Automating Spark (and Pipeline) Upgrades

While "Testing" in Production

Who am I?

- My name is Holden Karau
- Pronouns are she/her
- Apache Spark PMC (think committer with tenure)
- previously Apple, IBM, Alpine, Databricks, Google, Foursquare & Amazon
- My employer is hiring (Netflix)
- co-author of High Performance Spark, Learning Spark, and Kubeflow for Machine Learning, Scaling Python With {Ray, Dask}
- Twitter: @holdenkarau, bluesky holdenkarau.com, mastodon @holden@tech.lgbt
- OOS Livestreams: https://youtube.com/user/holdenkarau
- Github https://github.com/holdenk
- Dog mom

Our Problems

- We have unsupported versions of our data tools in production
- When things go wrong, I don't remember what we did ~5 months ago let alone ~5 years ago
- They often seam to go wrong when I'm trying to focus or sleep
- Spark 2 is very much EOLd, Spark 4 is coming soon

Why do we have these problems?

- Keeping code up to date is not a lot of fun
- Backporting is not fun
- Most humans prefer to do fun things (candy over say taxes)
- A lot of data pipelines are not very well tested
- Software is not "built to last" as they say (planned EOL etc.)
- Some of our data pipelines can have real world impacts when they go wrong

How can we work around our problem?

Software:

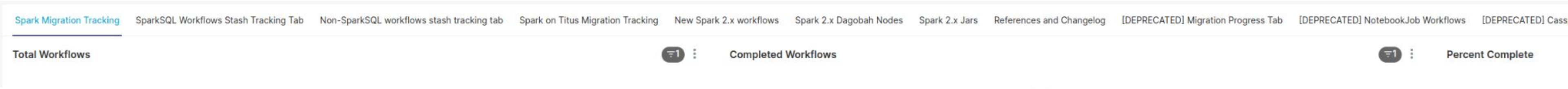
- Automated Code Update Tools
  - (AST transforms, or regexes both are fine)
- Generated Tests
- Automated Testing and Validation

Social:

- Increase visibility of out of date code & change incentives

Ok social first:

- People are way harder than computers
- We gave a deadline (and slipped) like a "normal" project
- Created visibility
- Found org champions



Spark Migration Tracking
SparkSQL Workflows Stash Tracking Tab
Non-SparkSQL workflows stash tracking tab
Spark on Titus Migration Tracking
New Spark 2.x workflows
Spark 2.x Dagobah Nodes
Spark 2.x Jars
References and Changelog
[DEPRECATED] Migration Progress Tab
[DEPRECATED] NotebookJob Workflows
[DEPRECATED] Cass
Total Workflows
Completed Workflows
Percent Complete

Spark Migration Newsletter

I have a problem, let's fix it with computers

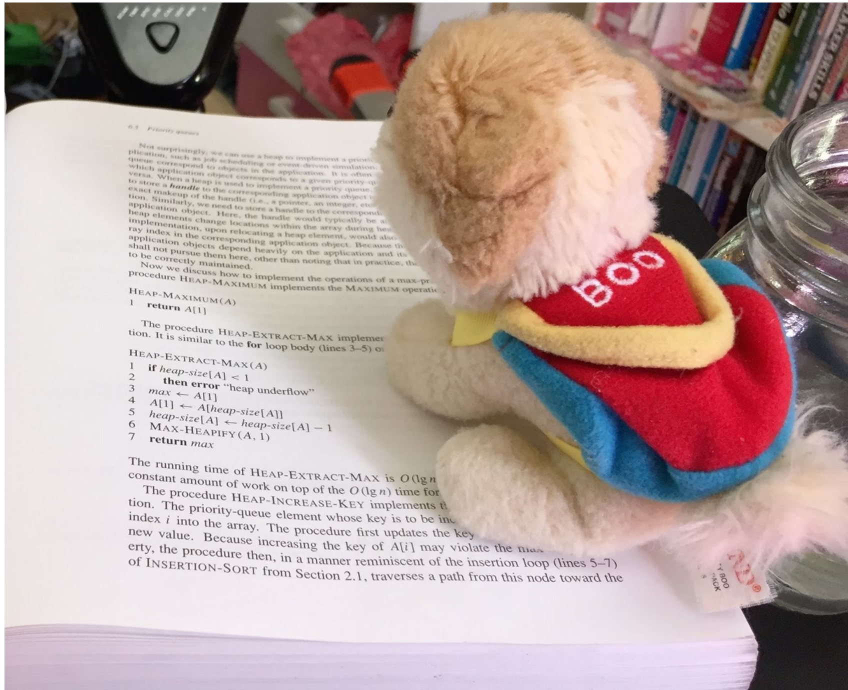
- "I played a role in helping ... become the deprecation-happy prima donnas that they are today, when I built Grok, which is a source-code understanding engine that facilitates automation and tooling on source code itself" https://medium.com/@steve.yegge/dear-google-cloud-your-deprecation-policy-is-killing-you-ee7525dc05dc
- Oh hey that sounds familiar
- Wait "killing you" -- that doesn't sound good -- w/e

Code Update Tools

- Generally not regular expressions. Buuuut….
- Scala: ScalaFix
- Python: PySparkler
- SQL: SQLFluff
- Java: (skipped, we didn't have that many)

How do you figure out the rules to make?

- Release notes (incomplete)
- MIMA changes (soooo many)
- Try and see what's broken :p (aka YOLO)



What do some rules look like?

- Let's just look at Scala & SQL

What do they look like [Scala]

```
override def fix(implicit doc: SemanticDocument): Patch = {
    val readerMatcher =
SymbolMatcher.normalized("org.apache.spark.sql.DataFrameReader")
    val jsonReaderMatcher =
SymbolMatcher.normalized("org.apache.spark.sql.DataFrameReader.json")
    val utils = new Utils()

    def matchOnTree(e: Tree): Patch = {
      e match {
        case ns @ Term.Apply(jsonReaderMatcher(reader), List(param)) =>
```

What do they look like [Scala] (continued)

```
          param match {
            case utils.rddMatcher(rdd) =>
              (Patch.addLeft(rdd, "session.createDataset(") +
Patch.addRight(rdd, ")(Encoders.STRING)") +

utils.addImportIfNotPresent(importer"org.apache.spark.sql.Encoders"))
            case _ =>
              Patch.empty
          }
```

What do they look like [Scala] (continued)

```
        case elem @ _ =>
          elem.children match {
            case Nil => Patch.empty
            case _ => elem.children.map(matchOnTree).asPatch
          }
      }
    }
    matchOnTree(doc.tree)
  }
```

SQL rules: It's like an AST transform but…. eh

```
def _eval(self, context: RuleContext) -> Optional[LintResult]:

      functional_context = FunctionalContext(context)

      children = functional_context.segment.children()

      function_name_id_seg = (

          children.first(sp.is_type("function_name"))

          .children()

          .first(sp.is_type("function_name_identifier"))[0]

      )
```

SQL rules: It's like an AST transform but…. eh

```
raw_function_name = function_name_id_seg.raw.upper().strip()

function_name = raw_function_name.upper().strip()

bracketed_segments = children.first(sp.is_type("bracketed"))

if function_name == "APPROX_PERCENTILE" or function_name == "PERCENTILE_APPROX":

    expression_count = 0

    expression_segment = None

    # Find "middle" of the approx_percentile(bloop) (e.g. bloop)

    for segment in bracketed_segments.children().iterate_segments(

        sp.is_type("expression")

    ):
```

SQL rules: It's like an AST transform but…. eh

```
        expression_count += 1
        if expression_count == 3:
            expression_segment = segment
    if expression_segment is not None:
        expression_child = expression_segment.children().first()
        # cast can either be a keyword or a function depending on if were iterating on
        # parsed on updated code.
        if expression_child[0].type == "keyword":
            if expression_child.child[0].raw == "cast":
                return None
```

SQL rules: It's like an AST transform but…. eh

```
elif expression_child[0].type == "function":
    function_name_id_seg = (
        expression_child.children()
        .first(sp.is_type("function_name"))
        .children()
        .first(sp.is_type("function_name_identifier"))[0]
    )
```

SQL rules: It's like an AST transform but…. eh

```
    raw_function_name = function_name_id_seg.raw.upper().strip()

    function_name = raw_function_name.upper().strip()

    # If we see a cast then we know this was already fixed.

    if function_name == "CAST":

        return None

expression_child = expression_child[0]
```

SQL rules: It's like an AST transform but…. eh

```
edits = [
    KeywordSegment("cast"),
    SymbolSegment("(", type="start_bracket"),
    expression_child,
    WhitespaceSegment(),
    KeywordSegment("as"),
    WhitespaceSegment(),
    KeywordSegment("int"),
    SymbolSegment(")", type="end_bracket"),
 ]
```

How do we know if it worked?

- Hope is not a plan
- Tests? (See https://github.com/holdenk/spark-testing-base)
- lakeFS or Iceberg + side by side runs https://github.com/holdenk/spark-upgrade/tree/main/pipelinecompare
  - An extension of the WAP pattern – see Michelle Winters from Netflix in her talk "Whoops the Numbers are Wrong."
  - We tried to do opt-out but ended up having to do opt-in tagging
  - Added some extensions to pick up changed partitions and not validate "too large" jobs
- Validation queries
  - SodaCL
  - https://datatest.readthedocs.io/en/latest/intro/pipeline-validation.html
  - spark-expectations

Is that expensive? Does it catch everything?

- Yes
  - Beyond doubling the cost for shadow jobs comparisons themselves took substantial compute resources.
- No
  - Jobs with side effects
  - Non-deterministic jobs
  - etc.

DEMO TIME

Let's hope it does not crash. Yay!

Ok, but where doesn't this work well?

- Dependencies
  - In my super informal survey of folks the #1 reason blocking upgrade was ElasticSearch connector
- Programming language version change
  - The reality is there's a lot of Scala 2.11 code out there, our resources are focused on 2.12->2.13 migration's but folks are further back
  - Scala version change was the #2 reason blocking Spark upgrades for folks
- API changes (what this "solves") came in #3

In conclusion:

- If you want to upgrade Spark and are lazy – https://github.com/holdenk/spark-upgrade
- The good news is we haven't made a system so powerful we can change APIs without caring
- The bad news is the same
- The excellent news is: my dog is cute AF