliJ.

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.

2.1.3 Build tool dependencies

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

```
quality_RUNTIME_SPARKCOMPATVERSION_SCALACOMPATVERSION-VERSION.jar
```

e.g.

```
quality_3.3.0.oss_3.3_2.12-0.7.0-SNAPSHOT.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
2.4.6	2.4		2.11
3.0.3	3.0		2.12
3.1.3	3.1		2.12
3.1.3	3.1	9.1.dbr_	2.12
3.2.0	3.2		2.12
3.2.1	3.2	3.2.1.oss_	2.12
3.2.1	3.2	10.4.dbr_	2.12
3.3.0	3.3	3.3.0.oss_	2.12
3.3.0	3.3	11.3.dbr_	2.12
3.3.0	3.3	12.2.dbr_	2.12
3.4.0	3.4		2.12

2.4 support is deprecated and will be removed in a future version. 3.1.2 support is replaced by 3.1.3 due to interpreted encoder issues.

Databricks 12.2 is experimental - pending Frameless 3.4 support

12.2 LTS is a mix of 3.3.0 and 3.4.0, as such until Frameless supports 3.4 [see here](#). This *should* not affect the sql function extensions.

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>0.7.0-SNAPSHOT</qualityVersion>
  <qualityTestPrefix>3.2.1.oss_</qualityTestPrefix>
  <qualityDatabricksPrefix>10.4.dbr_</qualityDatabricksPrefix>
  <sparkShortVersion>3.2</sparkShortVersion>
  <scalaCompatVersion>2.12</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
3.2.1	3.2	3.2.1.oss_	10.4.dbr_	2.12
3.3.0	3.3	3.3.0.oss_	11.0.dbr_	2.12

2.1.4 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.

Spark 2.4 runtimes are not supported

2.4 is not supported as Spark doesn't provide for SQL extensions in this version.

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.

Configuring on Databricks 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.

2.1.5 2.4 Support requires 2.4.6 or Janino 3.0.16

Due to [Janino #90](#) using 2.4.5 directly will bring in 3.0.9 janino which can cause VerifyErrors, use 2.4.6 if you can't use a 3.x Spark.

Last update: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

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.

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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

2.3 Those are some Quality flavours

Quality has three main flavours with sprinklings of other Quality ingredients like the [sql function suite](#).

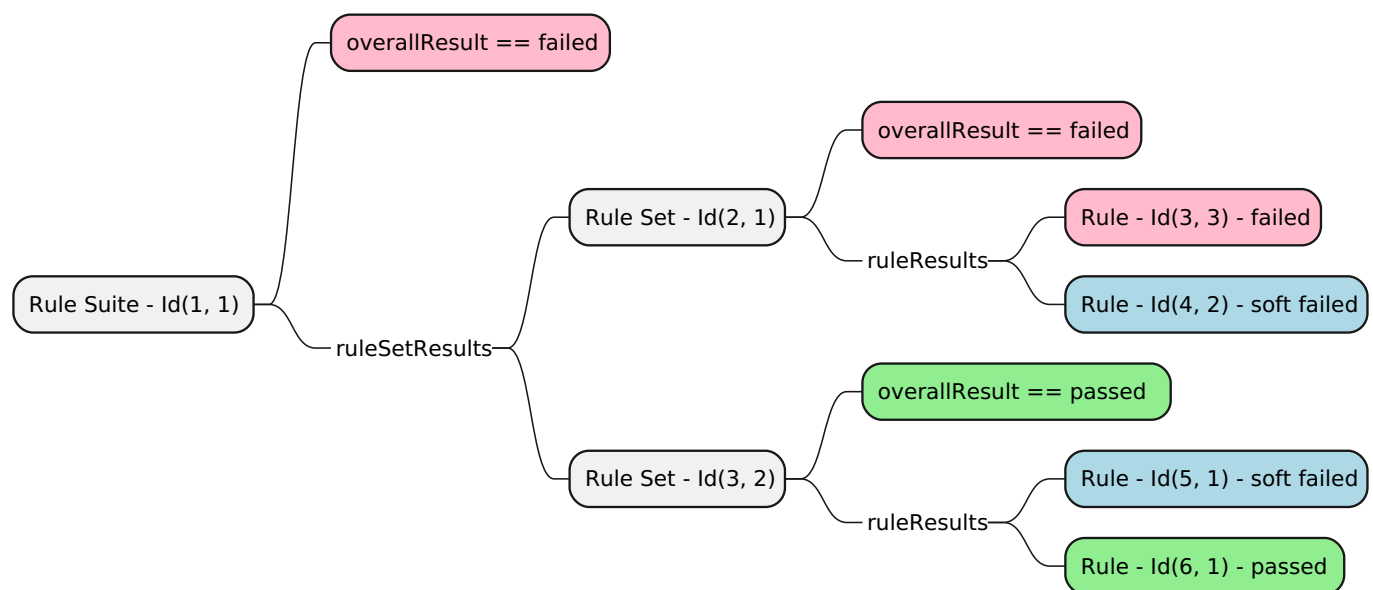
These flavours are provided by three "runners" which add a Column to a Spark Dataset/Dataframe.

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:



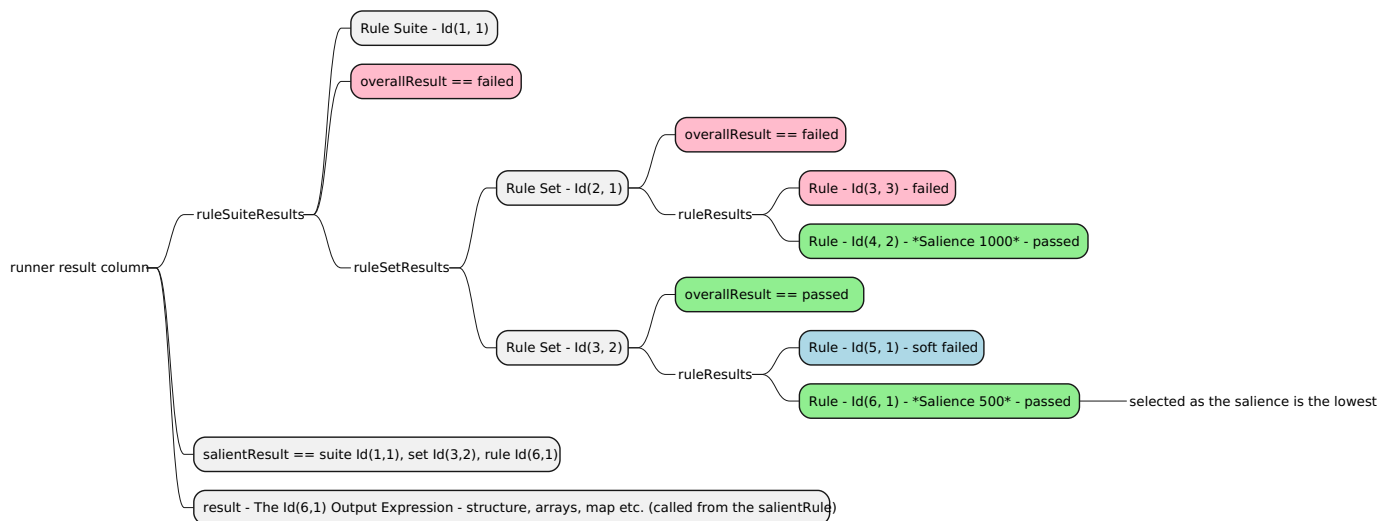
2.3.2 QualityRules - ruleEngineRunner

[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:



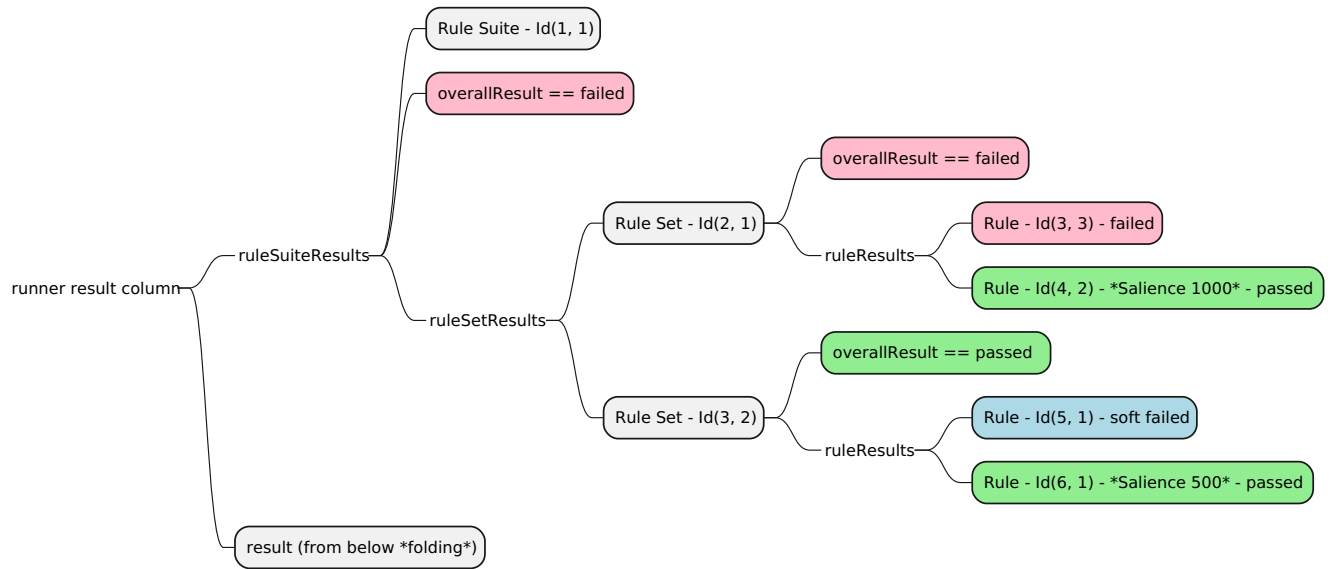
2.3.3 QualityFolder - ruleFolderRunner

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

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

What is stored:



Last update: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

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
- `softFailed()` - the value representing a failed rule which doesn't break the bank
- `disabledRule()` - 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
- `packInts(lower, higher)` - returns a Long with both the lower and higher int's packed in, used for id matching
- `softFail(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.
- `ruleSuiteResultDetails(ruleSuiteResult)` - separates the `RuleSuiteResult.overallResult` from the rest of the structure should it be needed typically this is done via the `addOverallResultsAndDetailsF`

Last update: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

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 be 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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

2.6 Running Quality on Databricks

The aim is to have explicit support for LTS', other interim versions may be supported as needed.

2.6.1 Running 3.1 builds on Databricks Runtime 9.1 LTS

Use the 9.1.dbr build / profile, the artefact name will also end with `_9.1.dbr`. OSS 3.1 do not need to worry about this and should not use this profile.

Databricks has back-ported `TreePattern` including the final `nodePatterns` in `HigherOrderFunction` and 3.2's `Conf` class. As such very old versions of non-opensource Quality ($\leq 0.5.0$) will fail with `AbstractMethodError`'s when `lambda`'s are used as 9.1 as the OSS binary version of `HigherOrderFunction` does not have `nodePattern`. Similarly the `quality_testshade` jar must use the 9.1.dbr version due to `Conf` changes.

The 9.1.dbr build class files are built on the fake `TreePattern` and `HigherOrderFunction` present in the 9.1.dbr-scala source directory, they are however removed in the jar.

`ResolveTableValuedFunctions` and `ResolveCreateNamedStruct` are removed from `resolveWith` as they are binary incompatible with OSS. This does not seem to effect building `namedstructs` using `resolveWith`.

2.6.2 Running 3.2.1 builds on Databricks Runtime 10.4

Use the 10.4.dbr build / profile, the artefact name will also end with `_10.4.dbr`.

DBR 10.4 backports canonicalisation changes which allow Quality and any other code using `explode` and `arrays` to functionally run. Performance is still known to be affected. These fixes are not present in the 3.2.1 OSS release, although performance improvements may be back-ported.

`ResolveTables`, `ResolveAlterTableCommands` and `ResolveHigherOrderFunctions` are removed from `resolveWith` as they are binary incompatible with OSS.

Only 10.4 LTS is supported

10.2 version support was removed in 0.0.1

2.6.3 Running 3.3.0 builds on Databricks Runtime 11.3 LTS

Use the 11.3.dbr build / profile, the artefact name will also end with `_11.3.dbr`. Due to a backport of [SPARK-39316](#) only 11.3 LTS is supported (although likely 11.2 will also run), this changed the result type of `Add` causing incorrect aggregation precision via `aggExpr` (`Sum` and `Average` stopped using `Add` for this reason).

2.6.4 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. As of the 14th March 23 the tests run against 12.2.

2.6.5 Testing out Quality via Notebooks

You can use the appropriate runtime `quality_testshade` artefact jar (e.g. [DBR 11.3](#)) 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.quality.tests.TestSuite
import com.sparkutils.qualityTests.SparkTestUtils
```

```
SparkTestUtils.setPath("path_where_test_files_should_be_generated")
TestSuite.runTests
```

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

In Databricks notebooks you can set the path up via:

```
val fileLoc = "/dbfs/databricks/quality_test"
SparkTestUtils.setPath(fileLoc)
```

Ideally at the end of your runs you'll see - after 10 minutes or so and some stdout - for example on DBR 11.3 a run provides:

```
Time: 682.626

OK (228 tests)

Finished. Result: Failures: 0. Ignored: 0. Tests run: 228. Time: 682626ms.
import com.sparkutils.quality.tests.TestSuite
import com.sparkutils.qualityTests.SparkTestUtils
fileLoc: String = /dbfs/databricks/quality_test
```

NB: *due to no 3.4 frameless yet being available as of 14th March 23 the test counts are now expected to be 206 passed instead of 214*

Last update: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

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 performant and extensible row-level SQL based rule engine with low storage costs and a high degree of optimisation 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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

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.

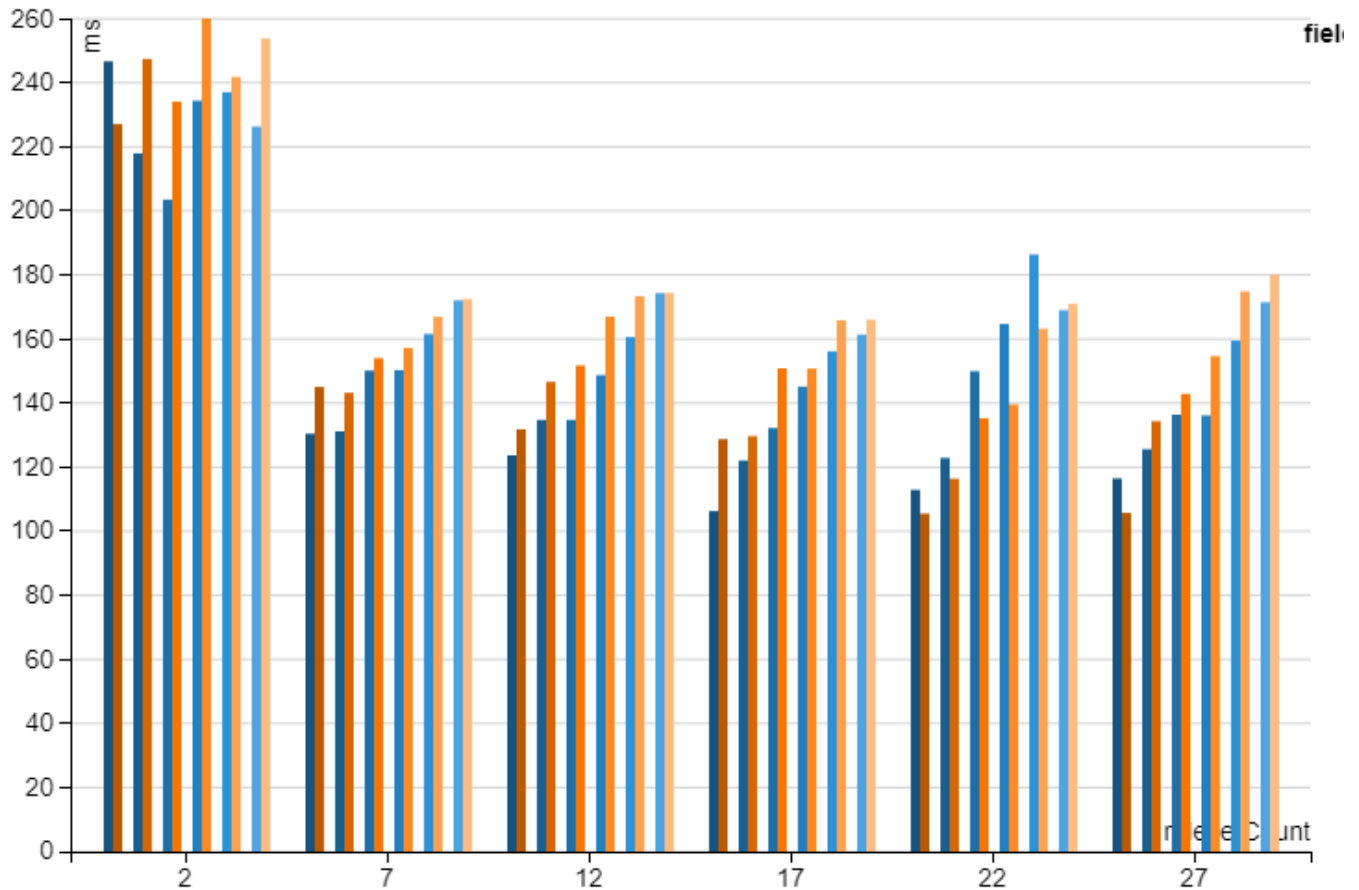
Overall storage winner is nested columns, it has lower storage costs, is as fast as json to serialize (via an Expression) and faster to query with predicate push down support for faster filtering. Details of the analysis are below.

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): $(\$c \% 2) = 0$



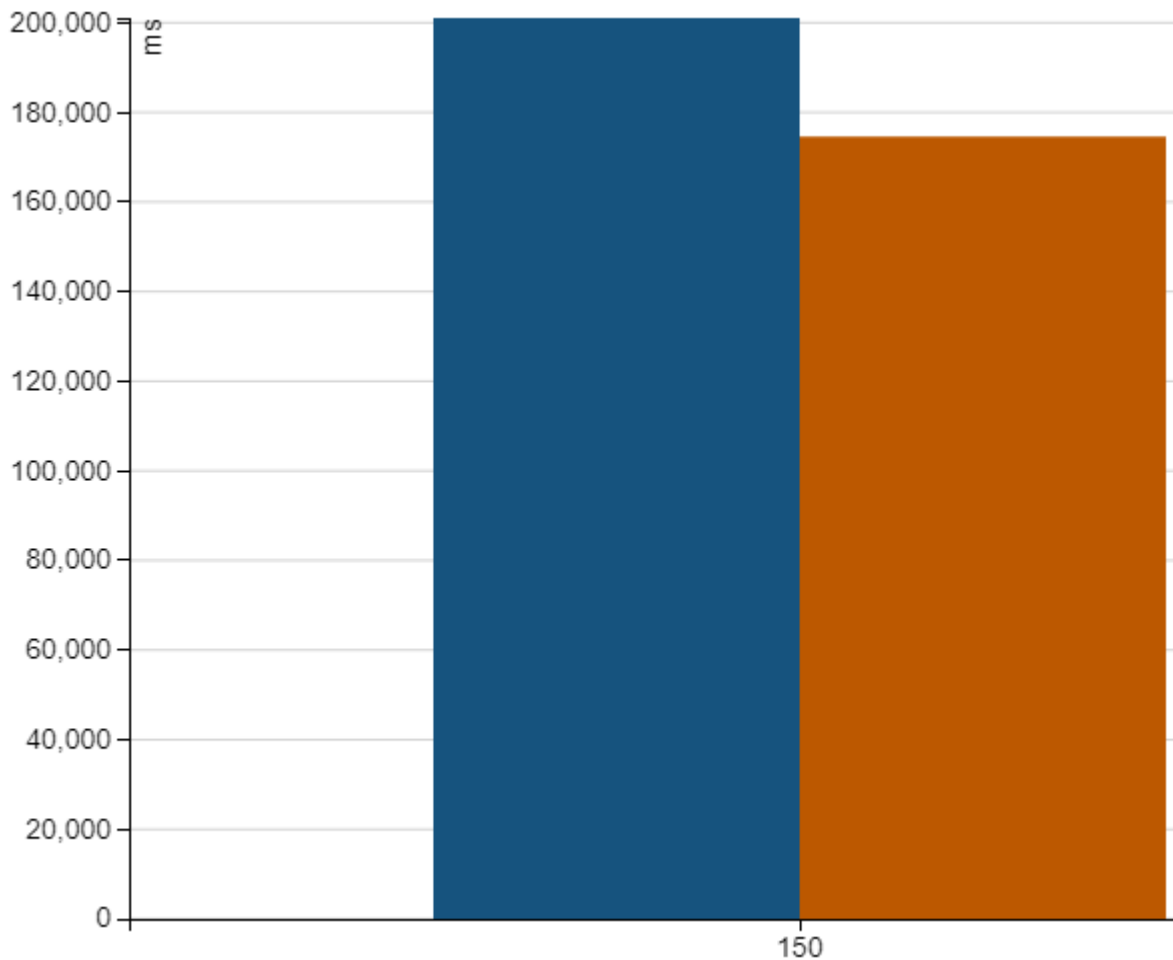
This measurement against 1k rows shows for the last column 230ms for 27 rules each with 27 columns applied, i.e. 0.23 ms per row for 84 rules total (albeit simple rules) on a single 4 core machine (24G heap). Orange representing the default compiled evaluations.

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.

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.

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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

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

Note

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

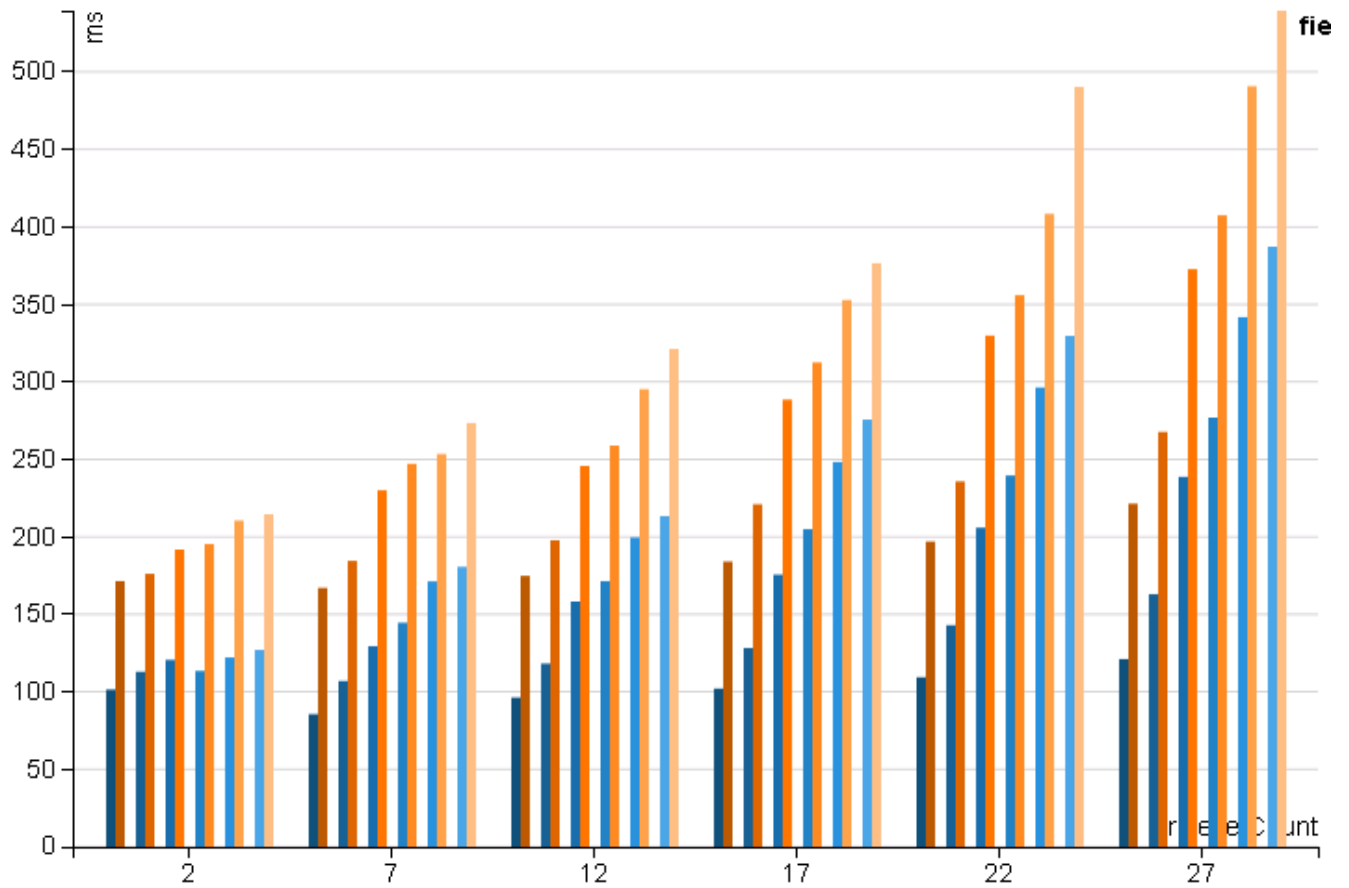
UDF Created Structures

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 loses its 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 separating 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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

3.3 Changelog

0.0.2 28th March, 2023

- #9 - Add AsymmetricFilterExpressions with AsUUID 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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

4. Model

4.1 Rule Model

4.1.1 Rules

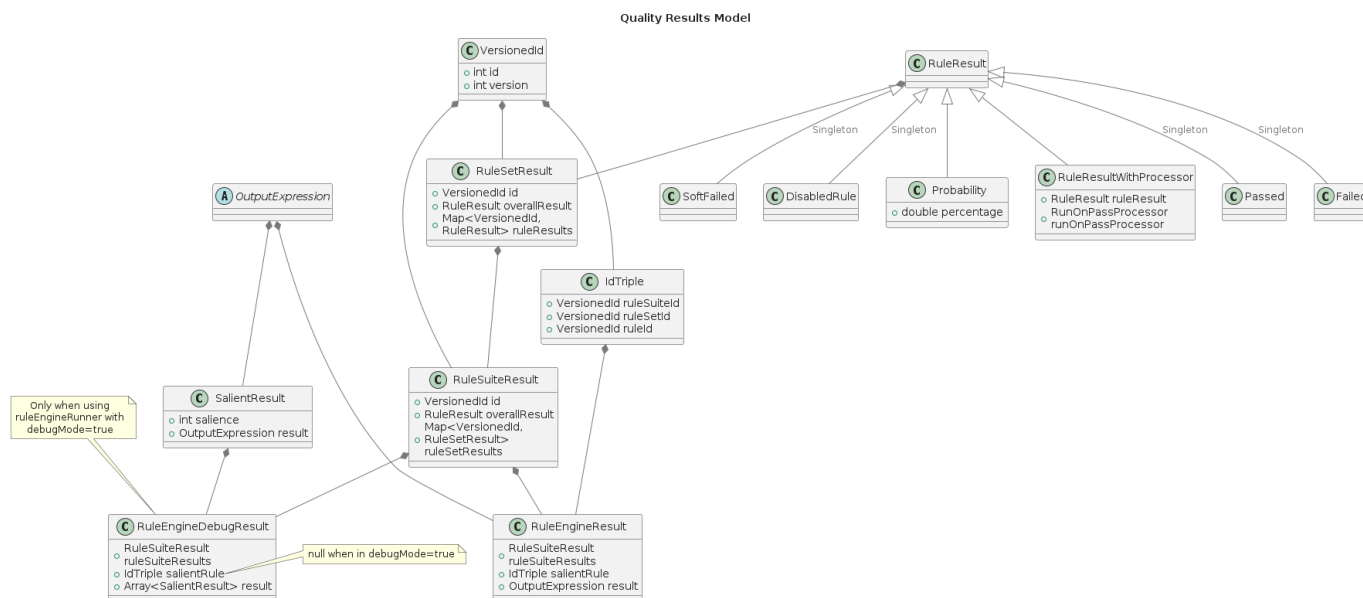
Quality RuleSuite Class Model



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.

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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

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 marshallng 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
- 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.implicit._
// 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 interegate the data quality results alongside their relevant data.

Last update: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

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.

```
// 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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

5. Advanced Usage

5.1 Bloom Filters

Bloom Filters are probabilistic 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 `smallBloom` and `bigBloom` 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("probabilityIn(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. Future versions will allow a flexible number of fields to be added to the hash function "see here" [#19](#).

- 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"smallBloom(uuid, $numRows, 0.01)")).head()
val thebytes = aggrow.getAs[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"bigBloom($bloomOn, $expectedSize, $fpp, '$bloomId')").head.getAs[Array[Byte]](0)
val bloom = com.sparkutils.quality.impl.bloom.parquet.BucketedFiles.deserialize(interim)
bloom.cleanupOthers()
bloom
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 Expressions which take expression parameters

- probabilityIn(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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

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 guarantees in memory operation only once they are loaded, no merges or joins required. However for larger data lookups either [Bloom Filters](#) should be preferred or simply use joins.

5.2.1 Building the Lookup Maps

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, new Column("country"), functions.expr("struct(funnycheck, ccy)"))
  } ),
  "ccyRate" -> ( () => {
    val df = ccyRate.toDF("ccy", "rate")
    (df, new Column("ccy"), new 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.

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.

5.2.2 Expressions which take expression parameters

- `mapLookup('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("mapLookup('countryCode', country).ccy")).show()
```

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

Last update: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

5.3 Aggregation Functions

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

At it's core all aggregations are formed using any number of `aggExpr` columns:

- `aggExpr(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("aggExpr(id % 2 > 0, sumWith(sum -> sum + id),
  resultsWith( (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 `aggExpr` as an optional default for any type other than bigint.

5.3.1 Aggregation Lambda Functions

- `sumWith(lambda entry -> entry)` - processes for each matched row the lambda with the given ddl type which defaults to LongType
- `resultsWith(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("aggExpr(id % 2 > 0, inc(id), meanF() )").as("aggExpr"))
res.show() // will show aggExpr with 10.0 as a result, sum + count would show 110..
```

- `mapWith(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("aggExpr('MAP<STRING, LONG>', 1 > 0, mapWith(date || ', ' || product, entry -> entry + 1 ), resultsWith( (sum, count) -> sum ) )").as("mapCountExpr")
// a summing example expr with embedded if's in the summing lambda for added fun
expr("aggExpr('MAP<STRING, DOUBLE>', 1 > 0, mapWith(date || ', ' || product, entry -> entry + IF(ccy='CHF', value, value * ccyrates) ),
  returnSum() )").as("mapSumExpr")
```

- `returnSum()` - just returns the sum and ignores the count param, expands to `resultsWith((sum, count) -> sum)`

5.3.2 What about my pre 0.7.1 aggExpr functions using ddl type parameters?

Prior to 0.7.1 functions such as `sumWith`, `mapWith`, `returnSum` and `returnWith` each had their own ddl parameters.

0.7.1 has moved this to the optional first parameter of `aggExpr` itself, this both reduces duplication and is less error-prone (stopping different sum types in both sum and result).

The syntax is backwards compatible however (with the exception of decimal handling), whilst the 'evaluate' (e.g. `returnSum`, `returnWith`) ddl type parameters are ignored the 'sum' type parameter is used for the whole of `aggExpr`'s 'sum type'.

If you are using the deprecated `sumWith('ddl type', ..)` and get differing types issues move the ddl to the first param of `aggExpr`.

5.3.3 I get a strange error mentioning casts and type incompatibility - what do?

In order to support the simplified single DDL parameter there are a number of Spark Expression tree re-writes taking place to 'inject' the right type. These re-writes depend on a fixed format, this may change between Spark runtimes but they may also not work beyond the use cases they are tested against (see [AggregatesTest.scala](#) for the cases).

Spark creates different plans and Expression trees from the simplified vs. the pre 0.7.1 versions, this could lead to unexpected re-write issues.

If an sql was working pre 0.7.1 with the deprecated syntax but fails with the simplified or indeed you simply wish to test out if the previous syntax would have worked you can supply 'NO_REWRITE' for the first parameter of aggExpr (instead of DDL) in addition to supplying the other two DDL's directly. The ability to provide types is present for inc, meanF, returnSum and returnWith as before.

Note

inc('DDL', expression) does not work with NO_REWRITE, as such it throws an exception telling you to use the default approach. You can use an attribute directly with NO_REWRITE just not expressions

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

aggExpr, mapWith, sumWith and returnSum 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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

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 ccyrates from the data frame are provided to the function as parameters theValue and ccy:

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

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.

5.4.1 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, ccyrates)", Id(1,2))
val rule1 = LambdaFunction("multValCCY", "theValue -> multValCCY(theValue, ccyrates)", 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, ccyrates)"))
```

5.4.2 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.

In 0.7.1 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.implicitly._

// 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 arities
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.implicitly._

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.implicit._

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).

The second test

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.

5.4.3 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:

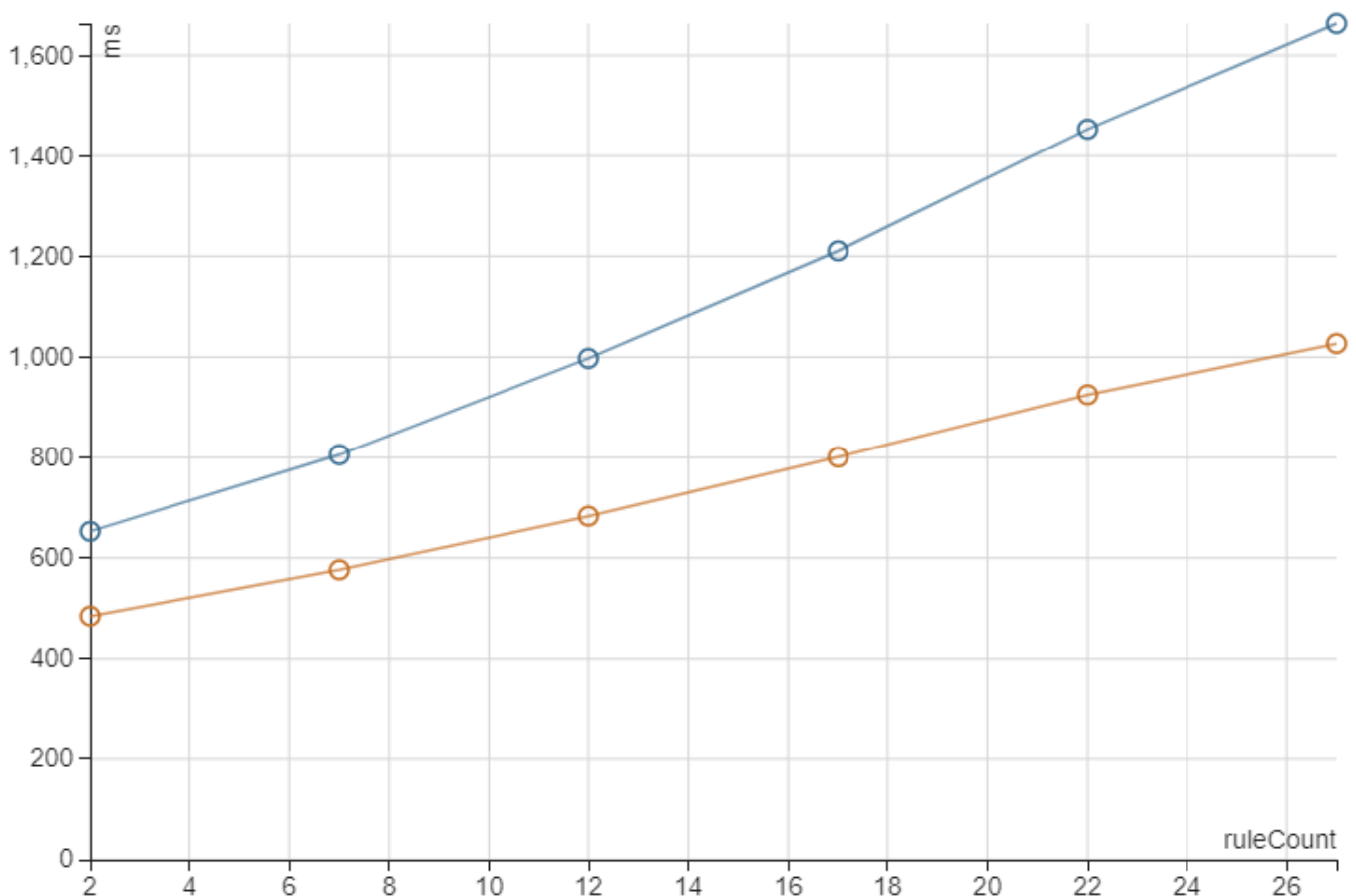
```
-Dquality.lambdaHandlers=org.apache.spark.sql.qualityFunctions.FunN=org.apache.spark.sql.qualityFunctions.DoCodegenFallbackHandler
```

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
```

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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

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 partition ignoring splittable / jumpable algorithms

The Quality pseudorandom 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 overlapping intervals across the cluster.

5.5.1 RNG Expressions

- `rngBytes([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.
- `rngUUID(expr)` - processes `expr` with either byte arrays or two longs into a UUID string, it's counterpart [longPairFromUUID](#) generates two longs

Last update: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

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")
```

- `rngID('prefix')` - generates a Random 128bit number with each column name prefixed for easy extraction
- `uniqueID('prefix')` - generates a unique 160bit ID with each column name prefixed for easy extraction
- `fieldBasedID('prefix', 'messagedigest', exp1, exp2, *)` - generates a digest based e.g. 'MD5' identifier based on an expression list
- `providedID('prefix', longArrayBasedExpression)` - generates a providedID based on supplied array of two longs expression
- `murmur3ID('prefix', exp1, exp2, *)` - generates an ID using hashes based on a version of murmur3 - not cryptographically secure but fast
- `idEqual('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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

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.

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.expr("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 ruleEngineSaliency 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<`saliency`: INTEGER, `result`: ARRAY<ORIGINALRESULTTYPE>>
```

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 saliency but no result)

flattenRuleResults

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

This sql function behaves the same way as per flattenResults, 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 showing or writing
3. Spark is free to optimise other actions than just count, ymmv in which ones work.

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 evaluations 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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

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, choose 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



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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

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.

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.

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

```
val ruleResultProcessor =
  RunOnPassProcessor(salience, Id(outputId, outputVersion),
    RuleLogicUtils.expr("thecurrent -> updateField(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 'current' for a variable on 2.4

It may be tempting to use 'current' as your lambda variable name, but this causes problems on 2.4 - every other version doesn't care.

Don't use `resolveWith` on 2.4

2.4 will NPE using `withResolve`, this does not occur on more recent Spark versions

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'), amount = 5 )

currentResult -> updateField(currentResult, 'account', concat(currentResult.account, '_suffix'), 'amount', 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 flattenFolderResults

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

This sql function behaves the same way as per flattenRuleResults 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.

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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

5.9 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:

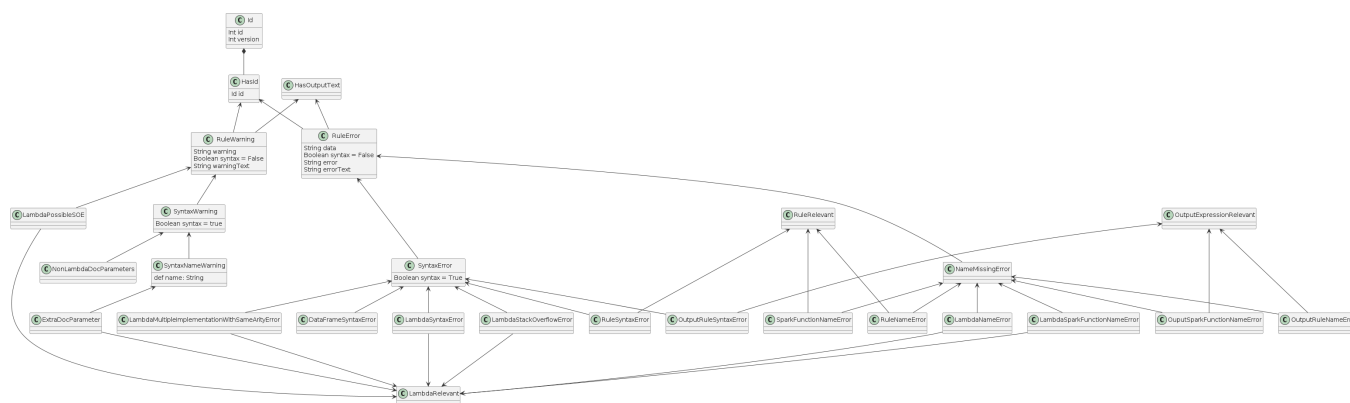
1. Schema Based - The schema representing your dictionary
2. DataFrame Based - Use an actual DataFrame to provide your dictionary

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.

5.9.1 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", recursiveLambdasOk: 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, which 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](#).

5.9.2 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.

5.9.3 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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

5.10 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: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05

6. SQL Functions Documentation

6.1 `_`

`_([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.

6.2 `_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

6.3 `aggExpr`

`aggExpr([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 `aggExpr`'s in a single pass select, use the first parameter to thread DDL type information through to the sum and result functions.

6.4 `as_uuid`

`as_uuid(lower_long, higher_long)` converts two longs into a uuid, equivalent to `rngUUID(longPair(lower, higher))`

6.5 `bigBloom`

`bigBloom(buildFrom, expectedSize, expectedFPP, 'bloom_id')` creates an aggregated bloom filter using the buildFrom expression.

The blooms are stored on a shared filesystem using the bloom_id, 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.

6.6 `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.

6.7 `coalesceIfAttributesMissing`

`coalesceIfAttributesMissing(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..
```

6.8 coalesceIfAttributesMissingDisable

`coalesceIfAttributesMissingDisable(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..
```

6.9 comparableMaps

`comparableMaps(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 ordering logic but allows for extension during the calls to the `registerQualityFunctions` via the `mapCompare` parameter and the `defaultMapCompare` function.

6.10 digestToLongs

`digestToLongs('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

6.11 digestToLongsStruct

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

6.12 disabledRule

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

6.13 failed

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

6.14 fieldBasedID

`fieldBasedID('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

6.15 flattenResults

`flattenResults(dataQualityExpr)` expands data quality results into a flat array

6.16 flattenRuleResults

flattenRuleResults(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

6.17 hashFieldBasedID

hashFieldBasedID('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

6.18 hashWith

hashWith('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.

6.19 hashWithStruct

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

6.20 idEqual

idEqual(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

6.21 inc

inc() increments the current sum by 1

Alternatives:

inc(x) use an expression of type Long to increment

6.22 longPair

longPair(lower, higher) creates a structure with these lower and higher longs

6.23 longPairEqual

longPairEqual(leftPrefix, rightPrefix) takes two prefixes which will be used to match leftPrefix_lower = rightPrefix_lower and leftPrefix_higher = rightPrefix_higher

6.24 longPairFromUUID

longPairFromUUID(expr) converts a UUID to a structure with lower and higher longs

6.25 mapContains

mapContains(expr, 'mapid') returns true if there is an item in the map

6.26 mapLookup

mapLookup(expr, 'mapid') returns either the lookup in map specified by mapid or null

6.27 meanF

meanF() simple mean on the results, expecting sum and count type Long

6.28 murmur3ID

murmur3ID('prefix', fields*) Generates a 160bit id using murmur3 hashing over input fields, prefix is used with the _base, _i0 and _i1 fields in the resulting structure

6.29 packInts

packInts(lower, higher) a packaged long from two ints, used within result compression

6.30 passed

passed() returns the Passed Integer for use in filtering: 10000

6.31 prefixedToLongPair

prefixedToLongPair(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.

6.32 printCode

printCode([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.

!!! "information" 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.

6.33 printExpr

printExpr([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).

6.34 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

6.35 probabilityIn

probabilityIn(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.

6.36 providedID

providedID('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

6.37 resultsWith

resultsWith(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:

resultsWith([sum ddl type], x) Use the given ddl type for the sum type e.g. 'MAP<STRING, DOUBLE>'

resultsWith([sum ddl type], [result ddl type], x) Use the given ddl type for the sum and result types

6.38 returnSum

returnSum(sum type ddl) just returns the sum and ignores the count param, expands to resultsWith([sum ddl_type], (sum, count) -> sum)

6.39 reverseComparableMaps

reverses a call to comparableMaps

6.40 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

6.41 rngBytes

rngBytes() Generates a 128bit random id using XO_RO_SHI_RO_128_PP, encoded as a byte array

Alternatives:

rngBytes('algorithm') Uses Commons RNG RandomSource to implement the RNG
 rngBytes('algorithm', seedL) Uses Commons RNG RandomSource to implement the RNG with a long seed
 rngBytes('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)

6.42 rngID

rngID('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:

rngId('prefix', 'algorithm') Uses Commons RNG RandomSource to implement the RNG, using other algorithm's may generate more long _iN fields
 rngId('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

6.43 rngUUID

rngUUID(expr) takes either a structure with lower and higher longs or a 128bit binary type and converts to a string uuid

6.44 ruleSuiteResultDetails

ruleSuiteResultDetails(dq) strips the overallResult from the dataquality results, suitable for keeping overall result as a top-level field with associated performance improvements

6.45 saferLongPair

deprecated use uniqueId - saferLongPair(expr, 'bloomid') Prefer to use uniqueID, this 'safer' rng repeatedly calls the expr rng function until there is no matching entry in the bloom id. It returns lower and higher longs.

6.46 smallBloom

smallBloom(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.

6.47 softFail

softFail(ruleexpr) will treat any rule failure (e.g. failed()) as returning softFailed()

6.48 softFailed

softFailed() returns the SoftFailed Integer result (-1) for use in filtering

6.49 sumWith

sumWith(x) adds expression x for each row processed in an aggExpr with a default of LongType

Alternatives:

sumWith([ddl type], x) Use the given ddl type e.g. 'MAP<STRING, DOUBLE>'

6.50 uniqueID

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

6.51 unpack

unpack(expr) takes a packed rule long and unpacks it to a .id and .version structure

6.52 unpackIdTriple

unpackIdTriple(expr) takes a packed rule triple of longs (ruleSuiteId, ruleSetId and ruleId) and unpacks it to (ruleSuiteId, ruleSuiteVersion, ruleSetId, ruleSetVersion, ruleId, ruleVersion)

6.53 updateField

updateField(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 wrapped an almost verbatim version of [Make Structs Easier' AddFields](#)

6.54 zaFieldBasedID

zaFieldBasedID('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

6.55 zaHashLongsWith

zaHashLongsWith('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

6.56 zaHashLongsWithStruct

similar to zaHashLongsWith('HASH', fields*) but generates an ID relevant multi length long struct, which is not suitable for blooms

6.57 zaHashWith

zaHashWith('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

6.58 zaHashWithStruct

similar to zaHashWith('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

6.59 zaLongsFieldBasedID

zaLongsFieldBasedID('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.

Last update: May 3, 2023 17:38:05

Created: May 3, 2023 17:38:05